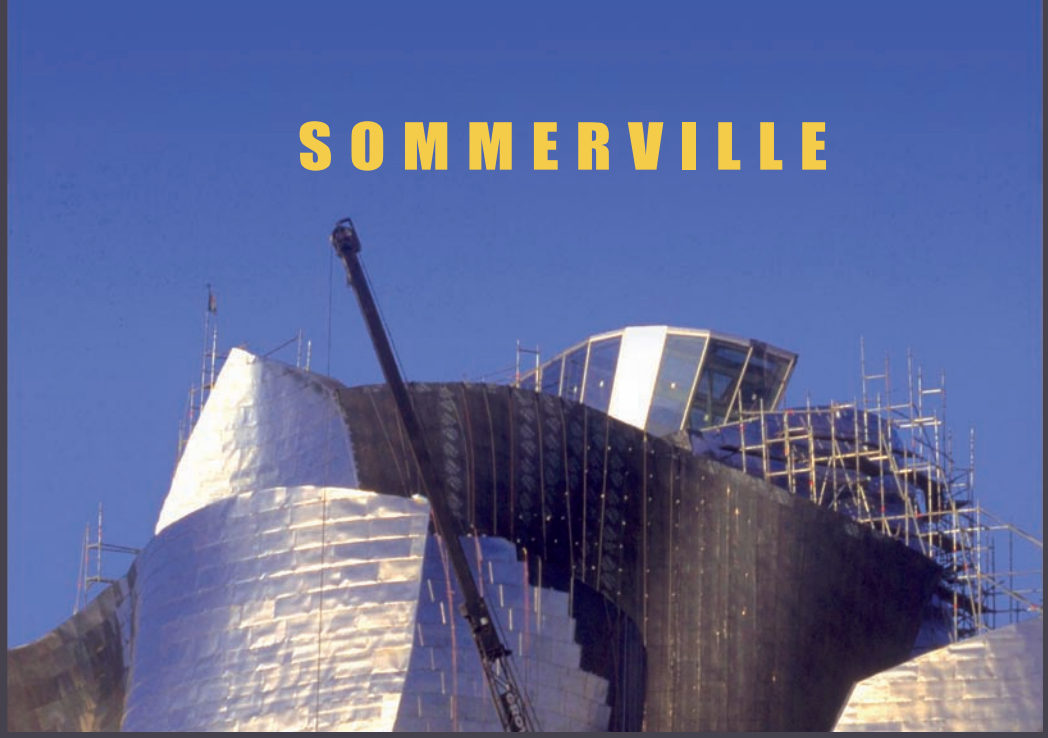
SOMMERVILLE



INGENIERÍA DE SOFTWARE

9

PEARSON

INGENIERÍA DE SOFTWARE

Novena edición

Ian Sommerville

Traducción:
Víctor Campos Olguín
Traductor especialista en Sistemas Computacionales

Revisión técnica:
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Miguel Ángel Torres Durán
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas-Instituto Politécnico Nacional, México

Darío Guillermo Cardacci
Universidad Abierta Interamericana, Buenos Aires, Argentina

Marcelo Martín Marciszack
Universidad Tecnológica Nacional, Córdoba, Argentina

Addison-Wesley

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Sommerville, Ian

Ingeniería de Software

PEARSON EDUCACIÓN, México, 2011

ISBN: 978-607-32-0603-7

Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 792

Authorized translation from the English language edition, entitled *Software engineering, 9th* edition, by *Ian Sommerville* published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2011. All rights reserved.
ISBN 9780137035151

Traducción autorizada de la edición en idioma inglés, titulada *Software engineering, 9a* edición por *Ian Sommerville* publicada por Pearson Education, Inc., publicada como Addison-Wesley, Copyright © 2011. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Editor: Luis M. Cruz Castillo
e-mail: luis.cruz@pearson.com
Editor de desarrollo: Felipe Hernández Carrasco
Supervisor de producción: Juan José García Guzmán

NOVENA EDICIÓN, 2011

D.R. © 2011 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

Addison-Wesley es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN VERSIÓN IMPRESA: 978-607-32-0603-7
ISBN VERSIÓN E-BOOK: 978-607-32-0604-4
ISBN E-CHAPTER: 978-607-32-0605-1

PRIMERA IMPRESIÓN

Impreso en México. *Printed in Mexico*.

1 2 3 4 5 6 7 8 9 0 - 14 13 12 11

Addison Wesley
es una marca de

PEARSON

PREFACIO

Mientras escribía los capítulos finales de este libro en el verano de 2009, me di cuenta de que la ingeniería de software tenía 40 años de existencia. El nombre "ingeniería de software" se propuso en 1969 en una conferencia de la Organización del Tratado del Atlántico Norte (OTAN) para analizar los problemas del desarrollo de software; en esa época había grandes sistemas de software que estaban rezagados, que no ofrecían la funcionalidad que requerían los usuarios, que costaban más de lo esperado y que no eran fiables. No asistí a dicha conferencia, pero un año después escribí mi primer programa e inicié mi vida profesional en el software.

El progreso en la ingeniería de software ha sido notable durante mi vida profesional. En la actualidad, nuestras sociedades no podrían funcionar sin grandes sistemas de software profesionales. Para construir sistemas empresariales existe una gran variedad de tecnologías (J2EE, .NET, SaaS, SAP, BPEL4WS, SOAP, CBSE, etcétera) que apoyan el desarrollo y la implementación de grandes aplicaciones empresariales. Los servicios públicos y la infraestructura nacionales (energía, comunicaciones y transporte) se apoyan en sistemas de cómputo complejos y fiables. El software ha permitido la exploración del espacio y la creación de la World Wide Web, el sistema de información más significativo en la historia de la humanidad. El mundo ahora enfrenta un nuevo conjunto de desafíos: cambio climático y temperaturas extremas, agotamiento de los recursos naturales, una creciente población mundial que demanda alimentos y vivienda, terrorismo internacional, y la necesidad de ayudar a los adultos mayores a tener vidas satisfactorias y plenas. Necesitamos nuevas tecnologías para enfrentar todos esos problemas y, desde luego, el software desempeñará un papel central en dichas tecnologías.

Por lo tanto, la ingeniería de software es una tecnología muy importante para el futuro de la humanidad. Debemos continuar educando a los ingenieros de software y desarrollar la disciplina de manera que puedan crearse sistemas de software más complejos. Desde luego, aún existen problemas con los proyectos de software. En ocasiones, el software todavía funciona con demoras y es más costoso de lo esperado. Sin embargo, no debe permitirse que dichos problemas oculten los verdaderos éxitos en la ingeniería de software, como tampoco los métodos y tecnologías impresionantes que se han desarrollado en ese campo.

La ingeniería de software es ahora una especialidad tan vasta, que sería imposible cubrir toda la materia en un libro. En consecuencia, el enfoque del presente texto se centra en los

temas clave para todos los procesos de desarrollo y, en particular, para el desarrollo de sistemas fiables. Hay un creciente énfasis en los métodos ágiles y la reutilización de software. Creo firmemente que los métodos ágiles tienen su lugar, al igual que la ingeniería de software dirigida por el plan "tradicional". Es necesario combinar lo mejor de estos enfoques para construir mejores sistemas de software.

Los libros inevitablemente reflejan las opiniones y los prejuicios de sus autores. Es probable que algunos lectores estén en desacuerdo con mis opiniones y con mi elección del material. Tal desacuerdo es un sano reflejo de la diversidad de la disciplina y es esencial para su evolución. No obstante, espero que todos los ingenieros de software y los estudiantes de esta materia puedan encontrar aquí un material de interés.

Integración con la Web

Existe una increíble cantidad de información acerca de ingeniería de software disponible en la Web, tanta, que algunas personas han cuestionado si los libros de texto como éste todavía son necesarios. Sin embargo, la calidad de la información disponible en la Web es muy irregular; en ocasiones se presenta muy mal estructurada, por lo que resulta difícil encontrar la información que se necesita. En consecuencia, creo que los libros de texto todavía desempeñan una función importante en el aprendizaje. Sirven como un mapa del campo de estudio; además, permiten organizar la información acerca de los métodos y las técnicas, y presentarla en forma coherente y legible. También ofrecen un punto de partida para efectuar una exploración más profunda de la literatura de investigación y del material disponible en la Web.

Creo firmemente que los libros de texto tienen futuro, pero sólo si están integrados y agregan valor al material en la Web. Por ello, este libro fue diseñado como un texto híbrido que combina material impreso con material publicado en la Web, pues la información central en la edición en papel se vincula con material complementario en medios electrónicos. Casi todos los capítulos incluyen "secciones Web" especialmente diseñadas, que se agregan a la información en dicho capítulo. También existen cuatro "capítulos Web" acerca de temas que no se cubrieron en la versión impresa del libro.

El sitio Web que se asocia con el libro es:

http://SoftwareEngineering-9.com

El material en el sitio Web del libro tiene cuatro componentes principales:

1. *Secciones Web* Se trata de secciones adicionales que agregan información al contenido presentado en cada capítulo. Dichas secciones Web se vinculan a partir de recuadros de conexión separados en cada capítulo.
2. *Capítulos Web* Existen cuatro capítulos Web que cubren métodos formales, diseño de interacción, documentación y arquitecturas de aplicación. Será posible agregar otros capítulos acerca de nuevos temas durante la vida del libro.
3. *Material para profesores* El material en esta sección tiene la intención de apoyar a los docentes que enseñan ingeniería de software. Véase la sección "Materiales de apoyo" en este prefacio.
4. *Estudios de caso* Ofrecen información adicional acerca de los estudios de caso analizados en el libro (bomba de insulina, sistema de atención a la salud mental,

sistema de clima selvático), así como información acerca de más estudios de caso, como la falla del cohete Ariane 5.

Paralelamente a estas secciones, también existen vínculos a otros sitios con material útil acerca de ingeniería de software: lecturas complementarias, blogs, boletines, etcétera.

Doy la bienvenida a sus comentarios y sugerencias acerca del libro y el sitio Web, en la dirección electrónica ian@SoftwareEngineering-9.com. Por favor, anote [SE9] en el asunto de su mensaje. De otro modo, mis filtros de spam probablemente rechazarán su mensaje y usted no recibirá una respuesta. Cabe aclarar que éste no es un espacio para resolver dudas de los estudiantes en relación con sus tareas escolares; sólo es un medio para comunicar comentarios y sugerencias sobre el texto.

Círculo de lectores

El libro se dirige principalmente a estudiantes universitarios y de niveles superiores, que están inscritos en cursos introductorios y avanzados de ingeniería de software y sistemas. Los ingenieros de software en la industria encontrarán el libro útil como lectura general y para actualizar sus conocimientos acerca de temas como reutilización de software, diseño arquitectónico, confiabilidad, seguridad, y mejora de procesos. Se parte de la suposición de que los lectores completaron un curso de introducción a la programación y están familiarizados con la terminología del tema.

Cambios respecto a ediciones anteriores

Esta edición conserva el material fundamental acerca de ingeniería de software que se cubrió en ediciones anteriores, pero todos los capítulos fueron revisados y actualizados, y se incluye nuevo material acerca de muchos temas diferentes. Los cambios más importantes son:

1. Se dejó atrás el enfoque centrado exclusivamente en un libro impreso, para dar paso a un enfoque híbrido que incluye un texto impreso y material Web firmemente integrado con las secciones del libro. Esto permitió reducir el número de capítulos en la versión impresa y enfocarse en material clave en cada capítulo.
2. Se realizó una reestructuración para facilitar el uso del libro en la enseñanza de la ingeniería de software. El libro consta ahora de cuatro partes en vez de ocho, y cada una puede usarse de manera independiente o en combinación con otras como la base de un curso de ingeniería de software. Las cuatro partes son: Introducción a la ingeniería de software, Confiabilidad y seguridad, Ingeniería de software avanzada y Gestión de ingeniería de software.
3. Muchos temas de las ediciones anteriores se presentan de manera más concisa en un solo capítulo, con material adicional trasladado a la Web.
4. Algunos capítulos adicionales, basados en capítulos de ediciones anteriores que no se incluyen aquí, están disponibles en la Web.

5. Se actualizó y revisó el contenido de todos los capítulos. Entre el 30 y 40% del texto se rescribió por completo.
6. Se agregaron nuevos capítulos acerca del desarrollo de software ágil y sistemas embebidos.
7. Además de esos nuevos capítulos, hay nuevo material acerca de ingeniería dirigida por modelo, el desarrollo de fuente abierta, el desarrollo dirigido por pruebas, el modelo de queso suizo de Reason, las arquitecturas de sistemas confiables, el análisis estático y la comprobación de modelos, la reutilización COTS, el software como servicio y la planeación ágil.
8. En muchos capítulos se incorporó un nuevo estudio de caso acerca de un sistema de registro de pacientes que se someten a tratamiento para problemas de salud mental.

Uso del libro para la enseñanza

El libro está diseñado de forma que pueda utilizarse en tres tipos diferentes de cursos de ingeniería de software:

1. *Cursos introductorios generales acerca de ingeniería de software* La primera parte del libro se diseñó explícitamente para apoyar un curso de un semestre de introducción a la ingeniería de software.
2. *Cursos introductorios o intermedios acerca de temas específicos de ingeniería de software* Es posible crear varios cursos más avanzados con los capítulos de las partes 2 a 4. Por ejemplo, yo impartí un curso acerca de ingeniería de sistemas críticos usando los capítulos de la parte 2, junto con los capítulos acerca de administración de la calidad y administración de la configuración.
3. *Cursos más avanzados acerca de temas específicos de ingeniería de software* En este caso, los capítulos del libro constituyen un fundamento para el curso. Luego, éste se complementa con lecturas que exploran el tema con mayor detalle. Por ejemplo, un curso acerca de reutilización de software podría basarse en los capítulos 16, 17, 18 y 19.

En el sitio Web del libro está disponible más información acerca del uso del libro para la enseñanza, incluida una comparación con ediciones anteriores.

Materiales de apoyo

Una gran variedad de materiales de apoyo está disponible para ayudar a los profesores que usen el libro para impartir sus cursos de ingeniería de software. Entre ellos se incluyen:

- Presentaciones PowerPoint para todos los capítulos del libro.
- Figuras en PowerPoint.

- Una guía para el profesor que da consejos acerca de cómo usar el libro en diferentes cursos, y que explica la relación entre los capítulos de esta edición y ediciones anteriores.
- Más información acerca de los estudios de caso del libro.
- Estudios de caso adicionales que pueden usarse en cursos de ingeniería de software.
- Presentaciones PowerPoint adicionales acerca de ingeniería de sistemas.
- Cuatro capítulos Web que cubren métodos formales, diseño de interacción, arquitecturas de aplicación y documentación.

Todo ese material está disponible de manera gratuita para los usuarios en el sitio Web del libro o en el sitio de apoyo de Pearson que se menciona más adelante. Material adicional para los maestros está disponible de manera restringida solamente para profesores acreditados:

- Respuestas modelo para ejercicios seleccionados de fin de capítulo.
- Preguntas y respuestas de exámenes para cada capítulo.

Todo el material de apoyo, incluido el material protegido con contraseña, está disponible en:

www.pearsoneducacion.net/sommerville

Los profesores que usen el libro para sus cursos pueden obtener una contraseña para ingresar al material restringido al registrarse en el sitio Web de Pearson o al ponerse en contacto con su representante local de Pearson. El autor no proporciona las contraseñas.

Reconocimientos

Muchas personas contribuyeron a través de los años a la evolución de este libro. Quisiera agradecer a todos los revisores, estudiantes y usuarios del libro que comentaron ediciones anteriores, e hicieron sugerencias constructivas para realizar cambios.

En particular, quiero agradecer a mi familia (Anne, Ali y Jane) por su ayuda y apoyo mientras escribía el libro. Un enorme agradecimiento especial para mi hija Jane, quien dio muestra de su talento al corregir las pruebas de la edición. Su participación fue de muchísima ayuda, pues realizó un excelente trabajo al leer todo el libro, y marcar y corregir una gran cantidad de errores tipográficos y gramaticales.

Ian Sommerville
Octubre de 2009

Contenido breve

Prefacio iii

Parte 1 Introducción a la ingeniería de software 1

Capítulo 1 Introducción 3
Capítulo 2 Procesos de software 27
Capítulo 3 Desarrollo ágil de software 56
Capítulo 4 Ingeniería de requerimientos 82
Capítulo 5 Modelado del sistema 118
Capítulo 6 Diseño arquitectónico 147
Capítulo 7 Diseño e implementación 176
Capítulo 8 Pruebas de software 205
Capítulo 9 Evolución del software 234

Parte 2 Confiabilidad y seguridad 261

Capítulo 10 Sistemas sociotécnicos 263
Capítulo 11 Confiabilidad y seguridad 289
Capítulo 12 Especificación de confiabilidad y seguridad 309
Capítulo 13 Ingeniería de confiabilidad 341
Capítulo 14 Ingeniería de seguridad 366
Capítulo 15 Garantía de confiabilidad y seguridad 393

Parte 3 Ingeniería de software avanzada 423

Capítulo 16 Reutilización de software 425
Capítulo 17 Ingeniería de software basada en componentes 452
Capítulo 18 Ingeniería de software distribuido 479
Capítulo 19 Arquitectura orientada a servicios 508
Capítulo 20 Software embebido 537
Capítulo 21 Ingeniería de software orientada a aspectos 565

Parte 4 Gestión de software 591

Capítulo 22 Gestión de proyectos 593
Capítulo 23 Planeación de proyectos 618
Capítulo 24 Gestión de la calidad 651
Capítulo 25 Administración de la configuración 681
Capítulo 26 Mejora de procesos 705

Glosario 733
Índice analítico 749
Índice de autores 767

CONTENIDO

Prefacio iii

Parte 1 Introducción a la ingeniería de software 1

Capítulo 1 Introducción 3

1.1 Desarrollo de software profesional 5
1.2 Ética en la ingeniería de software 14
1.3 Estudios de caso 17

Capítulo 2 Procesos de software 27

2.1 Modelos de proceso de software 29
2.2 Actividades del proceso 36
2.3 Cómo enfrentar el cambio 43
2.4 El Proceso Unificado Racional 50

Capítulo 3 Desarrollo ágil de software 56

3.1 Métodos ágiles 58
3.2 Desarrollo dirigido por un plan y desarrollo ágil 62

3.3 Programación extrema 64
3.4 Administración de un proyecto ágil 72
3.5 Escalamiento de métodos ágiles 74

Capítulo 4 Ingeniería de requerimientos 82

4.1 Requerimientos funcionales y no funcionales 84
4.2 El documento de requerimientos de software 91
4.3 Especificación de requerimientos 94
4.4 Procesos de ingeniería de requerimientos 99
4.5 Adquisición y análisis de requerimientos 100
4.6 Validación de requerimientos 110
4.7 Administración de requerimientos 111

Capítulo 5 Modelado del sistema 118

5.1 Modelos de contexto 121
5.2 Modelos de interacción 124
5.3 Modelos estructurales 129
5.4 Modelos de comportamiento 133
5.5 Ingeniería dirigida por modelo 138

Capítulo 6 Diseño arquitectónico 147

6.1 Decisiones en el diseño arquitectónico 151
6.2 Vistas arquitectónicas 153
6.3 Patrones arquitectónicos 155
6.4 Arquitecturas de aplicación 164

Capítulo 7 Diseño e implementación 176

7.1 Diseño orientado a objetos con el uso del UML 178
7.2 Patrones de diseño 189

7.3 Conflictos de implementación 193
7.4 Desarrollo de código abierto 198

Capítulo 8 Pruebas de software 205

8.1 Pruebas de desarrollo 210
8.2 Desarrollo dirigido por pruebas 221
8.3 Pruebas de versión 224
8.4 Pruebas de usuario 228

Capítulo 9 Evolución del software 234

9.1 Procesos de evolución 237
9.2 Evolución dinámica del programa 240
9.3 Mantenimiento del software 242
9.4 Administración de sistemas heredados 252

Parte 2 Confiabilidad y seguridad 261

Capítulo 10 Sistemas sociotécnicos 263

10.1 Sistemas complejos 266
10.2 Ingeniería de sistemas 273
10.3 Procuración del sistema 275
10.4 Desarrollo del sistema 278
10.5 Operación del sistema 281

Capítulo 11 Confiabilidad y seguridad 289

11.1 Propiedades de confiabilidad 291
11.2 Disponibilidad y fiabilidad 295
11.3 Protección 299
11.4 Seguridad 302

Capítulo 12 Especificación de confiabilidad y seguridad 309

12.1 Especificación de requerimientos dirigida por riesgos 311
12.2 Especificación de protección 313
12.3 Especificación de fiabilidad 320
12.4 Especificación de seguridad 329
12.5 Especificación formal 333

Capítulo 13 Ingeniería de confiabilidad 341

13.1 Redundancia y diversidad 343
13.2 Procesos confiables 345
13.3 Arquitecturas de sistemas confiables 348
13.4 Programación confiable 355

Capítulo 14 Ingeniería de seguridad 366

14.1 Gestión del riesgo de seguridad 369
14.2 Diseño para la seguridad 375
14.3 Supervivencia del sistema 386

Capítulo 15 Garantía de confiabilidad y seguridad 393

15.1 Análisis estático 395
15.2 Pruebas de fiabilidad 401
15.3 Pruebas de seguridad 404
15.4 Aseguramiento del proceso 406
15.5 Casos de protección y confiabilidad 410

Parte 3 Ingeniería de software avanzada 423

Capítulo 16 Reutilización de software 425

16.1 Panorama de la reutilización 428
16.2 Frameworks de aplicación 431

16.3 Líneas de productos de software 434
16.4 Reutilización de productos COTS 440

Capítulo 17 Ingeniería de software basada en componentes 452

17.1 Componentes y modelos de componentes 455
17.2 Procesos CBSE 461
17.3 Composición de componentes 468

Capítulo 18 Ingeniería de software distribuido 479

18.1 Conflictos de los sistemas distribuidos 481
18.2 Computación cliente-servidor 488
18.3 Patrones arquitectónicos para sistemas distribuidos 490
18.4 Software como servicio 501

Capítulo 19 Arquitectura orientada a servicios 508

19.1 Servicios como componentes de reutilización 514
19.2 Ingeniería de servicio 518
19.3 Desarrollo de software con servicios 527

Capítulo 20 Software embebido 537

20.1 Diseño de sistemas embebidos 540
20.2 Patrones arquitectónicos 547
20.3 Análisis de temporización 554
20.4 Sistemas operativos de tiempo real 558

Capítulo 21 Ingeniería de software orientada a aspectos 565

21.1 La separación de intereses 567
21.2 Aspectos, puntos de enlaces y puntos de corte 571
21.3 Ingeniería de software con aspectos 576

Parte 4 **Gestión de software** 591

Capítulo 22 Gestión de proyectos 593

22.1 Gestión del riesgo 595
22.2 Gestión de personal 602
22.3 Trabajo en equipo 607

Capítulo 23 Planeación de proyectos 618

23.1 Fijación de precio al software 621
23.2 Desarrollo dirigido por un plan 623
23.3 Calendarización de proyectos 626
23.4 Planeación ágil 631
23.5 Técnicas de estimación 633

Capítulo 24 Gestión de la calidad 651

24.1 Calidad del software 655
24.2 Estándares de software 657
24.3 Revisiones e inspecciones 663
24.4 Medición y métricas del software 668

Capítulo 25 Administración de la configuración 681

25.1 Administración del cambio 685
25.2 Gestión de versiones 690
25.3 Construcción del sistema 693
25.4 Gestión de entregas de software (*release*) 699

Capítulo 26 Mejora de procesos 705

26.1 El proceso de mejora de procesos 708
26.2 Medición del proceso 711

26.3 Análisis del proceso 715

26.4 Cambios en los procesos 718

26.5 El marco de trabajo para la mejora de procesos CMMI 721

Glosario 733
Índice analítico 749
Índice de autores 767

PARTE 1

Introducción a la ingeniería de software

La meta de esta parte del libro es ofrecer una introducción general a la ingeniería de software. Se incluyen conceptos importantes como procesos de software y métodos ágiles; además, se describen actividades esenciales del desarrollo de software, desde la especificación inicial del software hasta la evolución del sistema. Los capítulos de esta parte se diseñaron para apoyar un curso de un semestre en ingeniería de software.

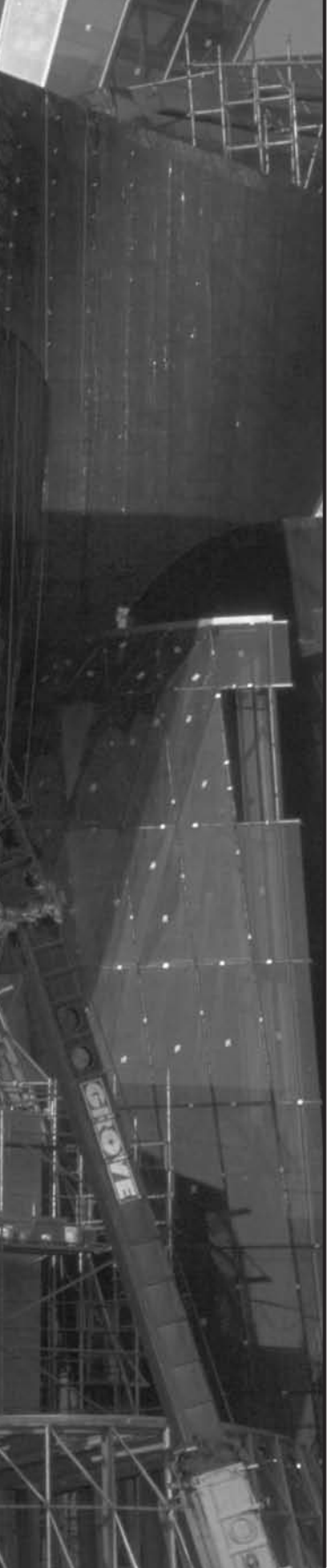
El capítulo 1 introduce al lector de manera general en la ingeniería de software profesional y define algunos conceptos al respecto. También se escribe un breve análisis de los conflictos éticos en la ingeniería de software. Es importante que los ingenieros de software consideren las numerosas implicaciones de su trabajo. Este capítulo además presenta tres estudios de caso que se usan en el libro, a saber: un sistema para administrar registros de pacientes que se someten a tratamiento por problemas de salud mental, un sistema de control para una bomba de insulina portátil y un sistema meteorológico a campo abierto.

Los capítulos 2 y 3 tratan los procesos de ingeniería de software y el desarrollo ágil. En el capítulo 2 se presentan modelos de proceso de software genérico de uso común, como el waterfall (cascada), y se estudian las actividades básicas que son parte de dichos procesos. El capítulo 3 complementa esto con un estudio de los métodos de desarrollo ágil para ingeniería de software. Se usa sobre todo la programación extrema como

ejemplo de un método ágil, aunque también en esta sección se introduce brevemente Scrum.

Los capítulos restantes son amplias descripciones de las actividades del proceso de software que se introducirán en el capítulo 2. En el capítulo 4 se trata un tema importante de la ingeniería de requerimientos, donde se definen las necesidades de lo que debe hacer un sistema. El capítulo 5 muestra el modelado de sistemas usando el UML, donde se enfoca el uso de los diagramas de caso, diagramas de clase, diagramas de secuencia y diagramas de estado para modelar un sistema de software. El capítulo 6 introduce al diseño arquitectónico y estudia la importancia de la arquitectura y el uso de patrones arquitectónicos en el diseño de software.

El capítulo 7 trata sobre el diseño orientado a objetos y el uso de patrones de diseño. Aquí también se observan importantes problemas de implementación: reutilización, manejo de configuración y el desarrollo de host-target (anfitrión destino); también se estudia el desarrollo de código abierto. El capítulo 8 se enfoca en las pruebas del software, desde la prueba de unidad durante el desarrollo del sistema, hasta la prueba de la puesta en venta del software. También se analiza el uso del desarrollo impulsado por prueba, un enfoque pionero en los métodos ágiles, pero con gran aplicabilidad. Finalmente, el capítulo 9 brinda un panorama de los temas sobre la evolución del software. Se describen los procesos evolutivos, el mantenimiento del software y la gestión de sistemas legados.



1

Introducción

Objetivos

Los objetivos de este capítulo consisten en introducir al lector a la ingeniería de software y ofrecer un marco conceptual para entender el resto del libro. Al estudiar este capítulo:

- conocerá qué es la ingeniería de software y por qué es importante;
- comprenderá que el desarrollo de diferentes tipos de sistemas de software puede requerir distintas técnicas de ingeniería de software;
- entenderá algunos conflictos éticos y profesionales que son importantes para los ingenieros de software;
- conocerá tres sistemas de diferentes tipos, que se usarán como ejemplos a lo largo del libro.

Contenido

1.1 Desarrollo de software profesional
1.2 Ética en la ingeniería de software
1.3 Estudios de caso

Es imposible operar el mundo moderno sin software. Las infraestructuras nacionales y los servicios públicos se controlan mediante sistemas basados en computadoras, y la mayoría de los productos eléctricos incluyen una computadora y un software de control. La fabricación y la distribución industrial están completamente computarizadas, como el sistema financiero. El entretenimiento, incluida la industria musical, los juegos por computadora, el cine y la televisión, usan software de manera intensiva. Por lo tanto, la ingeniería de software es esencial para el funcionamiento de las sociedades, tanto a nivel nacional como internacional.

Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales, regidos por leyes físicas ni por procesos de fabricación. Esto simplifica la ingeniería de software, pues no existen límites naturales a su potencial. Sin embargo, debido a la falta de restricciones físicas, los sistemas de software pueden volverse rápidamente muy complejos, difíciles de entender y costosos de cambiar.

Hay muchos tipos diferentes de sistemas de software, desde los simples sistemas embebidos, hasta los complejos sistemas de información mundial. No tiene sentido buscar notaciones, métodos o técnicas universales para la ingeniería de software, ya que diferentes tipos de software requieren distintos enfoques. Desarrollar un sistema organizacional de información es completamente diferente de un controlador para un instrumento científico. Ninguno de estos sistemas tiene mucho en común con un juego por computadora de gráficos intensivos. Aunque todas estas aplicaciones necesitan ingeniería de software, no todas requieren las mismas técnicas de ingeniería de software.

Aún existen muchos reportes tanto de proyectos de software que salen mal como de "fallas de software". Por ello, a la ingeniería de software se le considera inadecuada para el desarrollo del software moderno. Sin embargo, desde la perspectiva del autor, muchas de las llamadas fallas del software son consecuencia de dos factores:

1. *Demandas crecientes* Conforme las nuevas técnicas de ingeniería de software ayudan a construir sistemas más grandes y complejos, las demandas cambian. Los sistemas tienen que construirse y distribuirse más rápidamente; se requieren sistemas más grandes e incluso más complejos; los sistemas deben tener nuevas capacidades que anteriormente se consideraban imposibles. Los métodos existentes de ingeniería de software no pueden enfrentar la situación, y tienen que desarrollarse nuevas técnicas de ingeniería de software para satisfacer nuevas demandas.
2. *Expectativas bajas* Es relativamente sencillo escribir programas de cómputo sin usar métodos y técnicas de ingeniería de software. Muchas compañías se deslizan hacia la ingeniería de software conforme evolucionan sus productos y servicios. No usan métodos de ingeniería de software en su trabajo diario. Por lo tanto, su software con frecuencia es más costoso y menos confiable de lo que debiera. Es necesaria una mejor educación y capacitación en ingeniería de software para solucionar este problema.

Los ingenieros de software pueden estar orgullosos de sus logros. Desde luego, todavía se presentan problemas al desarrollar software complejo, pero, sin ingeniería de software, no se habría explorado el espacio ni se tendría Internet o las telecomunicaciones modernas. Todas las formas de viaje serían más peligrosas y caras. La ingeniería de software ha contribuido en gran medida, y sus aportaciones en el siglo XXI serán aún mayores.

Historia de la ingeniería de software

El concepto "ingeniería de software" se propuso originalmente en 1968, en una conferencia realizada para discutir lo que entonces se llamaba la "crisis del software" (Naur y Randell, 1969). Se volvió claro que los enfoques individuales al desarrollo de programas no escalaban hacia los grandes y complejos sistemas de software. Éstos no eran confiables, costaban más de lo esperado y se distribuían con demora.

A lo largo de las décadas de 1970 y 1980 se desarrolló una variedad de nuevas técnicas y métodos de ingeniería de software, tales como la programación estructurada, el encubrimiento de información y el desarrollo orientado a objetos. Se perfeccionaron herramientas y notaciones estándar y ahora se usan de manera extensa.

http://www.SoftwareEngineering-9.com/Web/History/

1.1 Desarrollo de software profesional

Muchos individuos escriben programas. En las empresas los empleados hacen programas de hoja de cálculo para simplificar su trabajo; científicos e ingenieros elaboran programas para procesar sus datos experimentales, y los aficionados crean programas para su propio interés y satisfacción. Sin embargo, la gran mayoría del desarrollo de software es una actividad profesional, donde el software se realiza para propósitos de negocios específicos, para su inclusión en otros dispositivos o como productos de software, por ejemplo, sistemas de información, sistemas de CAD, etcétera. El software profesional, destinado a usarse por alguien más aparte de su desarrollador, se lleva a cabo en general por equipos, en vez de individualmente. Se mantiene y cambia a lo largo de su vida.

La ingeniería de software busca apoyar el desarrollo de software profesional, en lugar de la programación individual. Incluye técnicas que apoyan la especificación, el diseño y la evolución del programa, ninguno de los cuales son normalmente relevantes para el desarrollo de software personal. Con el objetivo de ayudarlo a obtener una amplia visión de lo que trata la ingeniería de software, en la figura 1.1 se resumen algunas preguntas planteadas con frecuencia.

Muchos suponen que el software es tan sólo otra palabra para los programas de cómputo. No obstante, cuando se habla de ingeniería de software, esto no sólo se refiere a los programas en sí, sino también a toda la documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta. Un sistema de software desarrollado profesionalmente es usualmente más que un solo programa. El sistema por lo regular consta de un número de programas separados y archivos de configuración que se usan para instalar dichos programas. Puede incluir documentación del sistema, que describe la estructura del sistema; documentación del usuario, que explica cómo usar el sistema, y los sitios web para que los usuarios descarguen información reciente del producto.

Ésta es una de las principales diferencias entre el desarrollo de software profesional y el de aficionado. Si usted diseña un programa personal, nadie más lo usará ni tendrá que preocuparse por elaborar guías del programa, documentar el diseño del programa, etcétera. Por el contrario, si crea software que otros usarán y otros ingenieros cambiarán, entonces, en general debe ofrecer información adicional, así como el código del programa.

Pregunta	Respuesta
¿Qué es software?	Programas de cómputo y documentación asociada. Los productos de software se desarrollan para un cliente en particular o para un mercado en general.
¿Cuáles son los atributos del buen software?	El buen software debe entregar al usuario la funcionalidad y el desempeño requeridos, y debe ser sustentable, confiable y utilizable.
¿Qué es ingeniería de software?	La ingeniería de software es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software.
¿Cuáles son las actividades fundamentales de la ingeniería de software?	Especificación, desarrollo, validación y evolución del software.
¿Cuál es la diferencia entre ingeniería de software y ciencias de la computación?	Las ciencias de la computación se enfocan en teoría y fundamentos; mientras la ingeniería de software se enfoca en el sentido práctico del desarrollo y en la distribución de software.
¿Cuál es la diferencia entre ingeniería de software e ingeniería de sistemas?	La ingeniería de sistemas se interesa por todos los aspectos del desarrollo de sistemas basados en computadoras, incluidos hardware, software e ingeniería de procesos. La ingeniería de software es parte de este proceso más general.
¿Cuáles son los principales retos que enfrenta la ingeniería de software?	Se enfrentan con una diversidad creciente, demandas por tiempos de distribución limitados y desarrollo de software confiable.
¿Cuáles son los costos de la ingeniería de software?	Aproximadamente 60% de los costos del software son de desarrollo, y 40% de prueba. Para el software elaborado específicamente, los costos de evolución superan con frecuencia los costos de desarrollo.
¿Cuáles son los mejores métodos y técnicas de la ingeniería de software?	Aun cuando todos los proyectos de software deben gestionarse y desarrollarse de manera profesional, existen diferentes técnicas que son adecuadas para distintos tipos de sistema. Por ejemplo, los juegos siempre deben diseñarse usando una serie de prototipos, mientras que los sistemas críticos de control de seguridad requieren de una especificación completa y analizable para su desarrollo. Por lo tanto, no puede decirse que un método sea mejor que otro.
¿Qué diferencias ha marcado la Web a la ingeniería de software?	La Web ha llevado a la disponibilidad de servicios de software y a la posibilidad de desarrollar sistemas basados en servicios distribuidos ampliamente. El desarrollo de sistemas basados en Web ha conducido a importantes avances en lenguajes de programación y reutilización de software.

Figura 1.1 Preguntas planteadas con frecuencia sobre el software

Los ingenieros de software están interesados por el desarrollo de productos de software (es decir, software que puede venderse a un cliente). Existen dos tipos de productos de software:

1. *Productos genéricos* Consisten en sistemas independientes que se producen por una organización de desarrollo y se venden en el mercado abierto a cualquier cliente

que desee comprarlos. Ejemplos de este tipo de productos incluyen software para PC, como bases de datos, procesadores de texto, paquetes de dibujo y herramientas de administración de proyectos. También abarcan las llamadas aplicaciones verticales diseñadas para cierto propósito específico, tales como sistemas de información de librería, sistemas de contabilidad o sistemas para mantener registros dentales.

2. *Productos personalizados (o a la medida)* Son sistemas que están destinados para un cliente en particular. Un contratista de software desarrolla el programa especialmente para dicho cliente. Ejemplos de este tipo de software incluyen los sistemas de control para dispositivos electrónicos, sistemas escritos para apoyar cierto proceso empresarial y los sistemas de control de tráfico aéreo.

Una diferencia importante entre estos tipos de software es que, en productos genéricos, la organización que desarrolla el software controla la especificación del mismo. Para los productos personalizados, la organización que compra el software generalmente desarrolla y controla la especificación, por lo que los desarrolladores de software deben trabajar siguiendo dicha especificación.

Sin embargo, la distinción entre estos tipos de producto de sistemas se vuelve cada vez más difusa. Ahora, cada vez más sistemas se construyen con un producto genérico como base, que luego se adapta para ajustarse a los requerimientos de un cliente. Los sistemas Enterprise Resource Planning (ERP, planeación de recursos empresariales), como el sistema SAP, son los mejores ejemplos de este enfoque. Aquí, un sistema grande y complejo se adapta a una compañía al incorporar la información acerca de las reglas y los procesos empresariales, los reportes requeridos, etcétera.

Cuando se habla de la calidad del software profesional, se debe considerar que el software lo usan y cambian personas, además de sus desarrolladores. En consecuencia, la calidad no tiene que ver sólo con lo que hace el software. En cambio, debe incluir el comportamiento del software mientras se ejecuta, y la estructura y organización de los programas del sistema y la documentación asociada. Esto se refleja en los llamados calidad o atributos no funcionales del software. Ejemplos de dichos atributos son el tiempo de respuesta del software ante la duda de un usuario y la comprensibilidad del código del programa.

El conjunto específico de atributos que se espera de un sistema de software depende evidentemente de su aplicación. Así, un sistema bancario debe ser seguro, un juego interactivo debe tener capacidad de respuesta, un sistema de conmutación telefónica debe ser confiable, etcétera. Esto puede generalizarse en el conjunto de atributos que se muestra en la figura 1.2, los cuales consideran las características esenciales de un sistema de software profesional.

1.1.1 Ingeniería de software

La ingeniería de software es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después de que se pone en operación. En esta definición se presentan dos frases clave:

1. *Disciplina de ingeniería* Los ingenieros hacen que las cosas funcionen. Aplican teorías, métodos y herramientas donde es adecuado. Sin embargo, los usan de manera

Características del producto	Descripción
Mantenimiento	El software debe escribirse de tal forma que pueda evolucionar para satisfacer las necesidades cambiantes de los clientes. Éste es un atributo crítico porque el cambio del software es un requerimiento inevitable de un entorno empresarial variable.
Confiabilidad y seguridad	La confiabilidad del software incluye un rango de características que abarcan fiabilidad, seguridad y protección. El software confiable no tiene que causar daño físico ni económico, en caso de falla del sistema. Los usuarios malintencionados no deben tener posibilidad de acceder al sistema o dañarlo.
Eficiencia	El software no tiene que desperdiciar los recursos del sistema, como la memoria y los ciclos del procesador. Por lo tanto, la eficiencia incluye capacidad de respuesta, tiempo de procesamiento, utilización de memoria, etcétera.
Aceptabilidad	El software debe ser aceptable al tipo de usuarios para quienes se diseña. Esto significa que necesita ser comprensible, utilizable y compatible con otros sistemas que ellos usan.

Figura 1.2 Atributos esenciales del buen software

selectiva y siempre tratan de encontrar soluciones a problemas, incluso cuando no hay teorías ni métodos aplicables. Los ingenieros también reconocen que deben trabajar ante restricciones organizacionales y financieras, de modo que buscan soluciones dentro de tales limitaciones.

2. *Todos los aspectos de la producción del software* La ingeniería de software no sólo se interesa por los procesos técnicos del desarrollo de software, sino también incluye actividades como la administración del proyecto de software y el desarrollo de herramientas, así como métodos y teorías para apoyar la producción de software.

La ingeniería busca obtener resultados de la calidad requerida dentro de la fecha y del presupuesto. A menudo esto requiere contraer compromisos: los ingenieros no deben ser perfeccionistas. Sin embargo, las personas que diseñan programas para sí mismas podrían pasar tanto tiempo como deseen en el desarrollo del programa.

En general, los ingenieros de software adoptan en su trabajo un enfoque sistemático y organizado, pues usualmente ésta es la forma más efectiva de producir software de alta calidad. No obstante, la ingeniería busca seleccionar el método más adecuado para un conjunto de circunstancias y, de esta manera, un acercamiento al desarrollo más creativo y menos formal sería efectivo en ciertas situaciones. El desarrollo menos formal es particularmente adecuado para la creación de sistemas basados en la Web, que requieren una mezcla de habilidades de software y diseño gráfico.

La ingeniería de software es importante por dos razones:

1. Cada vez con mayor frecuencia, los individuos y la sociedad se apoyan en los avanzados sistemas de software. Por ende, se requiere producir económica y rápidamente sistemas confiables.

2. A menudo resulta más barato a largo plazo usar métodos y técnicas de ingeniería de software para los sistemas de software, que sólo diseñar los programas como si fuera un proyecto de programación personal. Para muchos tipos de sistemas, la mayoría de los costos consisten en cambiar el software después de ponerlo en operación.

El enfoque sistemático que se usa en la ingeniería de software se conoce en ocasiones como proceso de software. Un proceso de software es una secuencia de actividades que conducen a la elaboración de un producto de software. Existen cuatro actividades fundamentales que son comunes a todos los procesos de software, y éstas son:

1. Especificación del software, donde clientes e ingenieros definen el software que se producirá y las restricciones en su operación.
2. Desarrollo del software, donde se diseña y programa el software.
3. Validación del software, donde se verifica el software para asegurar que sea lo que el cliente requiere.
4. Evolución del software, donde se modifica el software para reflejar los requerimientos cambiantes del cliente y del mercado.

Diferentes tipos de sistemas necesitan distintos procesos de desarrollo. Por ejemplo, el software en tiempo real en una aeronave debe especificarse por completo antes de comenzar el desarrollo. En los sistemas de comercio electrónico, la especificación y el programa por lo general se desarrollan en conjunto. En consecuencia, tales actividades genéricas pueden organizarse en diferentes formas y describirse en distintos niveles de detalle, dependiendo del tipo de software que se vaya a desarrollar. En el capítulo 2 se describen con más puntualidad los procesos de software.

La ingeniería de software se relaciona con las ciencias de la computación y la ingeniería de sistemas:

1. Las ciencias de la computación se interesan por las teorías y los métodos que subyacen en las computadoras y los sistemas de software, en tanto que la ingeniería de software se preocupa por los asuntos prácticos de la producción del software. Cierto conocimiento de ciencias de la computación es esencial para los ingenieros de software, del mismo modo que cierto conocimiento de física lo es para los ingenieros electricistas. Sin embargo, con frecuencia la teoría de las ciencias de la computación es más aplicable a programas relativamente pequeños. Las teorías de las ciencias de la computación no siempre pueden aplicarse a grandes problemas complejos que requieren una solución de software.
2. La ingeniería de sistemas se interesa por todos los aspectos del desarrollo y la evolución de complejos sistemas, donde el software tiene un papel principal. Por lo tanto, la ingeniería de sistemas se preocupa por el desarrollo de hardware, el diseño de políticas y procesos, la implementación del sistema, así como por la ingeniería de software. Los ingenieros de sistemas intervienen en la especificación del sistema, definiendo su arquitectura global y, luego, integrando las diferentes partes para crear el sistema terminado. Están menos preocupados por la ingeniería de los componentes del sistema (hardware, software, etcétera).

Como se expone en la siguiente sección, hay muchos tipos diferentes de software. No existe un método o una técnica universales en la ingeniería de software que sea aplicable para todos éstos. No obstante, tres problemas generales afectan a muy diversos tipos de software:

1. *Heterogeneidad* Cada vez con mayor frecuencia se requieren sistemas que operen como distribuidos a través de redes que incluyan diferentes tipos de computadoras y dispositivos móviles. Es posible que el software se ejecute tanto en computadoras de propósito general como en teléfonos móviles. Se tendrá que integrar con frecuencia el nuevo software con sistemas legados más viejos, escritos en diferentes lenguajes de programación. El reto aquí es desarrollar técnicas para construir software confiable que sea suficientemente flexible para enfrentar esa heterogeneidad.
2. *Cambio empresarial y social* Los negocios y la sociedad cambian de manera increíblemente rápida, conforme se desarrollan las economías emergentes y nuevas tecnologías están a la disposición. Ambos necesitan tener la posibilidad de cambiar su software existente y desarrollar rápidamente uno nuevo. Muchas técnicas tradicionales de la ingeniería de software consumen tiempo, y generalmente la entrega de los nuevos sistemas tarda más de lo planeado. Requieren evolucionar de modo que se reduzca el tiempo necesario para que el software dé valor a sus clientes.
3. *Seguridad y confianza* Dado que el software está vinculado con todos los aspectos de la vida, es esencial confiar en dicho software. Esto es especialmente cierto para los sistemas de software remoto a los que se accede a través de una página Web o una interfaz de servicio Web. Es necesario asegurarse de que usuarios malintencionados no puedan atacar el software y que se conserve la seguridad de la información.

Desde luego, éstos no son problemas independientes. Por ejemplo, quizá sea necesario realizar cambios rápidos a un sistema legado con la finalidad de dotarlo con una interfaz de servicio Web. Para enfrentar dichos retos se necesitarán nuevas herramientas y técnicas, así como formas innovadoras de combinar y usar los métodos existentes de ingeniería de software.

1.1.2 Diversidad de la ingeniería de software

La ingeniería de software es un enfoque sistemático para la producción de software que toma en cuenta los temas prácticos de costo, fecha y confiabilidad, así como las necesidades de clientes y fabricantes de software. Como este enfoque sistemático realmente implementado varía de manera drástica dependiendo de la organización que desarrolla el software, el tipo de software y los individuos que intervienen en el proceso de desarrollo, no existen métodos y técnicas universales de ingeniería de software que sean adecuados para todos los sistemas y las compañías. Más bien, durante los últimos 50 años evolucionó un conjunto de métodos y herramientas de ingeniería de software.

Quizás el factor más significativo en la determinación de qué métodos y técnicas de la ingeniería de software son más importantes, es el tipo de aplicación que está siendo desarrollada. Existen muchos diferentes tipos de aplicación, incluidos los siguientes:

1. *Aplicaciones independientes* Se trata de sistemas de aplicación que corren en una computadora local, como una PC, e incluyen toda la funcionalidad necesaria

y no requieren conectarse a una red. Ejemplos de tales aplicaciones son las de oficina en una PC, programas CAD, software de manipulación de fotografías, etcétera.

2. *Aplicaciones interactivas basadas en transacción* Consisten en aplicaciones que se ejecutan en una computadora remota y a las que los usuarios acceden desde sus propias PC o terminales. Evidentemente, en ellas se incluyen aplicaciones Web como las de comercio electrónico, donde es posible interactuar con un sistema remoto para comprar bienes y servicios. Esta clase de aplicación también incluye sistemas empresariales, donde una organización brinda acceso a sus sistemas a través de un navegador Web o un programa de cliente de propósito específico y servicios basados en nube, como correo electrónico y compartición de fotografías. Las aplicaciones interactivas incorporan con frecuencia un gran almacén de datos al que se accede y actualiza en cada transacción.

3. *Sistemas de control embebido* Se trata de sistemas de control de software que regulan y gestionan dispositivos de hardware. Numéricamente, quizás existen más sistemas embebidos que cualquier otro tipo de sistema. Algunos ejemplos de sistemas embebidos incluyen el software en un teléfono móvil (celular), el software que controla los frenos antibloqueo de un automóvil y el software en un horno de microondas para controlar el proceso de cocinado.

4. *Sistemas de procesamiento en lotes* Son sistemas empresariales que se diseñan para procesar datos en grandes lotes (batch). Procesan gran cantidad de entradas individuales para crear salidas correspondientes. Los ejemplos de sistemas batch incluyen sistemas de facturación periódica, como los sistemas de facturación telefónica y los sistemas de pago de salario.

5. *Sistemas de entretenimiento* Son sistemas para uso sobre todo personal, que tienen la intención de entretener al usuario. La mayoría de estos sistemas son juegos de uno u otro tipo. La calidad de interacción ofrecida al usuario es la característica más importante de los sistemas de entretenimiento.

6. *Sistemas para modelado y simulación* Éstos son sistemas que desarrollan científicos e ingenieros para modelar procesos o situaciones físicas, que incluyen muchos objetos separados interactuantes. Dichos sistemas a menudo son computacionalmente intensivos y para su ejecución requieren sistemas paralelos de alto desempeño.

7. *Sistemas de adquisición de datos* Son sistemas que desde su entorno recopilan datos usando un conjunto de sensores, y envían dichos datos para su procesamiento a otros sistemas. El software tiene que interactuar con los sensores y se instala regularmente en un ambiente hostil, como en el interior de un motor o en una ubicación remota.

8. *Sistemas de sistemas* Son sistemas compuestos de un cierto número de sistemas de software. Algunos de ellos son producto del software genérico, como un programa de hoja de cálculo. Otros sistemas en el ensamble pueden estar especialmente escritos para ese entorno.

Desde luego, los límites entre estos tipos de sistemas son difusos. Si se desarrolla un juego para un teléfono móvil (celular), se debe tomar en cuenta las mismas restricciones (energía, interacción de hardware) que las de los desarrolladores del software del

teléfono. Los sistemas de procesamiento por lotes se usan con frecuencia en conjunción con sistemas basados en la Web. Por ejemplo, en una compañía, las solicitudes de gastos de viaje se envían mediante una aplicación Web, aunque se procesa en una aplicación batch para pago mensual.

Para cada tipo de sistema se usan distintas técnicas de ingeniería de software, porque el software tiene características muy diferentes. Por ejemplo, un sistema de control embebido en un automóvil es crítico para la seguridad y se quema en la ROM cuando se instala en el vehículo; por consiguiente, es muy costoso cambiarlo. Tal sistema necesita verificación y validación muy exhaustivas, de tal modo que se minimicen las probabilidades de volver a llamar para revisión a automóviles, después de su venta, para corregir los problemas del software. La interacción del usuario es mínima (o quizás inexistente), por lo que no hay necesidad de usar un proceso de desarrollo que se apoye en el prototipo de interfaz de usuario.

Para un sistema basado en la Web sería adecuado un enfoque basado en el desarrollo y la entrega iterativos, con un sistema de componentes reutilizables. Sin embargo, tal enfoque podría no ser práctico para un sistema de sistemas, donde tienen que definirse por adelantado las especificaciones detalladas de las interacciones del sistema, de modo que cada sistema se desarrolle por separado.

No obstante, existen fundamentos de ingeniería de software que se aplican a todos los tipos de sistema de software:

1. Deben llevarse a cabo usando un proceso de desarrollo administrado y comprendido. La organización que diseña el software necesita planear el proceso de desarrollo, así como tener ideas claras acerca de lo que producirá y el tiempo en que estará completado. Desde luego, se usan diferentes procesos para distintos tipos de software.
2. La confiabilidad y el desempeño son importantes para todos los tipos de sistemas. El software tiene que comportarse como se espera, sin fallas, y cuando se requiera estar disponible. Debe ser seguro en su operación y, tanto como sea posible, también contra ataques externos. El sistema tiene que desempeñarse de manera eficiente y no desperdiciar recursos.
3. Es importante comprender y gestionar la especificación y los requerimientos del software (lo que el software debe hacer). Debe conocerse qué esperan de él los diferentes clientes y usuarios del sistema, y gestionar sus expectativas, para entregar un sistema útil dentro de la fecha y presupuesto.
4. Tiene que usar de manera tan efectiva como sea posible los recursos existentes. Esto significa que, donde sea adecuado, hay que reutilizar el software que se haya desarrollado, en vez de diseñar uno nuevo.

Estas nociones fundamentales sobre proceso, confiabilidad, requerimientos, gestión y reutilización, son temas importantes de este libro. Diferentes métodos los reflejan de formas diversas, pero subyacen en todo el desarrollo de software profesional.

Hay que destacar que estos fundamentos no cubren la implementación ni la programación. En este libro no se estudian técnicas específicas de programación, ya que ellas varían drásticamente de un tipo de sistema a otro. Por ejemplo, un lenguaje de guiones (scripts), como Ruby, sirve para programación de sistemas basados en la Web, aunque sería totalmente inadecuado para ingeniería de sistemas embebidos.

1.1.3 Ingeniería de software y la Web

El desarrollo de la World Wide Web tuvo un profundo efecto en todas nuestras vidas. En un inicio, la Web fue básicamente un almacén de información universal accesible que tuvo escaso efecto sobre los sistemas de software. Dichos sistemas corrían en computadoras locales y eran sólo accesibles desde el interior de una organización. Alrededor del año 2000, la Web comenzó a evolucionar, y a los navegadores se les agregaron cada vez más funcionalidades. Esto significó que los sistemas basados en la Web podían desarrollarse donde se tuviera acceso a dichos sistemas usando un navegador Web, en lugar de una interfaz de usuario de propósito específico. Esta situación condujo al desarrollo de una gran variedad de nuevos productos de sistemas que entregaban servicios innovadores, a los cuales se ingresaba desde la Web. A menudo los financiaban los anuncios publicitarios que se desplegaban en la pantalla del usuario y no requerían del pago directo de los usuarios.

Así como estos productos de sistemas, el desarrollo de navegadores Web que corrieran pequeños programas y realizaran cierto procesamiento local condujo a una evolución en los negocios y el software organizacional. En lugar de elaborar software e implementarlo en las PC de los usuarios, el software se implementaba en un servidor Web. Este avance hizo mucho más barato cambiar y actualizar el software, pues no había necesidad de instalar el software en cada PC. También redujo costos, ya que el desarrollo de interfaces de usuario es bastante caro. En consecuencia, dondequiera que fuera posible hacerlo, muchos negocios se mudaron a la interacción basada en la Web con sistemas de software de la compañía.

La siguiente etapa en el desarrollo de los sistemas basados en la Web fue la noción de los servicios Web. Estos últimos son componentes de software que entregan funcionalidad específica y útil, y a los que se accede desde la Web. Las aplicaciones se construyen al integrar dichos servicios Web que ofrecen diferentes compañías. En principio, esta vinculación suele ser dinámica, de modo que se utilice una aplicación cada vez que se ejecutan diferentes servicios Web. En el capítulo 19 se analiza este acercamiento al desarrollo del software.

En los últimos años se desarrolló la noción de "software como servicio". Se propuso que el software no correría usualmente en computadoras locales, sino en "nubes de cómputo" a las que se accede a través de Internet. Si usted utiliza un servicio como el correo basado en la Web, usa un sistema basado en nube. Una nube de computación es un enorme número de sistemas de cómputo vinculados que comparten muchos usuarios. Éstos no compran software, sino que pagan según el tiempo de software que se utiliza, o también se les otorga acceso gratuito a cambio de ver anuncios publicitarios que se despliegan en sus pantallas.

Por consiguiente, la llegada de la Web condujo a un significativo cambio en la forma en que se organiza el software empresarial. Antes de la Web, las aplicaciones empresariales eran básicamente monolíticas, los programas corrían en computadoras individuales o en grupos de computadoras. Las comunicaciones eran locales dentro de una organización. Ahora el software está ampliamente distribuido, en ocasiones a lo largo del mundo. Las aplicaciones empresariales no se programan desde cero, sino que requieren la reutilización extensiva de componentes y programas.

En efecto, este cambio radical en la organización del software tuvo que conducir a modificaciones en las formas en que los sistemas basados en la Web se someten a ingeniería. Por ejemplo:

1. La reutilización de software se ha convertido en el enfoque dominante para construir sistemas basados en la Web. Cuando se construyen tales sistemas, uno piensa en cómo ensamblarlos a partir de componentes y sistemas de software preexistentes.

2. Ahora se reconoce en general que no es práctico especificar por adelantado todos los requerimientos para tales sistemas. Los sistemas basados en la Web deben desarrollarse y entregarse de manera progresiva.

3. Las interfaces de usuario están restringidas por las capacidades de los navegadores Web. Aunque tecnologías como AJAX (Holdener, 2008) significan que es posible crear valiosas interfaces dentro de un navegador Web, dichas tecnologías aún son difíciles de emplear. Se usan más comúnmente los formatos Web con escritura de guiones local. Las interfaces de aplicación en sistemas basados en la Web con frecuencia son más deficientes que las interfaces de usuario específicamente diseñadas en productos de sistema PC.

Las ideas fundamentales de la ingeniería de software, discutidas en la sección anterior, se aplican en el software basado en la Web de la misma forma que en otros tipos de sistemas de software. En el siglo XX, la experiencia obtenida con el desarrollo de grandes sistemas todavía es relevante para el software basado en la Web.

1.2 Ética en la ingeniería de software

Como otras disciplinas de ingeniería, la ingeniería de software se realiza dentro de un marco social y legal que limita la libertad de la gente que trabaja en dicha área. Como ingeniero de software, usted debe aceptar que su labor implica responsabilidades mayores que la simple aplicación de habilidades técnicas. También debe comportarse de forma ética y moralmente responsable para ser respetado como un ingeniero profesional.

No sobra decir que debe mantener estándares normales de honestidad e integridad. No debe usar sus habilidades y experiencia para comportarse de forma deshonesta o de un modo que desacredite la profesión de ingeniería de software. Sin embargo, existen áreas donde los estándares de comportamiento aceptable no están acotados por la legislación, sino por la noción más difusa de responsabilidad profesional. Algunas de ellas son:

1. *Confidencialidad* Por lo general, debe respetar la confidencialidad de sus empleadores o clientes sin importar si se firmó o no un acuerdo formal sobre la misma.

2. *Competencia* No debe desvirtuar su nivel de competencia. Es decir, no hay que aceptar de manera intencional trabajo que esté fuera de su competencia.

3. *Derechos de propiedad intelectual* Tiene que conocer las leyes locales que rigen el uso de la propiedad intelectual, como las patentes y el *copyright*. Debe ser cuidadoso para garantizar que se protege la propiedad intelectual de empleadores y clientes.

4. *Mal uso de computadoras* No debe emplear sus habilidades técnicas para usar incorrectamente las computadoras de otros individuos. El mal uso de computadoras varía desde lo relativamente trivial (esto es, distraerse con los juegos de la PC del compañero) hasta lo extremadamente serio (diseminación de virus u otro malware).

Código de ética y práctica profesional de la ingeniería de software

ACM/IEEE-CS Fuerza de trabajo conjunta acerca de ética y prácticas profesionales de la ingeniería de software

PREÁMBULO

La versión corta del código resume las aspiraciones a un alto nivel de abstracción; las cláusulas que se incluyen en la versión completa dan ejemplos y detalles de cómo dichas aspiraciones cambian la forma en que actuamos como profesionales de la ingeniería de software. Sin las aspiraciones, los detalles pueden volverse legalistas y tediosos; mientras que sin los detalles, las aspiraciones suelen volverse muy resonantes pero vacías; en conjunto, aspiraciones y detalles forman un código cohesivo.

Los ingenieros de software deben comprometerse a hacer del análisis, la especificación, el diseño, el desarrollo, la prueba y el mantenimiento del software, una profesión benéfica y respetada. De acuerdo con su compromiso con la salud, la seguridad y el bienestar del público, los ingenieros de software tienen que adherirse a los ocho principios siguientes:

1. PÚBLICO: Los ingenieros de software deben actuar consecuentemente con el interés del público.
2. CLIENTE Y EMPLEADOR: Los ingenieros de software tienen que comportarse de tal forma que fomente el mejor interés para su cliente y empleador, en coherencia con el interés público.
3. PRODUCTO: Los ingenieros de software deben garantizar que sus productos y modificaciones relacionadas satisfagan los estándares profesionales más altos posibles.
4. JUICIO: Los ingenieros de software tienen que mantener integridad e independencia en su juicio profesional.
5. GESTIÓN: Los administradores y líderes en la ingeniería de software deben suscribir y promover un enfoque ético a la gestión del desarrollo y el mantenimiento del software.
6. PROFESIÓN: Los ingenieros de software tienen que fomentar la integridad y la reputación de la profesión consecuente con el interés público.
7. COLEGAS: Los ingenieros de software deben ser justos con sus colegas y apoyarlos.
8. UNO MISMO: Los ingenieros de software tienen que intervenir en el aprendizaje para toda la vida, en cuanto a la práctica de su profesión, y promover un enfoque ético.

Figura 1.3 El código de ética ACM/IEEE (© IEEE/ACM, 1999)

Las sociedades e instituciones profesionales tienen un importante papel que desempeñar en el establecimiento de estándares éticos. Organizaciones como la ACM, el instituto de ingenieros eléctricos y electrónicos (IEEE) y la British Computer Society publican un código de conducta profesional o código de ética. Los integrantes de tales organizaciones se comprometen a seguir dicho código cuando firman al afiliarse. Estos códigos de conducta se preocupan en general por el comportamiento ético fundamental.

Las asociaciones profesionales, sobre todo la ACM y el IEEE, han cooperado para elaborar conjuntamente un código de ética y práctica profesionales. Este código existe tanto de manera simplificada (figura 1.3) como pormenorizada (Gotterbarn *et al*., 1999) que agrega detalle y sustancia a la versión más corta. Los fundamentos detrás de este código se resumen en los primeros dos párrafos de la forma pormenorizada:

> *Las computadoras tienen una función central y creciente en el comercio, la industria, el gobierno, la medicina, la educación, el entretenimiento y la sociedad en general. Los ingenieros de software son quienes contribuyen, mediante la participación directa o con la enseñanza, al análisis, la especificación, el diseño, el desarrollo, la certificación, el mantenimiento y la prueba de los sistemas de software.*

Debido a su función en el desarrollo de los sistemas de software, los ingenieros de software tienen oportunidades significativas para hacer lo correcto o causar daño, para permitir que otros hagan lo correcto o causen daño, o para influir en otros para hacer lo correcto o causar daño. Para garantizar, tanto como sea posible, que sus esfuerzos serán usados correctamente, los ingenieros de software deben comprometerse a hacer de la ingeniería de software una profesión benéfica y respetada. En concordancia con dicho compromiso, los ingenieros de software tienen que adherirse al siguiente Código de Ética y Práctica Profesional.

El código contiene ocho principios relacionados con el comportamiento y las decisiones tomadas por ingenieros de software profesionales, incluidos practicantes, educadores, administradores, supervisores y políticos, así como por aprendices y estudiantes de la profesión. Los principios identifican las relaciones éticamente responsables en las que participan individuos, grupos y organizaciones, así como las obligaciones principales dentro de estas relaciones. Las cláusulas de cada principio son ilustraciones de algunas de las obligaciones incluidas en dichas relaciones. Tales obligaciones se fundamentan en el sentido humano del ingeniero de software, en el cuidado especial que se debe a las personas afectadas por el trabajo de los ingenieros de software, y los elementos únicos de la práctica de la ingeniería de software. El código las formula como obligaciones de quienquiera que afirme o aspire a ser ingeniero de software.

En cualquier situación donde distintos individuos tengan diferentes visiones y objetivos, es probable que usted enfrente dilemas éticos. Por ejemplo, si está en desacuerdo, en principio, con las políticas de los ejecutivos de más alto nivel en la compañía, ¿cómo reaccionaría? Claramente, esto depende de cada individuo y de la naturaleza de la discrepancia. ¿Es mejor argumentar un caso para su posición desde el interior de la organización o renunciar en principio? Si siente que existen problemas con un proyecto de software, ¿cuándo los reporta a la administración? Si los discute mientras apenas son un indicio, puede estar exagerando su reacción ante una situación; y si los deja para más tarde, quizá sea ya imposible resolver las dificultades.

Estos dilemas éticos los enfrentamos todos en la vida profesional y, por fortuna, en la mayoría de los casos son relativamente menores o pueden resolverse sin demasiada dificultad. En caso de que no puedan solucionarse, el ingeniero afronta, tal vez, otro problema. La acción basada en los principios quizá sea renunciar al empleo, aunque esta decisión bien podría afectar a otros, como a su pareja o a sus hijos.

Una situación muy difícil para los ingenieros profesionales surge cuando su empleador actúa sin ética. Es decir, una compañía es responsable del desarrollo de un sistema crítico de seguridad y, debido a presión del tiempo, falsifica los registros de validación de seguridad. ¿Es responsabilidad del ingeniero mantener la confidencialidad o alertar al cliente o manifestar, de alguna forma, que el sistema entregado quizá sea inseguro?

El problema aquí es que no hay absolutos cuando se trata de seguridad. Aunque el sistema pueda no estar validado de acuerdo con criterios predefinidos, dichos criterios quizá sean demasiado estrictos. En realidad el sistema operará con seguridad a lo largo de su vida. También está el caso de que, aun cuando se valide de manera adecuada, el sistema falle y cause un accidente. La detección oportuna de los problemas puede resultar lesiva para el empleador y otros trabajadores; el fracaso por revelar los problemas podría ser dañino para otros.

El lector debe formar su propio criterio en estos asuntos. Aquí, la posición ética adecuada depende por completo de las percepciones de los individuos que están implicados. En este caso, el potencial de daño, el alcance del mismo y las personas afectadas deben influir en la decisión. Si el escenario es muy peligroso, estaría justificado anunciarlo a través de la prensa nacional (por ejemplo). Sin embargo, siempre hay que tratar de resolver la situación sin dejar de respetar los derechos de su empleador.

Otro conflicto ético es la participación en el desarrollo de sistemas militares y nucleares. Al respecto, algunas personas se sienten muy afectadas por estos temas y evitan participar en el desarrollo de algún sistema asociado con los sistemas militares. Otras más trabajarán en los sistemas militares, pero no en los de armamento. Incluso otras sentirán que la seguridad nacional es un principio fundamental y no tienen objeciones éticas para trabajar en sistemas de armamento.

En tal situación es importante que tanto empleadores como empleados dejen en claro con antelación sus percepciones o puntos de vista. Cuando una organización participa en trabajo militar o nuclear, debe contar con la capacidad de especificar que los empleados tienen la voluntad de aceptar cualquier trabajo asignado. De igual forma, si un empleado toma la responsabilidad y deja en claro que no quiere trabajar en tales sistemas, los empleadores no tendrán que presionarlo para que éste lo haga más tarde.

El área general de la ética y la responsabilidad profesional se vuelven más importantes conforme los sistemas intensivos en software prevalecen en cada vez más cuestiones del trabajo y la vida cotidiana. Puede considerarse desde un punto de vista filosófico, donde se tomen en cuenta los principios básicos de la ética y se analice la ética de la ingeniería de software en relación con dichos principios básicos. Éste es el enfoque que toma Laudon (1995) y, en menor medida, Huff y Martin (1995). El texto de Johnson sobre ética computacional (2001) también trata el tema desde una perspectiva filosófica.

Sin embargo, este enfoque filosófico resulta muy abstracto y difícil de relacionar con la experiencia cotidiana. Es preferible el enfoque más concreto plasmado en los códigos de conducta y práctica. Se considera que la ética se analiza mejor en un contexto de ingeniería de software y no como un tema por derecho propio. Por lo tanto, en este libro no se presentan, donde es adecuado, discusiones éticas abstractas, sino que se incluyen ejemplos en los ejercicios que son el punto de partida para una discusión grupal sobre conflictos éticos.

1.3 Estudios de caso

Para ilustrar los conceptos de la ingeniería de software, a lo largo del libro se utilizan ejemplos de tres tipos de sistemas diferentes. La razón de no usar un solo estudio de caso obedece a que uno de los mensajes clave de este libro es que la práctica de la ingeniería de software depende del tipo de sistemas a producir. Por consiguiente, se elegirá un ejemplo adecuado cuando se estudien conceptos como seguridad y confiabilidad, modelado de sistema, reutilización, etcétera.

Los tres tipos de sistemas que se usan como estudios de caso son:

1. *Un sistema embebido* Se trata de un sistema donde el software controla un dispositivo de hardware y está *embebido* en dicho dispositivo. Los conflictos en los sistemas *embebidos* incluyen por lo general tamaño físico, capacidad de reacción,

administración de la energía, etcétera. El ejemplo de un sistema *embebido* utilizado es un sistema de software para controlar un dispositivo médico.

2. *Un sistema de información* Es un sistema cuyo principal propósito es gestionar y dar acceso a una base de datos de información. Los conflictos en los sistemas de información incluyen seguridad, usabilidad, privacidad y mantenimiento de la integridad de los datos. Un sistema de registros médicos se utiliza como ejemplo de un sistema de información.

3. *Un sistema de adquisición de datos basado en sensores* Se trata de un sistema cuyo principal objetivo es recolectar datos de un conjunto de sensores y procesar esos datos de alguna forma. Los requerimientos clave de tales sistemas son fiabilidad, incluso en condiciones de ambientes hostiles, y capacidad de mantenimiento. Una estación meteorológica a campo abierto es el ejemplo que se usa como sistema de adquisición de datos.

En este capítulo se introduce cada uno de dichos sistemas, y sobre todos ellos hay más información disponible en la Web.

1.3.1 Sistema de control para una bomba de insulina

Una bomba de insulina es un sistema médico que simula la función del páncreas (un órgano interno). El software que controla este sistema es un sistema embebido, que recopila información de un sensor y controla una bomba que entrega al usuario una dosis regulada de insulina.

Las personas que sufren de diabetes usan el sistema. La diabetes es relativamente una condición común, donde el páncreas humano es incapaz de producir suficientes cantidades de una hormona llamada insulina. La insulina metaboliza la glucosa (azúcar) en la sangre. El tratamiento convencional de la diabetes incluye inyecciones regulares de insulina genéticamente manipulada. Los diabéticos calculan sus niveles de azúcar en la sangre usando un medidor externo y, luego, ajustan la dosis de insulina que deben inyectarse.

El problema con este tratamiento es que el nivel de insulina requerido no depende sólo del nivel de glucosa en la sangre, sino también del tiempo desde la última inyección de insulina. Esto podría conducir a niveles muy bajos de glucosa sanguínea (si hay mucha insulina) o niveles muy altos de azúcar sanguínea (si hay muy poca insulina). La baja en glucosa sanguínea es, a corto plazo, una condición más seria que puede resultar en mal funcionamiento temporal del cerebro y, finalmente, en inconsciencia y muerte. Y por otro lado, a largo plazo los continuos niveles elevados de glucosa en la sangre ocasionan daño ocular, renal y problemas cardiacos.

Los avances recientes en el desarrollo de sensores miniaturizados significan que ahora es posible desarrollar sistemas automatizados de suministro de insulina. Dichos sistemas monitorizan los niveles de azúcar en la sangre y, cuando se requiere, administran una dosis adecuada de insulina. Los sistemas de entrega de insulina como éste ya existen para el tratamiento de pacientes hospitalarios. En el futuro, muchos diabéticos tendrán tales sistemas permanentemente unidos a sus cuerpos.

Un sistema de suministro de insulina controlado por software puede funcionar al usar un microsensor embebido en el paciente, con la finalidad de medir ciertos parámetros sanguíneos que sean proporcionales al nivel de azúcar. Luego, esto se envía al controlador de la bomba, el cual calcula el nivel de azúcar y la cantidad de insulina que se nece-

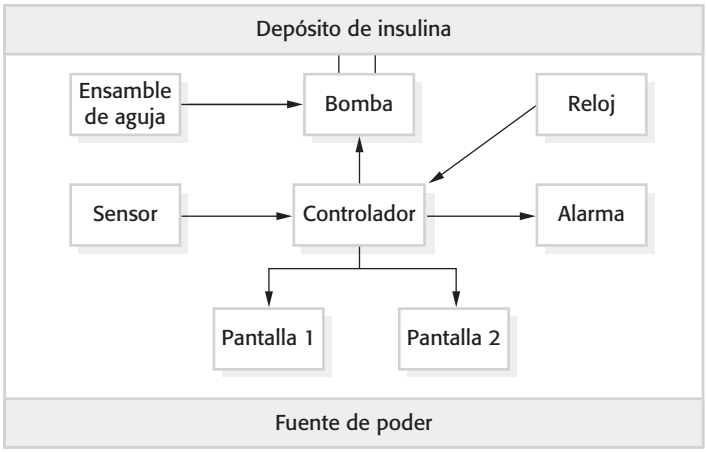


Figura 1.4 Hardware de bomba de insulina

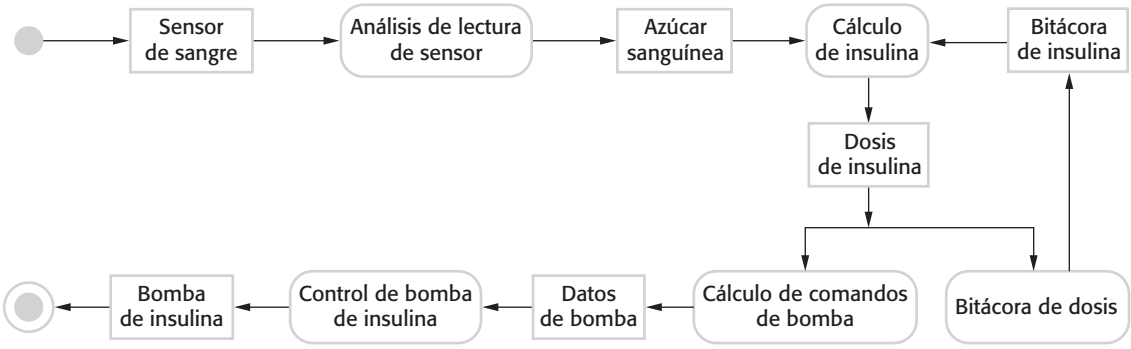


Figura 1.5 Modelo de actividad de la bomba de insulina

sita. Entonces envía señales a una bomba miniaturizada para administrar la insulina vía una aguja permanentemente unida.

La figura 1.4 muestra los componentes de hardware y la organización de la bomba de insulina. Para entender los ejemplos, todo lo que necesita saber es que el sensor de sangre mide la conductividad eléctrica de la sangre bajo diferentes condiciones y que dichos valores podrían relacionarse con el nivel de azúcar en la sangre. La bomba de insulina entrega una unidad de insulina en respuesta a un solo pulso de un controlador. Por lo tanto, para entregar 10 unidades de insulina, el controlador envía 10 pulsos a la bomba. La figura 1.5 es un modelo de actividad UML que ilustra cómo el software transforma una entrada de nivel de azúcar en la sangre, con una secuencia de comandos que impulsan la bomba de insulina.

Claramente, éste es un sistema crítico de seguridad. Si la bomba no opera o no lo hace de manera correcta, entonces la salud del usuario estaría en grave riesgo o éste caería en estado de coma debido a que sus niveles de azúcar en la sangre son muy altos o muy bajos. En consecuencia, hay dos requerimientos esenciales de alto nivel que debe satisfacer este sistema:

1. El sistema tiene que estar disponible para entregar insulina cuando se requiera.
2. El sistema requiere funcionar de manera confiable y entregar la cantidad correcta de insulina, para contrarrestar el nivel actual de azúcar en la sangre.

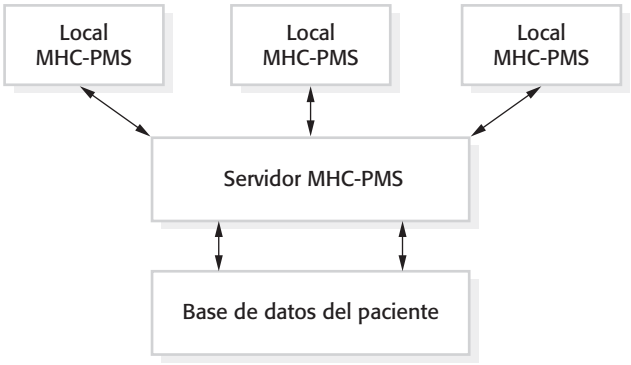


Figura 1.6 Organización del MHC-PMS

Por consiguiente, el sistema debe diseñarse e implementarse para garantizar que siempre satisfaga dichos requerimientos. En capítulos posteriores se estudian requerimientos más detallados y se discute acerca de cómo probar que el sistema sea seguro.

1.3.2 Sistema de información de pacientes para atención a la salud mental

Un sistema de información de pacientes para apoyar la atención a la salud mental es un sistema de información médica, que administra la información de pacientes que sufren problemas de salud mental y los tratamientos que reciben. La mayoría de los pacientes con problemas de salud mental no requieren tratamiento hospitalario dedicado, pero sí asistir regularmente a clínicas especializadas donde se reúnen con un médico que tiene conocimiento detallado de sus problemas. Para facilitar la asistencia de los pacientes, dichas clínicas no sólo funcionan en los hospitales sino también en consultorios médicos locales o centros comunitarios.

El MHC-PMS (sistema de administración de pacientes-atención a la salud mental) es un sistema de información destinado para usarse en clínicas. Utiliza una base de datos centralizada con información de los pacientes, aunque también se diseñó para operarse desde una PC, de modo que se puede acceder a ella y usarse desde sitios sin conectividad de red segura. Cuando los sistemas locales tienen acceso seguro a red, emplean la información de los pacientes en la base de datos, pero además son capaces de descargar y usar copias locales de registros de pacientes cuando los sistemas están desconectados. El sistema no es un sistema de registros médicos completo, por lo que no conserva información acerca de otras condiciones médicas. Sin embargo, interactúa e intercambia datos con otros sistemas de información clínica. La figura 1.6 ilustra la organización del MHC-PMS.

El MHC-PMS tiene dos metas globales:

1. Generar información de gestión que permita a los administradores de servicios de salud valorar el desempeño contra objetivos locales y de gobierno.
2. Proporcionar al personal médico información oportuna para apoyar el tratamiento de los pacientes.

La naturaleza de los problemas de salud mental es tal que los pacientes se hallan con frecuencia desorganizados y suelen faltar a sus citas, deliberada o accidentalmente, perder recetas y medicamentos, olvidar instrucciones y realizar demandas irracionales al personal médico. Pueden llegar a las clínicas de manera inesperada. En muy pocos casos, son un riesgo para sí mismos o para otros individuos. Regularmente pueden cambiar de dirección o no tener casa a corto o largo plazo. Cuando los pacientes son peligrosos, quizá deban "internarse": confinarse en un hospital seguro para tratamiento y observación.

Los usuarios del sistema incluyen personal clínico como médicos, enfermeros y visitadores de salud (enfermeros que visitan a las personas a domicilio para verificar su tratamiento). Los usuarios no médicos incluyen recepcionistas que hacen citas, personal de archivo médico que organiza el sistema de registros, y personal administrativo que redacta informes.

El sistema sirve para registrar información de pacientes (nombre, dirección, edad, pariente más cercano, etcétera), consultas (fecha, médico, impresiones personales del paciente, etcétera), condiciones y tratamientos. Los informes se elaboran a intervalos regulares para el personal médico y los administradores de la autoridad sanitaria. Por lo general, los reportes para el personal médico se enfocan en la información individual de pacientes, mientras que los reportes de la administración son anónimos y se interesan por las condiciones, costos de tratamiento, etcétera.

Las características clave del sistema son:

1. *Administración de atención individual* Los médicos de atención primaria crean registros para pacientes, editan la información en el sistema, ven el historial del paciente, etcétera. El sistema soporta resúmenes de datos para que los médicos que no se reunieron con anterioridad con el paciente se enteren rápidamente de los problemas y tratamientos clave que se prescribieron.
2. *Monitorización del paciente* El sistema monitoriza regularmente los registros de los pacientes que están involucrados en tratamiento y emite advertencias cuando se detectan posibles dificultades. En consecuencia, si un paciente no ha visto a un médico durante cierto tiempo, puede emitirse una advertencia. Uno de los elementos más importantes del sistema de monitorización es seguir la pista de los pacientes que fueron internados y garantizar que las verificaciones requeridas legalmente se lleven a cabo en el tiempo correcto.
3. *Informes administrativos* El sistema genera mensualmente informes administrativos que muestran el número de pacientes tratados en cada clínica, la cantidad de pacientes que ingresaron y salieron del sistema de salud, el total de pacientes internados, los medicamentos prescritos y sus costos, etcétera.

Dos leyes diferentes afectan al sistema. Se trata de leyes de protección de datos que rigen la confidencialidad de la información personal, y las leyes de salud mental, que establecen la detención obligatoria de los pacientes considerados como un peligro para sí mismos o para otros. La salud mental es única en este aspecto, pues es la única especialidad médica que puede recomendar la detención de pacientes contra la voluntad de éstos, lo cual está sujeto a protecciones legislativas muy estrictas. Una de las metas del MHC-PMS es asegurar que el personal siempre actúe en concordancia con la ley y que sus decisiones, si es necesario, se registren para revisión judicial.

Como en todos los sistemas médicos, la privacidad es un requerimiento de sistema crítico. Es básico que la información de los pacientes sea confidencial y nunca se revele a nadie

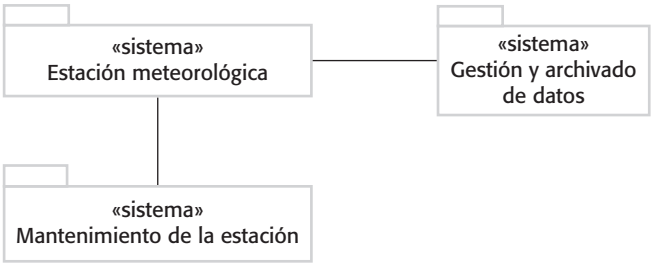


Figura 1.7 El entorno de la estación meteorológica

más, aparte del personal médico autorizado y los mismos pacientes. El MHC-PMS también es un sistema crítico de seguridad. Algunas patologías mentales hacen que los pacientes se vuelvan suicidas o un peligro para otros individuos. Siempre que sea posible, el sistema debe advertir al personal médico acerca de pacientes potencialmente suicidas o peligrosos.

El diseño global del sistema debe considerar requerimientos de privacidad y seguridad. El sistema tiene que estar disponible cuando se necesite, de otro modo la seguridad estaría comprometida y sería imposible prescribir a los pacientes el medicamento correcto. Aquí existe un conflicto potencial: la privacidad es más fácil de mantener cuando existe sólo una copia de los datos del sistema. Sin embargo, para garantizar la disponibilidad en el caso de fallas del servidor o desconexión de una red, hay que conservar varias copias de los datos. En capítulos posteriores se analizan las preferencias temporales entre tales requerimientos.

1.3.3 Estación meteorológica a campo abierto

Para ayudar a monitorizar el cambio climático y mejorar la exactitud de las predicciones meteorológicas en áreas remotas, el gobierno de un país con grandes áreas de campo abierto decidió instalar varios cientos de estaciones meteorológicas en dichas áreas. Las estaciones meteorológicas recopilan datos de un conjunto de instrumentos que miden temperatura y presión, luz solar, lluvia, y rapidez y dirección del viento.

Las estaciones meteorológicas a campo abierto son parte de un sistema más grande (figura 1.7), que es un sistema de información meteorológica que recolecta datos de estaciones meteorológicas y los pone a disposición de otros sistemas para su procesamiento. Los sistemas en la figura 1.7 son:

1. *El sistema de estación meteorológica* Es responsable de recolectar datos meteorológicos, realizar cierto procesamiento de datos inicial y transmitirlo al sistema de gestión de datos.
2. *El sistema de gestión y archivado de datos* Recolecta los datos de todas las estaciones meteorológicas a campo abierto, realiza procesamiento y análisis de datos, y los archiva de forma que los puedan recuperar otros sistemas, como los sistemas de predicción meteorológica.
3. *El sistema de mantenimiento de estación* Se comunica por satélite con todas las estaciones meteorológicas a campo abierto, para monitorizar el estado de dichos sistemas y dar reportes sobre problemas. Puede actualizar el software embebido en dichos sistemas. En caso de problemas del sistema, también sirve para controlar de manera remota un sistema meteorológico a campo abierto.

En la figura 1.7 se usó el símbolo de paquete UML para indicar que cada sistema es una colección de componentes, y se identificaron los sistemas separados usando el estereotipo UML «sistema». Las asociaciones entre los paquetes indican que ahí existe un intercambio de información pero, en esta etapa, no hay necesidad de definirlos con más detalle.

Cada estación meteorológica incluye algunos instrumentos que miden parámetros climatológicos como rapidez y dirección del viento, temperaturas del terreno y aire, presión barométrica y lluvia durante un periodo de 24 horas. Cada uno de dichos instrumentos está controlado por un sistema de software que toma periódicamente lecturas de parámetros y gestiona los datos recolectados desde los instrumentos.

El sistema de estación meteorológica opera mediante la recolección de observaciones meteorológicas a intervalos frecuentes; por ejemplo, las temperaturas se miden cada minuto. Sin embargo, puesto que el ancho de banda del satélite es relativamente estrecho, la estación meteorológica realiza cierto procesamiento local y concentración de los datos. Luego, transmite los datos concentrados cuando los solicita el sistema de adquisición de datos. Pero si, por cualquier razón, es imposible realizar una conexión, entonces la estación meteorológica mantiene los datos localmente hasta que se reanude la comunicación.

Cada estación meteorológica es alimentada por baterías y debe estar completamente autocontenida: no hay fuentes de energía externas o cables de red disponibles. Todas las comunicaciones son a través de un vínculo satelital de rapidez relativamente baja, y la estación meteorológica debe incluir algún mecanismo (solar o eólico) para cargar sus baterías. Puesto que se despliegan en áreas abiertas, están expuestas a severas condiciones ambientales y los animales llegan a dañarlas. Por lo tanto, el software de la estación no sólo se encarga de la adquisición de datos. También debe:

1. Monitorizar los instrumentos, la energía y el hardware de comunicación, y reportar los fallas al sistema de administración.
2. Administrar la energía del sistema, garantizar que las baterías estén cargadas siempre que las condiciones ambientales lo permitan; así como desconectar los generadores ante condiciones meteorológicas potencialmente adversas, como viento fuerte.
3. Permitir la reconfiguración dinámica donde partes del software se sustituyan con nuevas versiones, y los instrumentos de respaldo se enciendan en el sistema en caso de falla de éste.

Puesto que las estaciones meteorológicas deben estar autocontenidas y sin vigilancia, esto significa que el software instalado es complejo, aun cuando la funcionalidad de adquisición de datos sea bastante simple.

PUNTOS CLAVE

- La ingeniería de software es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software.
- El software no es sólo un programa o programas, sino que también incluye documentación. Los atributos esenciales de los productos de software son mantenimiento, confiabilidad, seguridad, eficiencia y aceptabilidad.
- El proceso de software incluye todas las actividades que intervienen en el desarrollo de software. Las actividades de alto nivel de especificación, desarrollo, validación y evolución son parte de todos los procesos de software.
- Las nociones fundamentales de la ingeniería de software son universalmente aplicables a todos los tipos de desarrollo de sistema. Dichos fundamentos incluyen procesos, confiabilidad, seguridad, requerimientos y reutilización de software.
- Existen muchos tipos diferentes de sistemas y cada uno requiere para su desarrollo de herramientas y técnicas adecuadas de ingeniería de software. Existen pocas, si es que hay alguna, técnicas específicas de diseño e implementación que son aplicables a todos los tipos de sistemas.
- Las ideas fundamentales de la ingeniería de software son aplicables a todos los tipos de sistemas de software. Dichos fundamentos incluyen procesos de administración de software, confiabilidad y seguridad del software, ingeniería de requerimientos y reutilización de software.
- Los ingenieros de software tienen responsabilidades con la profesión de ingeniería y la sociedad. No deben preocuparse únicamente por temas técnicos.
- Las sociedades profesionales publican códigos de conducta que establecen los estándares de comportamiento esperados de sus miembros.

LECTURAS SUGERIDAS

"No silver bullet: Essence and accidents of software engineering". A pesar de su fecha de publicación, este artículo es una buena introducción general a los problemas de la ingeniería de software. El mensaje esencial del artículo no ha cambiado. (F. P. Brooks, *IEEE Computer,* **20** (4), abril 1987.) http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532.

"Software engineering code of ethics is approved". Un artículo que analiza a fondo los antecedentes para el desarrollo del Código de ética ACM/IEEE y que incluye las formas corta y larga del código. (*Comm. ACM*, D. Gotterbarn, K. Miller, y S. Rogerson, octubre 1999.) http://portal.acm.org/citation.cfm?doid=317665.317682.

Professional Issues in Software Engineering. Éste es un excelente libro que examina conflictos legales y profesionales, así como éticos. El autor prefiere su enfoque práctico a textos más teóricos acerca de la ética. (F. Bott, A. Coleman, J. Eaton y D. Rowland, 3ª. edición, 2000, Taylor and Francis.)

IEEE Software, March/April 2002. Éste es un número especial de la revista dedicado al desarrollo de software basado en la Web. Esta área ha cambiado muy rápidamente, de modo que algunos artículos son un poco arcaicos, pero la mayoría todavía son relevantes. (*IEEE Software,* **19** (2), 2002.) http://www2.computer.org/portal/web/software.

"A View of 20th and 21st Century Software Engineering". Un vistazo atrás y adelante a la ingeniería de software de uno de los primeros y más distinguidos ingenieros de software. Barry Boehm identifica principios atemporales de la ingeniería de software, pero también sugiere que algunas prácticas de uso común son obsoletas. (B. Boehm, *Proc. 28th Software Engineering Conf.,* Shanghai. 2006.) http://doi.ieeecomputersociety.org/10.1145/1134285.1134288.

"Software Engineering Ethics". Número especial de IEEE Computer, con algunos artículos acerca del tema. (*IEEE Computer,* **42** (6), junio 2009.)

EJERCICIOS

1.1. Explique por qué el software profesional no sólo son programas que se desarrollan para un cliente.

1.2. ¿Cuál es la principal diferencia entre desarrollo de productos de software genéricos y desarrollo de software personalizado? ¿Qué significa esto en la práctica para los usuarios de productos de software genérico?

1.3. ¿Cuáles son los cuatro atributos importantes que debe tener todo software profesional? Sugiera otros cuatro atributos que en ocasiones sean significativos.

1.4. Además de los retos de la heterogeneidad, cambio empresarial y social, y confianza y seguridad, identifique otros problemas y retos que sea probable que enfrente la ingeniería de software en el siglo XXI. (Sugerencia: piense en el ambiente).

1.5. Con base en su conocimiento de algunos tipos de aplicación estudiados en la sección 1.1.2, explique, con ejemplos, por qué diferentes tipos de aplicación requieren técnicas especializadas de ingeniería de software, para apoyar su diseño y desarrollo.

1.6. Explique por qué existen ideas fundamentales de la ingeniería de software que se aplican a todos los tipos de sistemas de software.

1.7. Explique cómo el uso universal de la Web cambió los sistemas de software.

1.8. Analice el hecho de si los ingenieros profesionales deben ser certificados en la misma forma que los médicos o abogados.

1.9. Para cada una de las cláusulas del Código de ética ACM/IEEE que se muestra en la figura 1.3, sugiera un ejemplo adecuado que ilustre dicha cláusula.

1.10. Para ayudar a contrarrestar el terrorismo, muchos países planean o desarrollaron sistemas de cómputo que siguen la pista a gran cantidad de sus ciudadanos y sus acciones. Claramente esto tiene implicaciones en cuanto a la privacidad. Discuta la ética de trabajar en el desarrollo de este tipo de sistema.

REFERENCIAS

Gotterbarn, D., Miller, K. y Rogerson, S. (1999). Software Engineering Code of Ethics is Approved. Comm. ACM, **42** (10), 102–7.

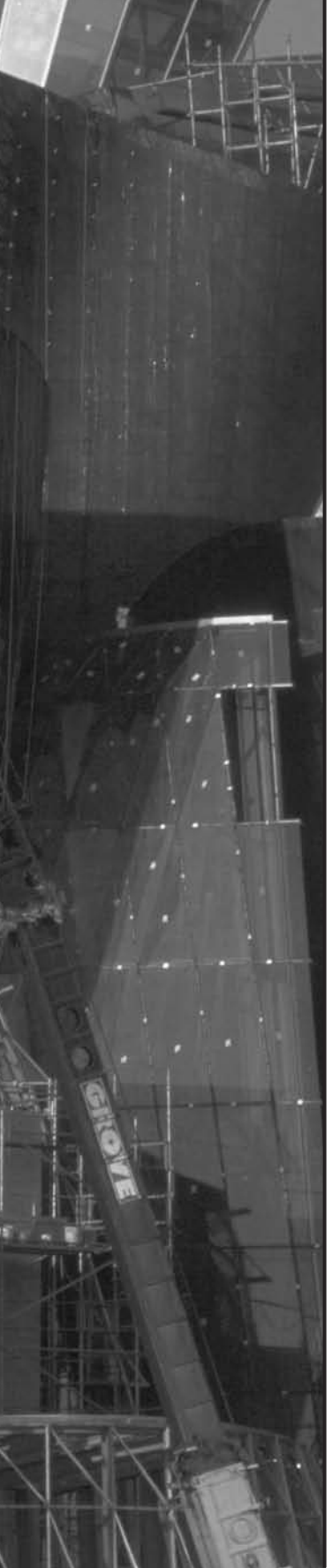
Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Ca.: O'Reilly and Associates.

Huff, C. y Martin, C. D. (1995). Computing Consequences: A Framework for Teaching Ethical Computing. Comm. ACM, **38** (12), 75–84.

Johnson, D. G. (2001). *Computer Ethics*. Englewood Cliffs, NJ: Prentice Hall.

Laudon, K. (1995). Ethical Concepts and Information Technology. Comm. ACM, **38** (12), 33–9.

Naur, P. y Randell, B. (1969). Ingeniería de software: Reporte de una conferencia patrocinada por el comité científico de la OTAN, Garmisch, Alemania, 7 a 11 de octubre de 1968.



2

Procesos de software

Objetivos

El objetivo de este capítulo es introducirlo hacia la idea de un proceso de software: un conjunto coherente de actividades para la producción de software. Al estudiar este capítulo:

- comprenderá los conceptos y modelos sobre procesos de software;
- se introducirá en los tres modelos de proceso de software genérico y sabrá cuándo usarlos;
- entenderá las principales actividades del proceso de ingeniería de requerimientos de software, así como del desarrollo, las pruebas y la evolución del software;
- comprenderá por qué deben organizarse los procesos para enfrentar los cambios en los requerimientos y el diseño de software;
- entenderá cómo el Proceso Unificado Racional (Rational Unified Process, RUP) integra buenas prácticas de ingeniería de software para crear procesos de software adaptables.

Contenido

2.1 Modelos de proceso de software
2.2 Actividades del proceso
2.3 Cómo enfrentar el cambio
2.4 El Proceso Unificado Racional

Un proceso de software es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Estas actividades pueden incluir el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, las aplicaciones de negocios no se desarrollan precisamente de esta forma. El nuevo software empresarial con frecuencia ahora se desarrolla extendiendo y modificando los sistemas existentes, o configurando e integrando el software comercial o componentes del sistema.

Existen muchos diferentes procesos de software, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software:

1. *Especificación del software* Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
2. *Diseño e implementación del software* Debe desarrollarse el software para cumplir con las especificaciones.
3. *Validación del software* Hay que validar el software para asegurarse de que cumple lo que el cliente quiere.
4. *Evolución del software* El software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

En cierta forma, tales actividades forman parte de todos los procesos de software. Por supuesto, en la práctica éstas son actividades complejas en sí mismas e incluyen subactividades tales como la validación de requerimientos, el diseño arquitectónico, la prueba de unidad, etcétera. También existen actividades de soporte al proceso, como la documentación y el manejo de la configuración del software.

Cuando los procesos se discuten y describen, por lo general se habla de actividades como especificar un modelo de datos, diseñar una interfaz de usuario, etcétera, así como del orden de dichas actividades. Sin embargo, al igual que las actividades, también las descripciones de los procesos deben incluir:

1. Productos, que son los resultados de una actividad del proceso. Por ejemplo, el resultado de la actividad del diseño arquitectónico es un modelo de la arquitectura de software.
2. Roles, que reflejan las responsabilidades de la gente que interviene en el proceso. Ejemplos de roles: gerente de proyecto, gerente de configuración, programador, etcétera.
3. Precondiciones y postcondiciones, que son declaraciones válidas antes y después de que se realice una actividad del proceso o se cree un producto. Por ejemplo, antes de comenzar el diseño arquitectónico, una precondición es que el cliente haya aprobado todos los requerimientos; después de terminar esta actividad, una postcondición podría ser que se revisen aquellos modelos UML que describen la arquitectura.

Los procesos de software son complejos y, como todos los procesos intelectuales y creativos, se apoyan en personas con capacidad de juzgar y tomar decisiones. No hay un proceso ideal; además, la mayoría de las organizaciones han diseñado sus propios procesos de desarrollo de software. Los procesos han evolucionado para beneficiarse de las capacidades de la gente en una organización y de las características específicas de los

sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere de un proceso de desarrollo muy estructurado. Para los sistemas empresariales, con requerimientos rápidamente cambiantes, es probable que sea más efectivo un proceso menos formal y flexible.

En ocasiones, los procesos de software se clasifican como dirigidos por un plan (*plan-driven*) o como procesos ágiles. Los procesos dirigidos por un plan son aquellos donde todas las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan. En los procesos ágiles, que se estudiarán en el capítulo 3, la planeación es incremental y es más fácil modificar el proceso para reflejar los requerimientos cambiantes del cliente. Como plantean Boehm y Turner (2003), cada enfoque es adecuado para diferentes tipos de software. Por lo general, uno necesita encontrar un equilibrio entre procesos dirigidos por un plan y procesos ágiles.

Aunque no hay un proceso de software "ideal", en muchas organizaciones sí existe un ámbito para mejorar el proceso de software. Los procesos quizás incluyan técnicas obsoletas o tal vez no aprovechen las mejores prácticas en la industria de la ingeniería de software. En efecto, muchas organizaciones aún no sacan ventaja de los métodos de la ingeniería de software en su desarrollo de software.

Los procesos de software pueden mejorarse con la estandarización de los procesos, donde se reduce la diversidad en los procesos de software en una organización. Esto conduce a mejorar la comunicación, a reducir el tiempo de capacitación, y a que el soporte de los procesos automatizados sea más económico. La estandarización también representa un primer paso importante tanto en la introducción de nuevos métodos y técnicas de ingeniería de software, como en sus buenas prácticas. En el capítulo 26 se analiza con más detalle la mejora en el proceso de software.

2.1 Modelos de proceso de software

Como se explicó en el capítulo 1, un modelo de proceso de software es una representación simplificada de este proceso. Cada modelo del proceso representa a otro desde una particular perspectiva y, por lo tanto, ofrece sólo información parcial acerca de dicho proceso. Por ejemplo, un modelo de actividad del proceso muestra las actividades y su secuencia, pero quizá sin presentar los roles de las personas que intervienen en esas actividades. En esta sección se introducen algunos modelos de proceso muy generales (en ocasiones llamados "paradigmas de proceso") y se muestran desde una perspectiva arquitectónica. En otras palabras, se ve el marco (framework) del proceso, pero no los detalles de las actividades específicas.

Tales modelos genéricos no son descripciones definitivas de los procesos de software. Más bien, son abstracciones del proceso que se utilizan para explicar los diferentes enfoques del desarrollo de software. Se pueden considerar marcos del proceso que se extienden y se adaptan para crear procesos más específicos de ingeniería de software.

Los modelos del proceso que se examinan aquí son:

1. *El modelo en cascada* (*waterfall*) Éste toma las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución y, luego, los representa como fases separadas del proceso, tal como especificación de requerimientos, diseño de software, implementación, pruebas, etcétera.

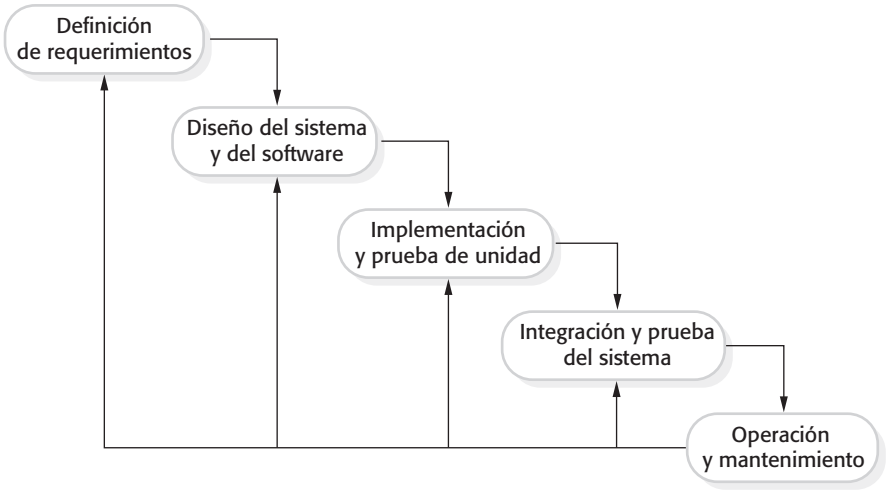


Figura 2.1 El modelo en cascada

2. *Desarrollo incremental* Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos), y cada versión añade funcionalidad a la versión anterior.

3. *Ingeniería de software orientada a la reutilización* Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en la integración de estos componentes en un sistema, en vez de desarrollarlo desde cero.

Dichos modelos no son mutuamente excluyentes y con frecuencia se usan en conjunto, sobre todo para el desarrollo de grandes sistemas. Para este tipo de sistemas, tiene sentido combinar algunas de las mejores características de los modelos de desarrollo en cascada e incremental. Se necesita contar con información sobre los requerimientos esenciales del sistema para diseñar la arquitectura de software que apoye dichos requerimientos. No puede desarrollarse de manera incremental. Los subsistemas dentro de un sistema más grande se desarrollan usando diferentes enfoques. Partes del sistema que son bien comprendidas pueden especificarse y desarrollarse al utilizar un proceso basado en cascada. Partes del sistema que por adelantado son difíciles de especificar, como la interfaz de usuario, siempre deben desarrollarse con un enfoque incremental.

2.1.1 El modelo en cascada

El primer modelo publicado sobre el proceso de desarrollo de software se derivó a partir de procesos más generales de ingeniería de sistemas (Royce, 1970). Este modelo se ilustra en la figura 2.1. Debido al paso de una fase en cascada a otra, este modelo se conoce como "modelo en cascada" o ciclo de vida del software. El modelo en cascada es un ejemplo de un proceso dirigido por un plan; en principio, usted debe planear y programar todas las actividades del proceso, antes de comenzar a trabajar con ellas.

Las principales etapas del modelo en cascada reflejan directamente las actividades fundamentales del desarrollo:

1. *Análisis y definición de requerimientos* Los servicios, las restricciones y las metas del sistema se establecen mediante consulta a los usuarios del sistema. Luego, se definen con detalle y sirven como una especificación del sistema.
2. *Diseño del sistema y del software* El proceso de diseño de sistemas asigna los requerimientos, para sistemas de hardware o de software, al establecer una arquitectura de sistema global. El diseño del software implica identificar y describir las abstracciones fundamentales del sistema de software y sus relaciones.
3. *Implementación y prueba de unidad* Durante esta etapa, el diseño de software se realiza como un conjunto de programas o unidades del programa. La prueba de unidad consiste en verificar que cada unidad cumpla con su especificación.
4. *Integración y prueba de sistema* Las unidades del programa o los programas individuales se integran y prueban como un sistema completo para asegurarse de que se cumplan los requerimientos de software. Después de probarlo, se libera el sistema de software al cliente.
5. *Operación y mantenimiento* Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida, donde el sistema se instala y se pone en práctica. El mantenimiento incluye corregir los errores que no se detectaron en etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema e incrementar los servicios del sistema conforme se descubren nuevos requerimientos.

En principio, el resultado de cada fase consiste en uno o más documentos que se autorizaron (“firmaron”). La siguiente fase no debe comenzar sino hasta que termine la fase previa. En la práctica, dichas etapas se traslapan y se nutren mutuamente de información. Durante el diseño se identifican los problemas con los requerimientos. En la codificación se descubren problemas de diseño, y así sucesivamente. El proceso de software no es un simple modelo lineal, sino que implica retroalimentación de una fase a otra. Entonces, es posible que los documentos generados en cada fase deban modificarse para reflejar los cambios que se realizan.

Debido a los costos de producción y aprobación de documentos, las iteraciones suelen ser onerosas e implicar un rediseño significativo. Por lo tanto, después de un pequeño número de iteraciones, es normal detener partes del desarrollo, como la especificación, y continuar con etapas de desarrollo posteriores. Los problemas se dejan para una resolución posterior, se ignoran o se programan. Este freno prematuro de los requerimientos quizá signifique que el sistema no hará lo que el usuario desea. También podría conducir a sistemas mal estructurados conforme los problemas de diseño se evadan con la implementación de trucos.

Durante la fase final del ciclo de vida (operación y mantenimiento), el software se pone en servicio. Se descubren los errores y las omisiones en los requerimientos originales del software. Surgen los errores de programa y diseño, y se detecta la necesidad de nueva funcionalidad. Por lo tanto, el sistema debe evolucionar para mantenerse útil. Hacer tales cambios (mantenimiento de software) puede implicar la repetición de etapas anteriores del proceso.

Ingeniería de software de cuarto limpio

Un ejemplo del proceso de desarrollo formal, diseñado originalmente por IBM, es el proceso de cuarto limpio (*cleanroom*). En el proceso de cuarto limpio, cada incremento de software se especifica formalmente y tal especificación se transforma en una implementación. La exactitud del software se demuestra mediante un enfoque formal. No hay prueba de unidad para defectos en el proceso y la prueba del sistema se enfoca en la valoración de la fiabilidad del sistema.

El objetivo del proceso de cuarto limpio es obtener un software con cero defectos, de modo que los sistemas que se entreguen cuenten con un alto nivel de fiabilidad.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

El modelo en cascada es consecuente con otros modelos del proceso de ingeniería y en cada fase se produce documentación. Esto hace que el proceso sea visible, de modo que los administradores monitoricen el progreso contra el plan de desarrollo. Su principal problema es la partición inflexible del proyecto en distintas etapas. Tienen que establecerse compromisos en una etapa temprana del proceso, lo que dificulta responder a los requerimientos cambiantes del cliente.

En principio, el modelo en cascada sólo debe usarse cuando los requerimientos se entiendan bien y sea improbable el cambio radical durante el desarrollo del sistema. Sin embargo, el modelo en cascada refleja el tipo de proceso utilizado en otros proyectos de ingeniería. Como es más sencillo emplear un modelo de gestión común durante todo el proyecto, aún son de uso común los procesos de software basados en el modelo en cascada.

Una variación importante del modelo en cascada es el desarrollo de sistemas formales, donde se crea un modelo matemático para una especificación del sistema. Después se corrige este modelo, mediante transformaciones matemáticas que preservan su consistencia en un código ejecutable. Con base en la suposición de que son correctas sus transformaciones matemáticas, se puede aseverar, por lo tanto, que un programa generado de esta forma es consecuente con su especificación.

Los procesos formales de desarrollo, como el que se basa en el método B (Schneider, 2001; Wordsworth, 1996) son muy adecuados para el desarrollo de sistemas que cuenten con rigurosos requerimientos de seguridad, fiabilidad o protección. El enfoque formal simplifica la producción de un caso de protección o seguridad. Esto demuestra a los clientes o reguladores que el sistema en realidad cumple sus requerimientos de protección o seguridad.

Los procesos basados en transformaciones formales se usan por lo general sólo en el desarrollo de sistemas críticos para protección o seguridad. Requieren experiencia especializada. Para la mayoría de los sistemas, este proceso no ofrece costo/beneficio significativos sobre otros enfoques en el desarrollo de sistemas.

2.1.2 Desarrollo incremental

El desarrollo incremental se basa en la idea de diseñar una implementación inicial, exponer ésta al comentario del usuario, y luego desarrollarla en sus diversas versiones hasta producir un sistema adecuado (figura 2.2). Las actividades de especificación, desarrollo

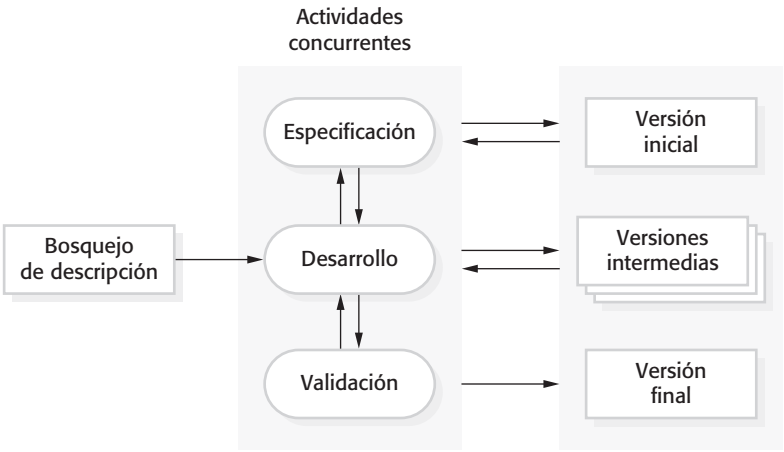


Figura 2.2 Desarrollo incremental

y validación están entrelazadas en vez de separadas, con rápida retroalimentación a través de las actividades.

El desarrollo de software incremental, que es una parte fundamental de los enfoques ágiles, es mejor que un enfoque en cascada para la mayoría de los sistemas empresariales, de comercio electrónico y personales. El desarrollo incremental refleja la forma en que se resuelven problemas. Rara vez se trabaja por adelantado una solución completa del problema, más bien se avanza en una serie de pasos hacia una solución y se retrocede cuando se detecta que se cometieron errores. Al desarrollar el software de manera incremental, resulta más barato y fácil realizar cambios en el software conforme éste se diseña.

Cada incremento o versión del sistema incorpora algunas de las funciones que necesita el cliente. Por lo general, los primeros incrementos del sistema incluyen la función más importante o la más urgente. Esto significa que el cliente puede evaluar el desarrollo del sistema en una etapa relativamente temprana, para constatar si se entrega lo que se requiere. En caso contrario, sólo el incremento actual debe cambiarse y, posiblemente, definir una nueva función para incrementos posteriores.

Comparado con el modelo en cascada, el desarrollo incremental tiene tres beneficios importantes:

1. Se reduce el costo de adaptar los requerimientos cambiantes del cliente. La cantidad de análisis y la documentación que tiene que reelaborarse son mucho menores de lo requerido con el modelo en cascada.
2. Es más sencillo obtener retroalimentación del cliente sobre el trabajo de desarrollo que se realizó. Los clientes pueden comentar las demostraciones del software y darse cuenta de cuánto se ha implementado. Los clientes encuentran difícil juzgar el avance a partir de documentos de diseño de software.
3. Es posible que sea más rápida la entrega e implementación de software útil al cliente, aun si no se ha incluido toda la funcionalidad. Los clientes tienen posibilidad de usar y ganar valor del software más temprano de lo que sería posible con un proceso en cascada.

Problemas con el desarrollo incremental

Aunque el desarrollo incremental tiene muchas ventajas, no está exento de problemas. La principal causa de la dificultad es el hecho de que las grandes organizaciones tienen procedimientos burocráticos que han evolucionado con el tiempo y pueden suscitar falta de coordinación entre dichos procedimientos y un proceso iterativo o ágil más informal.

En ocasiones, tales procedimientos se hallan ahí por buenas razones: por ejemplo, pueden existir procedimientos para garantizar que el software implementa de manera adecuada regulaciones externas (en Estados Unidos, por ejemplo, las regulaciones de contabilidad Sarbanes-Oxley). El cambio de tales procedimientos podría resultar imposible, de manera que los conflictos son inevitables.

http://www.SoftwareEngineering-9.com/Web/IncrementalDev/

El desarrollo incremental ahora es en cierta forma el enfoque más común para el desarrollo de sistemas de aplicación. Este enfoque puede estar basado en un plan, ser ágil o, más usualmente, una mezcla de dichos enfoques. En un enfoque basado en un plan se identifican por adelantado los incrementos del sistema; si se adopta un enfoque ágil, se detectan los primeros incrementos, aunque el desarrollo de incrementos posteriores depende del avance y las prioridades del cliente.

Desde una perspectiva administrativa, el enfoque incremental tiene dos problemas:

1. El proceso no es visible. Los administradores necesitan entregas regulares para medir el avance. Si los sistemas se desarrollan rápidamente, resulta poco efectivo en términos de costos producir documentos que reflejen cada versión del sistema.
2. La estructura del sistema tiende a degradarse conforme se tienen nuevos incrementos. A menos que se gaste tiempo y dinero en la refactorización para mejorar el software, el cambio regular tiende a corromper su estructura. La incorporación de más cambios de software se vuelve cada vez más difícil y costosa.

Los problemas del desarrollo incremental se tornan particularmente agudos para sistemas grandes, complejos y de larga duración, donde diversos equipos desarrollan diferentes partes del sistema. Los grandes sistemas necesitan de un marco o una arquitectura estable y es necesario definir con claridad, respecto a dicha arquitectura, las responsabilidades de los distintos equipos que trabajan en partes del sistema. Esto debe planearse por adelantado en vez de desarrollarse de manera incremental.

Se puede desarrollar un sistema incremental y exponerlo a los clientes para su comentario, sin realmente entregarlo e implementarlo en el entorno del cliente. La entrega y la implementación incrementales significan que el software se usa en procesos operacionales reales. Esto no siempre es posible, ya que la experimentación con un nuevo software llega a alterar los procesos empresariales normales. En la sección 2.3.2 se estudian las ventajas y desventajas de la entrega incremental.

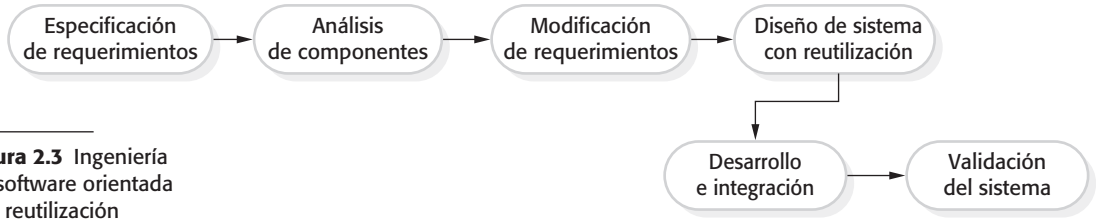


Figura 2.3 Ingeniería de software orientada a la reutilización

2.1.3 Ingeniería de software orientada a la reutilización

En la mayoría de los proyectos de software hay cierta reutilización de software. Sucede con frecuencia de manera informal, cuando las personas que trabajan en el proyecto conocen diseños o códigos que son similares a lo que se requiere. Los buscan, los modifican según se necesite y los incorporan en sus sistemas.

Esta reutilización informal ocurre independientemente del proceso de desarrollo que se emplee. Sin embargo, en el siglo XXI, los procesos de desarrollo de software que se enfocaban en la reutilización de software existente se utilizan ampliamente. Los enfoques orientados a la reutilización se apoyan en una gran base de componentes de software reutilizable y en la integración de marcos para la composición de dichos componentes. En ocasiones, tales componentes son sistemas por derecho propio (sistemas comerciales, off-the-shelf o COTS) que pueden mejorar la funcionalidad específica, como el procesador de textos o la hoja de cálculo.

En la figura 2.3 se muestra un modelo del proceso general para desarrollo basado en reutilización. Aunque la etapa inicial de especificación de requerimientos y la etapa de validación se comparan con otros procesos de software en un proceso orientado a la reutilización, las etapas intermedias son diferentes. Dichas etapas son:

1. *Análisis de componentes* Dada la especificación de requerimientos, se realiza una búsqueda de componentes para implementar dicha especificación. Por lo general, no hay coincidencia exacta y los componentes que se usan proporcionan sólo parte de la funcionalidad requerida.

2. *Modificación de requerimientos* Durante esta etapa se analizan los requerimientos usando información de los componentes descubiertos. Luego se modifican para reflejar los componentes disponibles. Donde las modificaciones son imposibles, puede regresarse a la actividad de análisis de componentes para buscar soluciones alternativas.

3. *Diseño de sistema con reutilización* Durante esta fase se diseña el marco conceptual del sistema o se reutiliza un marco conceptual existente. Los creadores toman en cuenta los componentes que se reutilizan y organizan el marco de referencia para atenderlo. Es posible que deba diseñarse algo de software nuevo, si no están disponibles los componentes reutilizables.

4. *Desarrollo e integración* Se diseña el software que no puede procurarse de manera externa, y se integran los componentes y los sistemas COTS para crear el nuevo sistema. La integración del sistema, en este modelo, puede ser parte del proceso de desarrollo, en vez de una actividad independiente.

Existen tres tipos de componentes de software que pueden usarse en un proceso orientado a la reutilización:

1. Servicios Web que se desarrollan en concordancia para atender servicios estándares y que están disponibles para la invocación remota.
2. Colecciones de objetos que se desarrollan como un paquete para su integración con un marco de componentes como .NET o J2EE.
3. Sistemas de software independientes que se configuran para usar en un entorno particular.

La ingeniería de software orientada a la reutilización tiene la clara ventaja de reducir la cantidad de software a desarrollar y, por lo tanto, la de disminuir costos y riesgos; por lo general, también conduce a entregas más rápidas del software. Sin embargo, son inevitables los compromisos de requerimientos y esto conduciría hacia un sistema que no cubra las necesidades reales de los usuarios. Más aún, se pierde algo de control sobre la evolución del sistema, conforme las nuevas versiones de los componentes reutilizables no estén bajo el control de la organización que los usa.

La reutilización de software es muy importante y en la tercera parte del libro se dedican varios capítulos a este tema. En el capítulo 16 se tratan los conflictos generales de la reutilización de software y la reutilización de COTS, en los capítulos 17 y 18 se estudia la ingeniería de software basada en componentes, y en el capítulo 19 se explican los sistemas orientados al servicio.

2.2 Actividades del proceso

Los procesos de software real son secuencias entrelazadas de actividades técnicas, colaborativas y administrativas con la meta general de especificar, diseñar, implementar y probar un sistema de software. Los desarrolladores de software usan en su trabajo diferentes herramientas de software. Las herramientas son útiles particularmente para dar apoyo a la edición de distintos tipos de documento y para manejar el inmenso volumen de información detallada que se reproduce en un gran proyecto de software.

Las cuatro actividades básicas de proceso de especificación, desarrollo, validación y evolución se organizan de diversa manera en diferentes procesos de desarrollo. En el modelo en cascada se organizan en secuencia, mientras que se entrelazan en el desarrollo incremental. La forma en que se llevan a cabo estas actividades depende del tipo de software, del personal y de la inclusión de estructuras organizativas. En la programación extrema, por ejemplo, las especificaciones se escriben en tarjetas. Las pruebas son ejecutables y se desarrollan antes del programa en sí. La evolución incluye la reestructuración o refactorización sustancial del sistema.

2.2.1 Especificación del software

La especificación del software o la ingeniería de requerimientos consisten en el proceso de comprender y definir qué servicios se requieren del sistema, así como la identificación de las restricciones sobre la operación y el desarrollo del sistema. La ingeniería de requerimientos es una etapa particularmente crítica del proceso de software, ya que los

Herramientas de desarrollo de software

Las herramientas de desarrollo del software (llamadas en ocasiones herramientas de Ingeniería de Software Asistido por Computadora o CASE, por las siglas de *Computer-Aided Software Engineering*) son programas usados para apoyar las actividades del proceso de la ingeniería de software. En consecuencia, estas herramientas incluyen editores de diseño, diccionarios de datos, compiladores, depuradores (*debuggers*), herramientas de construcción de sistema, etcétera.

Las herramientas de software ofrecen apoyo de proceso al automatizar algunas actividades del proceso y brindar información sobre el software que se desarrolla. Los ejemplos de actividades susceptibles de automatizarse son:

- Desarrollo de modelos de sistemas gráficos, como parte de la especificación de requerimientos o del diseño del software.
- Generación de código a partir de dichos modelos de sistemas gráficos.
- Producción de interfaces de usuario a partir de una descripción de interfaz gráfica, creada por el usuario de manera interactiva.
- Depuración del programa mediante el suministro de información sobre un programa que se ejecuta.
- Traducción automatizada de programas escritos, usando una versión anterior de un lenguaje de programación para tener una versión más reciente.

Las herramientas pueden combinarse en un marco llamado ambiente de desarrollo interactivo o IDE (por las siglas de *Interactive Development Environment*). Esto ofrece un conjunto común de facilidades, que usan las herramientas para comunicarse y operar con mayor destreza en una forma integrada. El ECLIPSE IDE se usa ampliamente y se diseñó para incorporar muchos tipos diferentes de herramientas de software.

http://www.SoftwareEngineering-9.com/Web/CASE/

errores en esta etapa conducen de manera inevitable a problemas posteriores tanto en el diseño como en la implementación del sistema.

El proceso de ingeniería de requerimientos (figura 2.4) se enfoca en producir un documento de requerimientos convenido que especifique los requerimientos de los interesados que cumplirá el sistema. Por lo general, los requerimientos se presentan en dos niveles de detalle. Los usuarios finales y clientes necesitan un informe de requerimientos de alto nivel; los desarrolladores de sistemas precisan una descripción más detallada del sistema.

Existen cuatro actividades principales en el proceso de ingeniería de requerimientos:

1. *Estudio de factibilidad* Se realiza una estimación sobre si las necesidades identificadas del usuario se cubren con las actuales tecnologías de software y hardware. El estudio considera si el sistema propuesto tendrá un costo-beneficio desde un punto de vista empresarial, y si éste puede desarrollarse dentro de las restricciones presupuestales existentes. Un estudio de factibilidad debe ser rápido y relativamente barato. El resultado debe informar la decisión respecto a si se continúa o no continúa con un análisis más detallado.
2. *Obtención y análisis de requerimientos* Éste es el proceso de derivar los requerimientos del sistema mediante observación de los sistemas existentes, discusiones con los usuarios y proveedores potenciales, análisis de tareas, etcétera. Esto puede incluir el desarrollo de uno o más modelos de sistemas y prototipos, lo que ayuda a entender el sistema que se va a especificar.

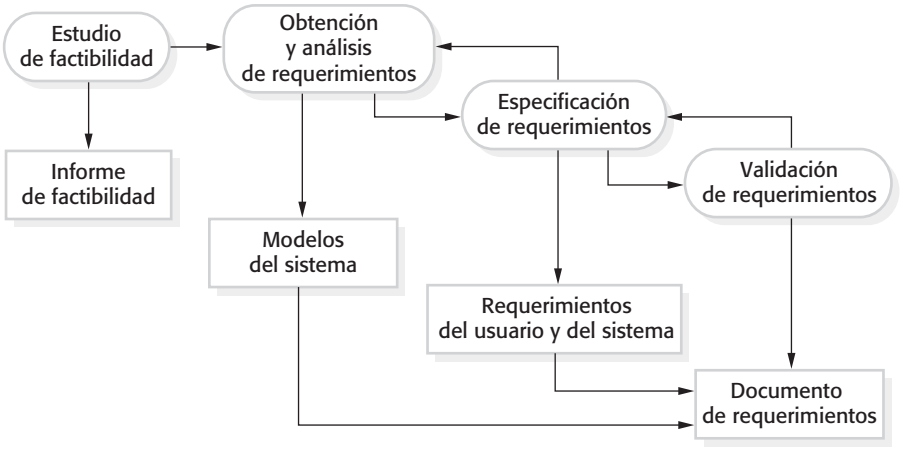


Figura 2.4 Proceso de ingeniería de requerimientos

3. *Especificación de requerimientos* Consiste en la actividad de transcribir la información recopilada durante la actividad de análisis, en un documento que define un conjunto de requerimientos. En este documento se incluyen dos clases de requerimientos. Los requerimientos del usuario son informes abstractos de requerimientos del sistema para el cliente y el usuario final del sistema; y los requerimientos de sistema son una descripción detallada de la funcionalidad a ofrecer.

4. *Validación de requerimientos* Esta actividad verifica que los requerimientos sean realistas, coherentes y completos. Durante este proceso es inevitable descubrir errores en el documento de requerimientos. En consecuencia, deberían modificarse con la finalidad de corregir dichos problemas.

Desde luego, las actividades en el proceso de requerimientos no se realizan simplemente en una secuencia estricta. El análisis de requerimientos continúa durante la definición y especificación, y a lo largo del proceso salen a la luz nuevos requerimientos; por lo tanto, las actividades de análisis, definición y especificación están vinculadas. En los métodos ágiles, como programación extrema, los requerimientos se desarrollan de manera incremental según las prioridades del usuario, en tanto que la obtención de requerimientos proviene de los usuarios que son parte del equipo de desarrollo.

2.2.2 Diseño e implementación del software

La etapa de implementación de desarrollo del software corresponde al proceso de convertir una especificación del sistema en un sistema ejecutable. Siempre incluye procesos de diseño y programación de software, aunque también puede involucrar la corrección en la especificación del software, si se utiliza un enfoque incremental de desarrollo.

Un diseño de software se entiende como una descripción de la estructura del software que se va a implementar, los modelos y las estructuras de datos utilizados por el sistema, las interfaces entre componentes del sistema y, en ocasiones, los algoritmos usados. Los diseñadores no llegan inmediatamente a una creación terminada, sino que desarrollan el diseño de manera iterativa. Agregan formalidad y detalle conforme realizan su diseño con *backtracking* (vuelta atrás) constante para corregir diseños anteriores.

La figura 2.5 es un modelo abstracto de este proceso, que ilustra las entradas al proceso de diseño, las actividades del proceso y los documentos generados como salidas de este proceso.

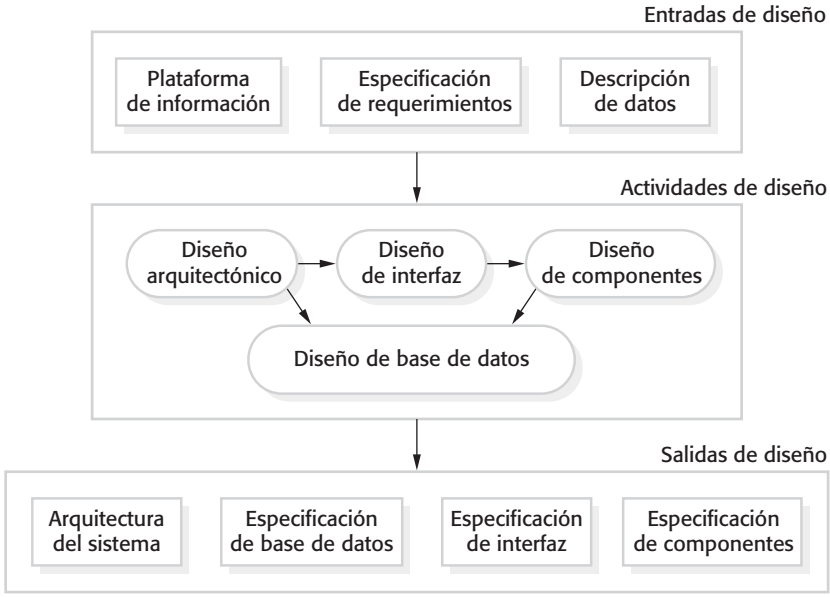


Figura 2.5 Modelo general del proceso de diseño

El diagrama sugiere que las etapas del proceso de diseño son secuenciales. De hecho, las actividades de proceso de diseño están vinculadas. En todos los procesos de diseño es inevitable la retroalimentación de una etapa a otra y la consecuente reelaboración del diseño.

La mayoría del software tiene interfaz junto con otros sistemas de software. En ellos se incluyen sistema operativo, base de datos, *middleware* y otros sistemas de aplicación. Éstos constituyen la "plataforma de software", es decir, el entorno donde se ejecutará el software. La información sobre esta plataforma es una entrada esencial al proceso de diseño, así que los diseñadores tienen que decidir la mejor forma de integrarla con el entorno de software. La especificación de requerimientos es una descripción de la funcionalidad que debe brindar el software, en conjunción con sus requerimientos de rendimiento y confiabilidad. Si el sistema debe procesar datos existentes, entonces en la especificación de la plataforma se incluirá la descripción de tales datos; de otro modo, la descripción de los datos será una entrada al proceso de diseño, de manera que se defina la organización del sistema de datos.

Las actividades en el proceso de diseño varían dependiendo del tipo de sistema a desarrollar. Por ejemplo, los sistemas de tiempo real precisan del diseño de temporización, pero sin incluir una base de datos, por lo que no hay que integrar un diseño de base de datos. La figura 2.5 muestra cuatro actividades que podrían formar parte del proceso de diseño para sistemas de información:

1. *Diseño arquitectónico*, aquí se identifica la estructura global del sistema, los principales componentes (llamados en ocasiones subsistemas o módulos), sus relaciones y cómo se distribuyen.
2. *Diseño de interfaz*, en éste se definen las interfaces entre los componentes de sistemas. Esta especificación de interfaz no tiene que presentar ambigüedades. Con una interfaz precisa, es factible usar un componente sin que otros tengan que saber cómo se implementó. Una vez que se acuerdan las especificaciones de interfaz, los componentes se diseñan y se desarrollan de manera concurrente.

Métodos estructurados

Los métodos estructurados son un enfoque al diseño de software donde se definen los modelos gráficos que hay que desarrollar, como parte del proceso de diseño. El método también define un proceso para diseñar los modelos y las reglas que se aplican a cada tipo de modelo. Los métodos estructurados conducen a documentación estandarizada para un sistema y son muy útiles al ofrecer un marco de desarrollo para los creadores de software con menor experiencia.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

3. *Diseño de componentes*, en él se toma cada componente del sistema y se diseña cómo funcionará. Esto puede ser un simple dato de la funcionalidad que se espera implementar, y al programador se le deja el diseño específico. Como alternativa, habría una lista de cambios a realizar sobre un componente que se reutiliza o sobre un modelo de diseño detallado. El modelo de diseño sirve para generar en automático una implementación.
4. *Diseño de base de datos*, donde se diseñan las estructuras del sistema de datos y cómo se representarán en una base de datos. De nuevo, el trabajo aquí depende de si una base de datos se reutilizará o se creará una nueva.

Tales actividades conducen a un conjunto de salidas de diseño, que también se muestran en la figura 2.5. El detalle y la representación de las mismas varían considerablemente. Para sistemas críticos, deben producirse documentos de diseño detallados que establezcan descripciones exactas del sistema. Si se usa un enfoque dirigido por un modelo, dichas salidas serían sobre todo diagramas. Donde se usen métodos ágiles de desarrollo, las salidas del proceso de diseño no podrían ser documentos de especificación separados, sino que tendrían que representarse en el código del programa.

Los métodos estructurados para el diseño se desarrollaron en las décadas de 1970 y 1980, y fueron precursores del UML y del diseño orientado a objetos (Budgen, 2003). Se apoyan en la producción de modelos gráficos del sistema y, en muchos casos, generan instantáneamente un código a partir de dichos modelos. El desarrollo dirigido por modelo (MDD) o la ingeniería dirigida por modelo (Schmidt, 2006), donde se crean modelos de software a diferentes niveles de abstracción, es una evolución de los métodos estructurados. En el MDD hay mayor énfasis en los modelos arquitectónicos con una separación entre modelos abstractos independientes de implementación y modelos específicos de implementación. Los modelos se desarrollan con detalle suficiente, de manera que el sistema ejecutable puede generarse a partir de ellos. En el capítulo 5 se estudia este enfoque de desarrollo.

El diseño de un programa para implementar el sistema se sigue naturalmente de los procesos de elaboración del sistema. Aunque algunas clases de programa, como los sistemas críticos para la seguridad, por lo general se diseñan con detalle antes de comenzar cualquier implementación, es más común que se entrelacen en etapas posteriores del diseño y el desarrollo del programa. Las herramientas de desarrollo de software se usan para generar un programa de "esqueleto" a partir de un diseño. Esto incluye un código para definir e implementar interfaces y, en muchos casos, el desarrollador sólo necesita agregar detalles de la operación de cada componente del programa.

La programación es una actividad personal y no hay un proceso que se siga de manera general. Algunos programadores comienzan con componentes que entienden, los desarrollan y, luego, cambian hacia componentes que entienden menos. Otros toman el enfoque opuesto,

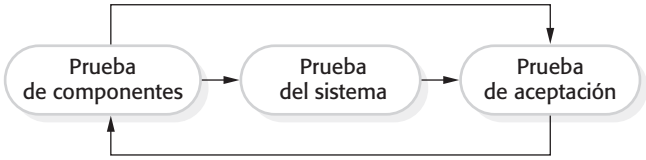


Figura 2.6 Etapas de pruebas

y dejan hasta el último los componentes familiares, porque saben cómo diseñarlos. A algunos desarrolladores les agrada definir con anticipación datos en el proceso, que luego usan para impulsar el desarrollo del programa; otros dejan datos sin especificar tanto como sea posible.

Por lo general, los programadores realizan algunas pruebas del código que desarrollaron. Esto revela con frecuencia defectos del programa que deben eliminarse del programa. A esta actividad se le llama depuración (*debugging*). La prueba de defectos y la depuración son procesos diferentes. La primera establece la existencia de defectos, en tanto que la segunda se dedica a localizar y corregir dichos defectos.

Cuando se depura, uno debe elaborar una hipótesis sobre el comportamiento observable del programa y, luego, poner a prueba dichas hipótesis con la esperanza de encontrar la falla que causó la salida anómala. Poner a prueba las hipótesis quizá requiera rastrear manualmente el código del programa; o bien, tal vez se necesiten nuevos casos de prueba para localizar el problema. Con la finalidad de apoyar el proceso de depuración, se deben utilizar herramientas interactivas que muestren valores intermedios de las variables del programa, así como el rastro de las instrucciones ejecutadas.

2.2.3 Validación de software

La validación de software o, más generalmente, su verificación y validación (V&V), se crea para mostrar que un sistema cumple tanto con sus especificaciones como con las expectativas del cliente. Las pruebas del programa, donde el sistema se ejecuta a través de datos de prueba simulados, son la principal técnica de validación. Esta última también puede incluir procesos de comprobación, como inspecciones y revisiones en cada etapa del proceso de software, desde la definición de requerimientos del usuario hasta el desarrollo del programa. Dada la predominancia de las pruebas, se incurre en la mayoría de los costos de validación durante la implementación y después de ésta.

Con excepción de los programas pequeños, los sistemas no deben ponerse a prueba como una unidad monolítica. La figura 2.6 muestra un proceso de prueba en tres etapas, donde los componentes del sistema se ponen a prueba; luego, se hace lo mismo con el sistema integrado y, finalmente, el sistema se pone a prueba con los datos del cliente. De manera ideal, los defectos de los componentes se detectan oportunamente en el proceso, en tanto que los problemas de interfaz se localizan cuando el sistema se integra. Sin embargo, conforme se descubran los defectos, el programa deberá depurarse y esto quizá requiera la repetición de otras etapas en el proceso de pruebas. Los errores en los componentes del programa pueden salir a la luz durante las pruebas del sistema. En consecuencia, el proceso es iterativo, con información retroalimentada desde etapas posteriores hasta las partes iniciales del proceso.

Las etapas en el proceso de pruebas son:

1. *Prueba de desarrollo* Las personas que desarrollan el sistema ponen a prueba los componentes que constituyen el sistema. Cada componente se prueba de manera independiente, es decir, sin otros componentes del sistema. Éstos pueden ser simples

entidades, como funciones o clases de objeto, o agrupamientos coherentes de dichas entidades. Por lo general, se usan herramientas de automatización de pruebas, como JUnit (Massol y Husted, 2003), que pueden volver a correr pruebas de componentes cuando se crean nuevas versiones del componente.

2. *Pruebas del sistema* Los componentes del sistema se integran para crear un sistema completo. Este proceso tiene la finalidad de descubrir errores que resulten de interacciones no anticipadas entre componentes y problemas de interfaz de componente, así como de mostrar que el sistema cubre sus requerimientos funcionales y no funcionales, y poner a prueba las propiedades emergentes del sistema. Para sistemas grandes, esto puede ser un proceso de múltiples etapas, donde los componentes se conjuntan para formar subsistemas que se ponen a prueba de manera individual, antes de que dichos subsistemas se integren para establecer el sistema final.

3. *Pruebas de aceptación* Ésta es la etapa final en el proceso de pruebas, antes de que el sistema se acepte para uso operacional. El sistema se pone a prueba con datos suministrados por el cliente del sistema, en vez de datos de prueba simulados. Las pruebas de aceptación revelan los errores y las omisiones en la definición de requerimientos del sistema, ya que los datos reales ejercitan el sistema en diferentes formas a partir de los datos de prueba. Asimismo, las pruebas de aceptación revelan problemas de requerimientos, donde las instalaciones del sistema en realidad no cumplan las necesidades del usuario o cuando sea inaceptable el rendimiento del sistema.

Por lo general, los procesos de desarrollo y de pruebas de componentes están entrelazados. Los programadores construyen sus propios datos de prueba y experimentan el código de manera incremental conforme lo desarrollan. Éste es un enfoque económicamente sensible, ya que el programador conoce el componente y, por lo tanto, es el más indicado para generar casos de prueba.

Si se usa un enfoque incremental para el desarrollo, cada incremento debe ponerse a prueba conforme se diseña, y tales pruebas se basan en los requerimientos para dicho incremento. En programación extrema, las pruebas se desarrollan junto con los requerimientos antes de comenzar el desarrollo. Esto ayuda a los examinadores y desarrolladores a comprender los requerimientos, y garantiza que no haya demoras conforme se creen casos de prueba.

Cuando se usa un proceso de software dirigido por un plan (como en el desarrollo de sistemas críticos), las pruebas se realizan mediante un conjunto de planes de prueba. Un equipo independiente de examinadores trabaja con base en dichos planes de prueba preformulados, que se desarrollaron a partir de la especificación y el diseño del sistema. La figura 2.7 ilustra cómo se vinculan los planes de prueba entre las actividades de pruebas y desarrollo. A esto se le conoce en ocasiones como modelo V de desarrollo (colóquelo de lado para distinguir la V).

En ocasiones, a las pruebas de aceptación se les identifica como “pruebas alfa”. Los sistemas a la medida se desarrollan sólo para un cliente. El proceso de prueba alfa continúa hasta que el desarrollador del sistema y el cliente estén de acuerdo en que el sistema entregado es una implementación aceptable de los requerimientos.

Cuando un sistema se marca como producto de software, se utiliza con frecuencia un proceso de prueba llamado “prueba beta”. Ésta incluye entregar un sistema a algunos clientes potenciales que están de acuerdo con usar ese sistema. Ellos reportan los problemas a los desarrolladores del sistema. Dicho informe expone el producto a uso real y detecta errores que no anticiparon los constructores del sistema. Después de esta retroalimentación, el sistema se modifica y libera, ya sea para más pruebas beta o para su venta general.

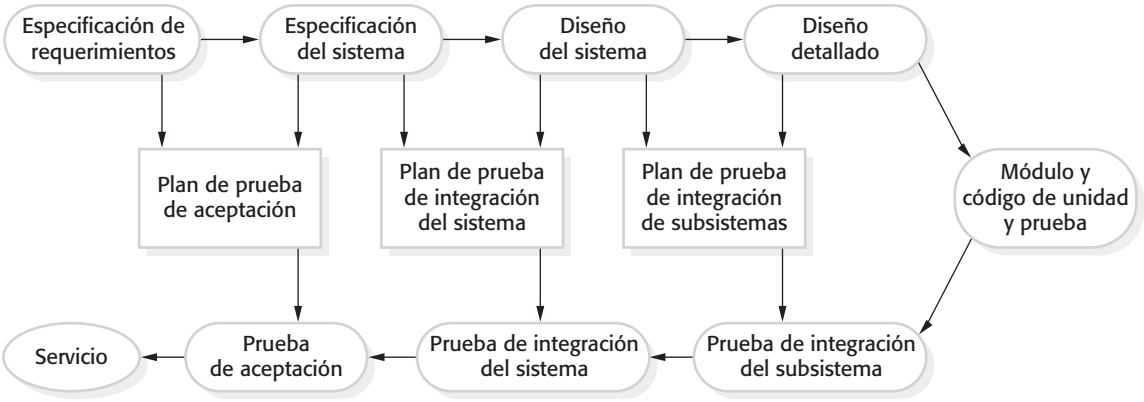


Figura 2.7 Probando fases en un proceso de software dirigido por un plan

2.2.4 Evolución del software

La flexibilidad de los sistemas de software es una de las razones principales por las que cada vez más software se incorpora en los sistemas grandes y complejos. Una vez tomada la decisión de fabricar hardware, resulta muy costoso hacer cambios a su diseño. Sin embargo, en cualquier momento durante o después del desarrollo del sistema, pueden hacerse cambios al software. Incluso los cambios mayores son todavía más baratos que los correspondientes cambios al hardware del sistema.

En la historia, siempre ha habido división entre el proceso de desarrollo del software y el proceso de evolución del software (mantenimiento de software). Las personas consideran el desarrollo de software como una actividad creativa, en la cual se diseña un sistema de software desde un concepto inicial y a través de un sistema de trabajo. No obstante, consideran en ocasiones el mantenimiento del software como insulso y poco interesante. Aunque en la mayoría de los casos los costos del mantenimiento son varias veces los costos iniciales de desarrollo, los procesos de mantenimiento se consideran en ocasiones como menos desafiantes que el desarrollo de software original.

Esta distinción entre desarrollo y mantenimiento es cada vez más irrelevante. Es muy difícil que cualquier sistema de software sea un sistema completamente nuevo, y tiene mucho más sentido ver el desarrollo y el mantenimiento como un continuo. En lugar de dos procesos separados, es más realista pensar en la ingeniería de software como un proceso evolutivo (figura 2.8), donde el software cambia continuamente a lo largo de su vida, en función de los requerimientos y las necesidades cambiantes del cliente.

2.3 Cómo enfrentar el cambio

El cambio es inevitable en todos los grandes proyectos de software. Los requerimientos del sistema varían conforme la empresa procura que el sistema responda a presiones externas y se modifican las prioridades administrativas. A medida que se ponen a disposición nuevas tecnologías, surgen nuevas posibilidades de diseño e implementación. Por ende, cualquiera que sea el modelo del proceso de software utilizado, es esencial que ajuste los cambios al software a desarrollar.

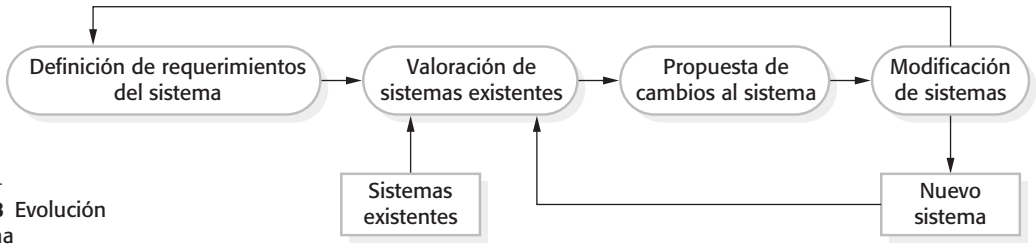


Figura 2.8 Evolución del sistema

El cambio se agrega a los costos del desarrollo de software debido a que, por lo general, significa que el trabajo ya terminado debe volver a realizarse. A esto se le llama rehacer. Por ejemplo, si se analizaron las relaciones entre los requerimientos en un sistema y se identifican nuevos requerimientos, parte o todo el análisis de requerimientos tiene que repetirse. Entonces, es necesario rediseñar el sistema para entregar los nuevos requerimientos, cambiar cualquier programa que se haya desarrollado y volver a probar el sistema.

Existen dos enfoques relacionados que se usan para reducir los costos del rehacer:

1. Evitar el cambio, donde el proceso de software incluye actividades que anticipan cambios posibles antes de requerirse la labor significativa de rehacer. Por ejemplo, puede desarrollarse un sistema prototipo para demostrar a los clientes algunas características clave del sistema. Ellos podrán experimentar con el prototipo y refinar sus requerimientos, antes de comprometerse con mayores costos de producción de software.
2. Tolerancia al cambio, donde el proceso se diseña de modo que los cambios se ajusten con un costo relativamente bajo. Por lo general, esto comprende algunas formas de desarrollo incremental. Los cambios propuestos pueden implementarse en incrementos que aún no se desarrollan. Si no es posible, entonces tal vez sólo un incremento (una pequeña parte del sistema) tendría que alterarse para incorporar el cambio.

En esta sección se estudian dos formas de enfrentar el cambio y los requerimientos cambiantes del sistema. Se trata de lo siguiente:

1. Prototipo de sistema, donde rápidamente se desarrolla una versión del sistema o una parte del mismo, para comprobar los requerimientos del cliente y la factibilidad de algunas decisiones de diseño. Esto apoya el hecho de evitar el cambio, al permitir que los usuarios experimenten con el sistema antes de entregarlo y así refinar sus requerimientos. Como resultado, es probable que se reduzca el número de propuestas de cambio de requerimientos posterior a la entrega.
2. Entrega incremental, donde los incrementos del sistema se entregan al cliente para su comentario y experimentación. Esto apoya tanto al hecho de evitar el cambio como a tolerar el cambio. Por un lado, evita el compromiso prematuro con los requerimientos para todo el sistema y, por otro, permite la incorporación de cambios en incrementos mayores a costos relativamente bajos.

La noción de refactorización, esto es, el mejoramiento de la estructura y organización de un programa, es también un mecanismo importante que apoya la tolerancia al cambio. Este tema se explica en el capítulo 3, que se ocupa de los métodos ágiles.

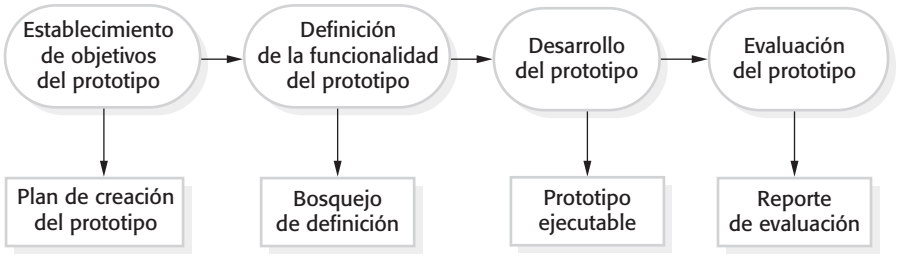


Figura 2.9 Proceso de desarrollo del prototipo

2.3.1 Creación del prototipo

Un prototipo es una versión inicial de un sistema de software que se usa para demostrar conceptos, tratar opciones de diseño y encontrar más sobre el problema y sus posibles soluciones. El rápido desarrollo iterativo del prototipo es esencial, de modo que se controlen los costos, y los interesados en el sistema experimenten por anticipado con el prototipo durante el proceso de software.

Un prototipo de software se usa en un proceso de desarrollo de software para contribuir a anticipar los cambios que se requieran:

1. En el proceso de ingeniería de requerimientos, un prototipo ayuda con la selección y validación de requerimientos del sistema.
2. En el proceso de diseño de sistemas, un prototipo sirve para buscar soluciones específicas de software y apoyar el diseño de interfaces del usuario.

Los prototipos del sistema permiten a los usuarios ver qué tan bien el sistema apoya su trabajo. Pueden obtener nuevas ideas para requerimientos y descubrir áreas de fortalezas y debilidades en el software. Entonces, proponen nuevos requerimientos del sistema. Más aún, conforme se desarrolla el prototipo, quizá se revelen errores y omisiones en los requerimientos propuestos. Una función descrita en una especificación puede parecer útil y bien definida. Sin embargo, cuando dicha función se combina con otras operaciones, los usuarios descubren frecuentemente que su visión inicial era incorrecta o estaba incompleta. Entonces, se modifica la especificación del sistema con la finalidad de reflejar su nueva comprensión de los requerimientos.

Mientras se elabora el sistema para la realización de experimentos de diseño, un prototipo del mismo sirve para comprobar la factibilidad de un diseño propuesto. Por ejemplo, puede crearse un prototipo del diseño de una base de datos y ponerse a prueba, con el objetivo de comprobar que soporta de forma eficiente el acceso de datos para las consultas más comunes del usuario. Asimismo, la creación de prototipos es una parte esencial del proceso de diseño de interfaz del usuario. Debido a la dinámica natural de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientemente buenos para expresar los requerimientos de la interfaz del usuario. Por lo tanto, la creación rápida de prototipos con la participación del usuario final es la única forma sensible para desarrollar interfaces de usuario gráficas para sistemas de software.

En la figura 2.9 se muestra un modelo del proceso para desarrollo de prototipos. Los objetivos de la creación de prototipos deben ser más explícitos desde el inicio del proceso. Esto tendría la finalidad de desarrollar un sistema para un prototipo de la interfaz

del usuario, y diseñar un sistema que valide los requerimientos funcionales del sistema o desarrolle un sistema que demuestre a los administradores la factibilidad de la aplicación. El mismo prototipo no puede cumplir con todos los objetivos, ya que si éstos quedan sin especificar, los administradores o usuarios finales quizá malinterpreten la función del prototipo. En consecuencia, es posible que no obtengan los beneficios esperados del desarrollo del prototipo.

La siguiente etapa del proceso consiste en decidir qué poner y, algo quizá más importante, qué dejar fuera del sistema de prototipo. Para reducir los costos de creación de prototipos y acelerar las fechas de entrega, es posible dejar cierta funcionalidad fuera del prototipo y, también, decidir hacer más flexible los requerimientos no funcionales, como el tiempo de respuesta y la utilización de memoria. El manejo y la gestión de errores pueden ignorarse, a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Además, es posible reducir los estándares de fiabilidad y calidad del programa.

La etapa final del proceso es la evaluación del prototipo. Hay que tomar provisiones durante esta etapa para la capacitación del usuario y usar los objetivos del prototipo para derivar un plan de evaluación. Los usuarios requieren tiempo para sentirse cómodos con un nuevo sistema e integrarse a un patrón normal de uso. Una vez que utilizan el sistema de manera normal, descubren errores y omisiones en los requerimientos.

Un problema general con la creación de prototipos es que quizás el prototipo no se utilice necesariamente en la misma forma que el sistema final. El revisor del prototipo tal vez no sea un usuario típico del sistema. También, podría resultar insuficiente el tiempo de capacitación durante la evaluación del prototipo. Si el prototipo es lento, los evaluadores podrían ajustar su forma de trabajar y evitar aquellas características del sistema con tiempos de respuesta lentos. Cuando se da una mejor respuesta en el sistema final, se puede usar de forma diferente.

En ocasiones, los desarrolladores están presionados por los administradores para entregar prototipos desechables, sobre todo cuando existen demoras en la entrega de la versión final del software. Sin embargo, por lo general esto no es aconsejable:

1. Puede ser imposible corregir el prototipo para cubrir requerimientos no funcionales, como los requerimientos de rendimiento, seguridad, robustez y fiabilidad, ignorados durante el desarrollo del prototipo.
2. El cambio rápido durante el desarrollo significa claramente que el prototipo no está documentado. La única especificación de diseño es el código del prototipo. Esto no es muy bueno para el mantenimiento a largo plazo.
3. Probablemente los cambios realizados durante el desarrollo de prototipos degradarán la estructura del sistema, y este último será difícil y costoso de mantener.
4. Por lo general, durante el desarrollo de prototipos se hacen más flexibles los estándares de calidad de la organización.

Los prototipos no tienen que ser ejecutables para ser útiles. Los modelos en papel de la interfaz de usuario del sistema (Rettig, 1994) pueden ser efectivos para ayudar a los usuarios a refinar un diseño de interfaz y trabajar a través de escenarios de uso. Su desarrollo es muy económico y suelen construirse en pocos días. Una extensión de esta técnica es un prototipo de Mago de Oz, donde sólo se desarrolle la interfaz del usuario. Los usuarios interactúan con esta interfaz, pero sus solicitudes pasan a una persona que los interpreta y les devuelve la respuesta adecuada.

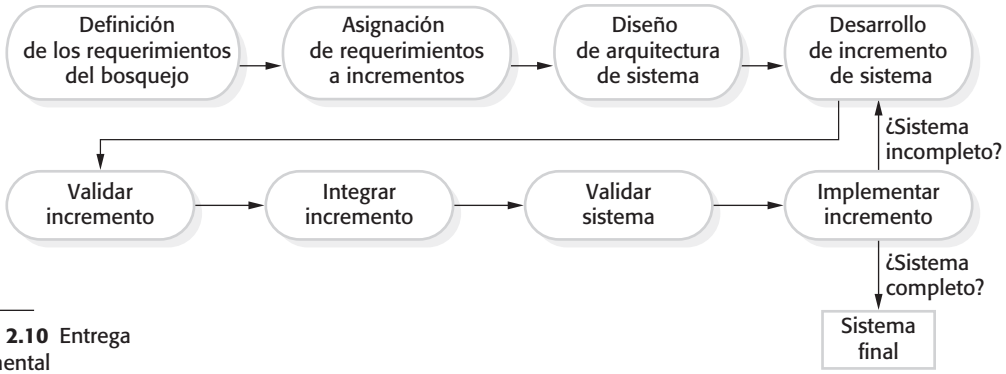


Figura 2.10 Entrega incremental

2.3.2 Entrega incremental

La entrega incremental (figura 2.10) es un enfoque al desarrollo de software donde algunos de los incrementos diseñados se entregan al cliente y se implementan para usarse en un entorno operacional. En un proceso de entrega incremental, los clientes identifican, en un bosquejo, los servicios que proporciona el sistema. Identifican cuáles servicios son más importantes y cuáles son menos significativos para ellos. Entonces, se define un número de incrementos de entrega, y cada incremento proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios por incrementos depende de la prioridad del servicio, donde los servicios de más alta prioridad se implementan y entregan primero.

Una vez identificados los incrementos del sistema, se definen con detalle los requerimientos de los servicios que se van a entregar en el primer incremento, y se desarrolla ese incremento. Durante el desarrollo, puede haber un mayor análisis de requerimientos para incrementos posteriores, aun cuando se rechacen cambios de requerimientos para el incremento actual.

Una vez completado y entregado el incremento, los clientes lo ponen en servicio. Esto significa que toman la entrega anticipada de la funcionalidad parcial del sistema. Pueden experimentar con el sistema que les ayuda a clarificar sus requerimientos, para posteriores incrementos del sistema. A medida que se completan nuevos incrementos, se integran con los incrementos existentes, de modo que con cada incremento entregado mejore la funcionalidad del sistema.

La entrega incremental tiene algunas ventajas:

1. Los clientes pueden usar los primeros incrementos como prototipos y adquirir experiencia que informe sobre sus requerimientos, para posteriores incrementos del sistema. A diferencia de los prototipos, éstos son parte del sistema real, de manera que no hay reaprendizaje cuando está disponible el sistema completo.
2. Los clientes deben esperar hasta la entrega completa del sistema, antes de ganar valor del mismo. El primer incremento cubre sus requerimientos más críticos, de modo que es posible usar inmediatamente el software.
3. El proceso mantiene los beneficios del desarrollo incremental en cuanto a que debe ser relativamente sencillo incorporar cambios al sistema.
4. Puesto que primero se entregan los servicios de mayor prioridad y luego se integran los incrementos, los servicios de sistema más importantes reciben mayores pruebas.

Esto significa que los clientes tienen menos probabilidad de encontrar fallas de software en las partes más significativas del sistema.

Sin embargo, existen problemas con la entrega incremental:

1. La mayoría de los sistemas requieren de una serie de recursos que se utilizan para diferentes partes del sistema. Dado que los requerimientos no están definidos con detalle sino hasta que se implementa un incremento, resulta difícil identificar recursos comunes que necesiten todos los incrementos.
2. Asimismo, el desarrollo iterativo resulta complicado cuando se diseña un sistema de reemplazo. Los usuarios requieren de toda la funcionalidad del sistema antiguo, ya que es común que no deseen experimentar con un nuevo sistema incompleto. Por lo tanto, es difícil conseguir retroalimentación útil del cliente.
3. La esencia de los procesos iterativos es que la especificación se desarrolla en conjunto con el software. Sin embargo, esto se puede contradecir con el modelo de adquisiciones de muchas organizaciones, donde la especificación completa del sistema es parte del contrato de desarrollo del sistema. En el enfoque incremental, no hay especificación completa del sistema, sino hasta que se define el incremento final. Esto requiere una nueva forma de contrato que los grandes clientes, como las agencias gubernamentales, encontrarían difícil de adoptar.

Existen algunos tipos de sistema donde el desarrollo incremental y la entrega no son el mejor enfoque. Hay sistemas muy grandes donde el desarrollo incluye equipos que trabajan en diferentes ubicaciones, algunos sistemas embebidos donde el software depende del desarrollo de hardware y algunos sistemas críticos donde todos los requerimientos tienen que analizarse para comprobar las interacciones que comprometan la seguridad o protección del sistema.

Estos sistemas, desde luego, enfrentan los mismos problemas de incertidumbre y requerimientos cambiantes. En consecuencia, para solucionar tales problemas y obtener algunos de los beneficios del desarrollo incremental, se utiliza un proceso donde un prototipo del sistema se elabore iterativamente y se utilice como plataforma, para experimentar con los requerimientos y el diseño del sistema. Con la experiencia obtenida del prototipo, pueden concertarse los requerimientos definitivos.

2.3.3 Modelo en espiral de Boehm

Boehm (1988) propuso un marco del proceso de software dirigido por el riesgo (el modelo en espiral), que se muestra en la figura 2.11. Aquí, el proceso de software se representa como una espiral, y no como una secuencia de actividades con cierto retroceso de una actividad a otra. Cada ciclo en la espiral representa una fase del proceso de software. Por ende, el ciclo más interno puede relacionarse con la factibilidad del sistema, el siguiente ciclo con la definición de requerimientos, el ciclo que sigue con el diseño del sistema, etcétera. El modelo en espiral combina el evitar el cambio con la tolerancia al cambio. Lo anterior supone que los cambios son resultado de riesgos del proyecto e incluye actividades de gestión de riesgos explícitas para reducir tales riesgos.

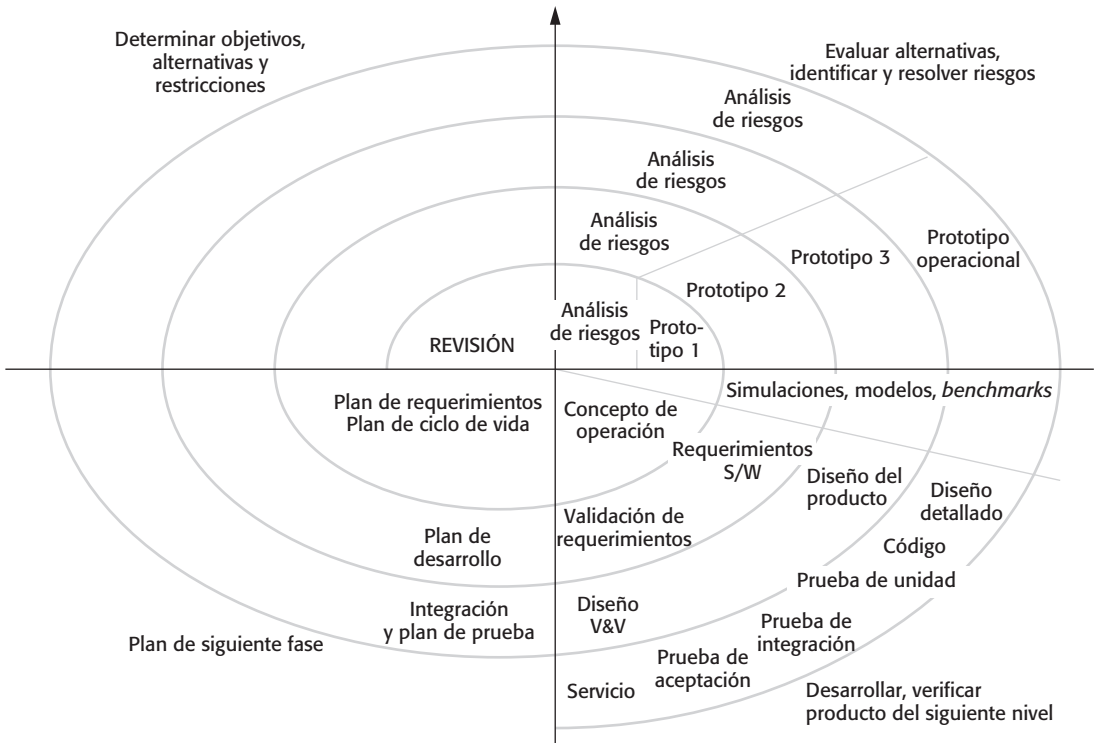


Figura 2.11 Modelo en espiral de Boehm del proceso de software (© IEEE, 1988)

Cada ciclo en la espiral se divide en cuatro sectores:

1. *Establecimiento de objetivos* Se definen objetivos específicos para dicha fase del proyecto. Se identifican restricciones en el proceso y el producto, y se traza un plan de gestión detallado. Se identifican los riesgos del proyecto. Pueden planearse estrategias alternativas, según sean los riesgos.

2. *Valoración y reducción del riesgo* En cada uno de los riesgos identificados del proyecto, se realiza un análisis minucioso. Se dan acciones para reducir el riesgo. Por ejemplo, si existe un riesgo de que los requerimientos sean inadecuados, puede desarrollarse un sistema prototipo.

3. *Desarrollo y validación* Después de una evaluación del riesgo, se elige un modelo de desarrollo para el sistema. Por ejemplo, la creación de prototipos desechables sería el mejor enfoque de desarrollo, si predominan los riesgos en la interfaz del usuario. Si la principal consideración son los riesgos de seguridad, el desarrollo con base en transformaciones formales sería el proceso más adecuado, entre otros. Si el principal riesgo identificado es la integración de subsistemas, el modelo en cascada sería el mejor modelo de desarrollo a utilizar.

4. *Planeación* El proyecto se revisa y se toma una decisión sobre si hay que continuar con otro ciclo de la espiral. Si se opta por continuar, se trazan los planes para la siguiente fase del proyecto.

La diferencia principal entre el modelo en espiral con otros modelos de proceso de software es su reconocimiento explícito del riesgo. Un ciclo de la espiral comienza por elaborar objetivos como rendimiento y funcionalidad. Luego, se numeran formas alternativas de alcanzar dichos objetivos y de lidiar con las restricciones en cada uno de ellos. Cada alternativa se valora contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver dichos riesgos, mediante actividades de recopilación de información, como análisis más detallado, creación de prototipos y simulación.

Una vez valorados los riesgos se realiza cierto desarrollo, seguido por una actividad de planeación para la siguiente fase del proceso. De manera informal, el riesgo significa simplemente algo que podría salir mal. Por ejemplo, si la intención es usar un nuevo lenguaje de programación, un riesgo sería que los compiladores disponibles no sean confiables o no produzcan un código-objeto suficientemente eficaz. Los riesgos conducen a propuestas de cambios de software y a problemas de proyecto como exceso en las fechas y el costo, de manera que la minimización del riesgo es una actividad muy importante de administración del proyecto. En el capítulo 22 se tratará la gestión del riesgo, una parte esencial de la administración del proyecto.

2.4 El Proceso Unificado Racional

El Proceso Unificado Racional (RUP, por las siglas de *Rational Unified Process*) (Krutchen, 2003) es un ejemplo de un modelo de proceso moderno que se derivó del trabajo sobre el UML y el proceso asociado de desarrollo de software unificado (Rumbaugh *et al*., 1999; Arlow y Neustadt, 2005). Aquí se incluye una descripción, pues es un buen ejemplo de un modelo de proceso híbrido. Conjunta elementos de todos los modelos de proceso genéricos (sección 2.1), ilustra la buena práctica en especificación y diseño (sección 2.2), y apoya la creación de prototipos y entrega incremental (sección 2.3).

El RUP reconoce que los modelos de proceso convencionales presentan una sola visión del proceso. En contraste, el RUP por lo general se describe desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo a través del tiempo.
2. Una perspectiva estática que presenta las actividades del proceso que se establecen.
3. Una perspectiva práctica que sugiere buenas prácticas a usar durante el proceso.

La mayoría de las descripciones del RUP buscan combinar las perspectivas estática y dinámica en un solo diagrama (Krutchen, 2003). Esto hace que el proceso resulte más difícil de entender, por lo que en este texto se usan descripciones separadas de cada una de estas perspectivas.

El RUP es un modelo en fases que identifica cuatro fases discretas en el proceso de software. Sin embargo, a diferencia del modelo en cascada, donde las fases se igualan con actividades del proceso, las fases en el RUP están más estrechamente vinculadas con la empresa que con las preocupaciones técnicas. La figura 2.11 muestra las fases del RUP. Éstas son:

1. *Concepción* La meta de la fase de concepción es establecer un caso empresarial para el sistema. Deben identificarse todas las entidades externas (personas y sistemas)

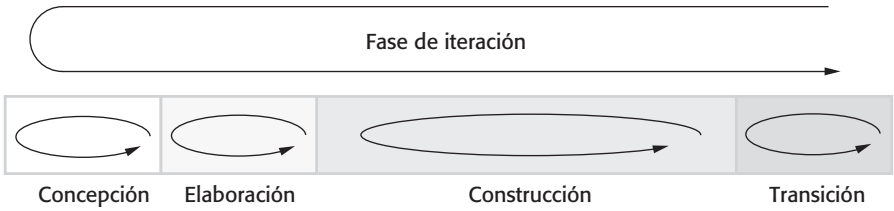


Figura 2.12 Fases en el Proceso Unificado Racional

que interactuarán con el sistema y definirán dichas interacciones. Luego se usa esta información para valorar la aportación del sistema hacia la empresa. Si esta aportación es menor, entonces el proyecto puede cancelarse después de esta fase.

2. *Elaboración* Las metas de la fase de elaboración consisten en desarrollar la comprensión del problema de dominio, establecer un marco conceptual arquitectónico para el sistema, diseñar el plan del proyecto e identificar los riesgos clave del proyecto. Al completar esta fase, debe tenerse un modelo de requerimientos para el sistema, que podría ser una serie de casos de uso del UML, una descripción arquitectónica y un plan de desarrollo para el software.

3. *Construcción* La fase de construcción incluye diseño, programación y pruebas del sistema. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar ésta, debe tenerse un sistema de software funcionando y la documentación relacionada y lista para entregarse al usuario.

4. *Transición* La fase final del RUP se interesa por el cambio del sistema desde la comunidad de desarrollo hacia la comunidad de usuarios, y por ponerlo a funcionar en un ambiente real. Esto es algo ignorado en la mayoría de los modelos de proceso de software aunque, en efecto, es una actividad costosa y en ocasiones problemática. En el complemento de esta fase se debe tener un sistema de software documentado que funcione correctamente en su entorno operacional.

La iteración con el RUP se apoya en dos formas. Cada fase puede presentarse en una forma iterativa, con los resultados desarrollados incrementalmente. Además, todo el conjunto de fases puede expresarse de manera incremental, como se muestra en la flecha en curva desde *transición* hasta *concepción* en la figura 2.12.

La visión estática del RUP se enfoca en las actividades que tienen lugar durante el proceso de desarrollo. Se les llama flujos de trabajo en la descripción RUP. En el proceso se identifican seis flujos de trabajo de proceso centrales y tres flujos de trabajo de apoyo centrales. El RUP se diseñó en conjunto con el UML, de manera que la descripción del flujo de trabajo se orienta sobre modelos UML asociados, como modelos de secuencia, modelos de objeto, etcétera. En la figura 2.13 se describen la ingeniería central y los flujos de trabajo de apoyo.

La ventaja en la presentación de las visiones dinámica y estática radica en que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. En principio, al menos, todos los flujos de trabajo RUP pueden estar activos en la totalidad de las etapas del proceso. En las fases iniciales del proceso, es probable que se use mayor esfuerzo en los flujos de trabajo como modelado del negocio y requerimientos y, en fases posteriores, en las pruebas y el despliegue.

Flujo de trabajo	Descripción
Modelado del negocio	Se modelan los procesos de negocios utilizando casos de uso de la empresa.
Requerimientos	Se identifican los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema.
Análisis y diseño	Se crea y documenta un modelo de diseño utilizando modelos arquitectónicos, de componentes, de objetos y de secuencias.
Implementación	Se implementan y estructuran los componentes del sistema en subsistemas de implementación. La generación automática de código a partir de modelos de diseño ayuda a acelerar este proceso.
Pruebas	Las pruebas son un proceso iterativo que se realiza en conjunto con la implementación. Las pruebas del sistema siguen al completar la implementación.
Despliegue	Se crea la liberación de un producto, se distribuye a los usuarios y se instala en su lugar de trabajo.
Administración de la configuración y del cambio	Este flujo de trabajo de apoyo gestiona los cambios al sistema (véase el capítulo 25).
Administración del proyecto	Este flujo de trabajo de apoyo gestiona el desarrollo del sistema (véase los capítulos 22 y 23).
Entorno	Este flujo de trabajo pone a disposición del equipo de desarrollo de software, las herramientas adecuadas de software.

Figura 2.13 Flujos de trabajo estáticos en el Proceso Unificado Racional

El enfoque práctico del RUP describe las buenas prácticas de ingeniería de software que se recomiendan para su uso en el desarrollo de sistemas. Las seis mejores prácticas fundamentales que se recomiendan son:

1. *Desarrollo de software de manera iterativa* Incrementar el plan del sistema con base en las prioridades del cliente, y desarrollar oportunamente las características del sistema de mayor prioridad en el proceso de desarrollo.
2. *Gestión de requerimientos* Documentar de manera explícita los requerimientos del cliente y seguir la huella de los cambios a dichos requerimientos. Analizar el efecto de los cambios sobre el sistema antes de aceptarlos.
3. *Usar arquitecturas basadas en componentes* Estructurar la arquitectura del sistema en componentes, como se estudió anteriormente en este capítulo.
4. *Software modelado visualmente* Usar modelos UML gráficos para elaborar representaciones de software estáticas y dinámicas.
5. *Verificar la calidad del software* Garantizar que el software cumpla con los estándares de calidad de la organización.

6. *Controlar los cambios al software* Gestionar los cambios al software con un sistema de administración del cambio, así como con procedimientos y herramientas de administración de la configuración.

El RUP no es un proceso adecuado para todos los tipos de desarrollo, por ejemplo, para desarrollo de software embebido. Sin embargo, sí representa un enfoque que potencialmente combina los tres modelos de proceso genéricos que se estudiaron en la sección 2.1. Las innovaciones más importantes en el RUP son la separación de fases y flujos de trabajo, y el reconocimiento de que el despliegue del software en un entorno del usuario forma parte del proceso. Las fases son dinámicas y tienen metas. Los flujos de trabajo son estáticos y son actividades técnicas que no se asocian con una sola fase, sino que pueden usarse a lo largo del desarrollo para lograr las metas de cada fase.

PUNTOS CLAVE

- Los procesos de software son actividades implicadas en la producción de un sistema de software. Los modelos de proceso de software consisten en representaciones abstractas de dichos procesos.
- Los modelos de proceso general describen la organización de los procesos de software. Los ejemplos de estos modelos generales incluyen el modelo en cascada, el desarrollo incremental y el desarrollo orientado a la reutilización.
- La ingeniería de requerimientos es el proceso de desarrollo de una especificación de software. Las especificaciones tienen la intención de comunicar las necesidades de sistema del cliente a los desarrolladores del sistema.
- Los procesos de diseño e implementación tratan de transformar una especificación de requerimientos en un sistema de software ejecutable. Pueden usarse métodos de diseño sistemáticos como parte de esta transformación.
- La validación del software es el proceso de comprobar que el sistema se conforma a su especificación y que satisface las necesidades reales de los usuarios del sistema.
- La evolución del software tiene lugar cuando cambian los sistemas de software existentes para satisfacer nuevos requerimientos. Los cambios son continuos y el software debe evolucionar para seguir siendo útil.
- Los procesos deben incluir actividades para lidiar con el cambio. Esto puede implicar una fase de creación de prototipos que ayude a evitar malas decisiones sobre los requerimientos y el diseño. Los procesos pueden estructurarse para desarrollo y entrega iterativos, de forma que los cambios se realicen sin perturbar al sistema como un todo.
- El Proceso Unificado Racional es un modelo de proceso genérico moderno que está organizado en fases (concepción, elaboración, construcción y transición), pero separa las actividades (requerimientos, análisis y diseño, etcétera) de dichas fases.

LECTURAS SUGERIDAS

Managing Software Quality and Business Risk. Aun cuando éste es principalmente un libro sobre administración de software, incluye un excelente capítulo (capítulo 4) de modelos de proceso. (M. Ould, John Wiley and Sons Ltd, 1999.)

Process Models in Software Engineering. Ofrece una excelente visión de un amplio rango de modelos de proceso de ingeniería de software que se han propuesto. (W. Scacchi, *Encyclopaedia of Software Engineering,* ed. J.J. Marciniak, John Wiley and Sons, 2001.) http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf.

The Rational Unified Process—An Introduction (3rd edition). Éste es el libro más legible que hay disponible sobre RUP hasta ahora. Krutchen describe bien el proceso, pero sería más deseable ver las dificultades prácticas de usar el proceso. (P. Krutchen, Addison-Wesley, 2003.)

EJERCICIOS

2.1. Explicando las razones para su respuesta, y con base en el tipo de sistema a desarrollar, sugiera el modelo de proceso de software genérico más adecuado que se use como fundamento para administrar el desarrollo de los siguientes sistemas:

Un sistema para controlar el antibloqueo de frenos en un automóvil

Un sistema de realidad virtual para apoyar el mantenimiento de software

Un sistema de contabilidad universitario que sustituya a uno existente

Un sistema interactivo de programación de viajes que ayude a los usuarios a planear viajes con el menor impacto ambiental

2.2. Explique por qué el desarrollo incremental es el enfoque más efectivo para diseñar sistemas de software empresariales. ¿Por qué este modelo es menos adecuado para ingeniería de sistemas de tiempo real?

2.3. Considere el modelo de proceso basado en reutilización que se muestra en la figura 2.3. Explique por qué durante el proceso es esencial tener dos actividades separadas de ingeniería de requerimientos.

2.4. Sugiera por qué, en el proceso de ingeniería de requerimientos, es importante hacer una distinción entre desarrollar los requerimientos del usuario y desarrollar los requerimientos del sistema.

2.5. Describa las principales actividades en el proceso de diseño de software y las salidas de dichas actividades. Con un diagrama, muestre las posibles relaciones entre las salidas de dichas actividades.

2.6. Explique por qué el cambio es inevitable en los sistemas complejos, y mencione ejemplos (además de la creación de prototipos y la entrega incremental) de las actividades de proceso de software que ayudan a predecir los cambios y a lograr que el software por desarrollar sea más resistente al cambio.

2.7. Explique por qué los sistemas desarrollados como prototipos por lo general no deben usarse como sistemas de producción.

2.8. Exponga por qué el modelo en espiral de Boehm es un modelo adaptable que puede apoyar las actividades tanto de evitar el cambio como de tolerar el cambio. En la práctica, este modelo no se ha usado ampliamente. Sugiera por qué éste podría ser el caso.

2.9. ¿Cuáles son las ventajas de proporcionar visiones estática y dinámica del proceso de software como en el Proceso Unificado Racional?

2.10. Históricamente, la introducción de la tecnología ha causado profundos cambios en el mercado laboral y, al menos temporalmente, ha reemplazado a personas en los puestos de trabajo. Explique si es probable que la introducción de extensos procesos de automatización tenga las mismas consecuencias para los ingenieros de software. Si no cree que haya consecuencias, explique por qué. Si cree que reducirá las oportunidades laborales, ¿es ético que los ingenieros afectados resistan pasiva o activamente la introducción de esta tecnología?

REFERENCIAS

Arlow, J. y Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Boston: Addison-Wesley.

Boehm, B. y Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, **21** (5), 61–72.

Budgen, D. (2003). *Software Design (2nd Edition)*. Harlow, UK.: Addison-Wesley.

Krutchen, P. (2003). *The Rational Unified Process—An Introduction (3rd Edition)*. Reading, MA: Addison-Wesley.

Massol, V. y Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Rettig, M. (1994). "Practical Programmer: Prototyping for Tiny Fingers". *Comm. ACM*, **37** (4), 21–7.

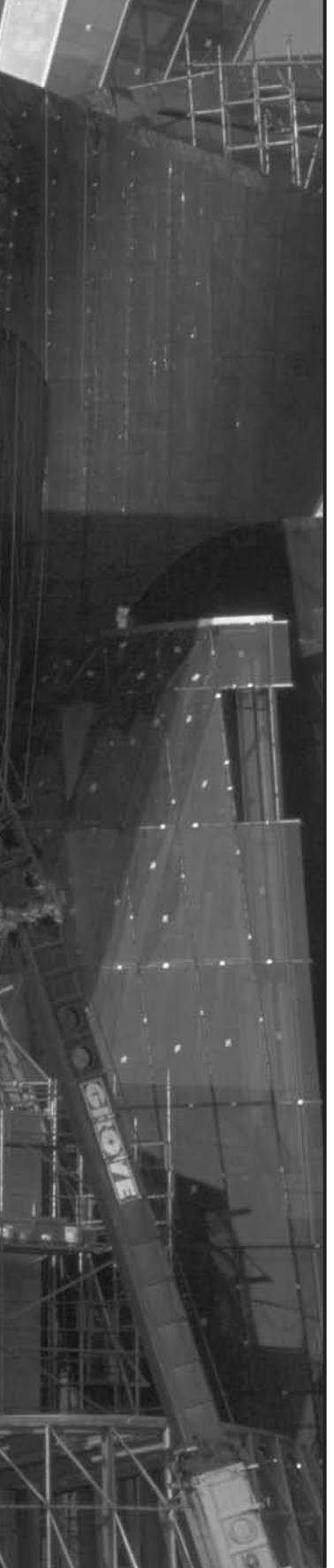
Royce, W. W. (1970). "Managing the Development of Large Software Systems: Concepts and Techniques". IEEE WESTCON, Los Angeles CA: 1–9.

Rumbaugh, J., Jacobson, I. y Booch, G. (1999). *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley.

Schmidt, D. C. (2006). "Model-Driven Engineering". *IEEE Computer*, **39** (2), 25–31.

Schneider, S. (2001). *The B Method*. Houndmills, UK: Palgrave Macmillan.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.



3

Desarrollo ágil de software

Objetivos

El objetivo de este capítulo es introducirlo a los métodos de desarrollo ágil de software. Al estudiar este capítulo:

- comprenderá las razones de los métodos de desarrollo ágil de software, el manifiesto ágil, así como las diferencias entre el desarrollo ágil y el dirigido por un plan;
- conocerá las prácticas clave en la programación extrema y cómo se relacionan con los principios generales de los métodos ágiles;
- entenderá el enfoque de Scrum para la administración de un proyecto ágil;
- reconocerá los conflictos y problemas de escalar los métodos de desarrollo ágil para el diseño de sistemas de software grandes.

Contenido

3.1 Métodos ágiles
3.2 Desarrollo dirigido por un plan y desarrollo ágil
3.3 Programación extrema
3.4 Administración de un proyecto ágil
3.5 Escalamiento de métodos ágiles

Las empresas operan ahora en un entorno global que cambia rápidamente. En ese sentido, deben responder frente a nuevas oportunidades y mercados, al cambio en las condiciones económicas, así como al surgimiento de productos y servicios competitivos. El software es parte de casi todas las operaciones industriales, de modo que el nuevo software se desarrolla rápidamente para aprovechar las actuales oportunidades, con la finalidad de responder ante la amenaza competitiva. En consecuencia, en la actualidad la entrega y el desarrollo rápidos son por lo general el requerimiento fundamental de los sistemas de software. De hecho, muchas empresas están dispuestas a negociar la calidad del software y el compromiso con los requerimientos, para lograr con mayor celeridad la implementación que necesitan del software.

Debido a que dichos negocios funcionan en un entorno cambiante, a menudo es prácticamente imposible derivar un conjunto completo de requerimientos de software estable. Los requerimientos iniciales cambian de modo inevitable, porque los clientes encuentran imposible predecir cómo un sistema afectará sus prácticas operacionales, cómo interactuará con otros sistemas y cuáles operaciones de usuarios se automatizarán. Es posible que sea sólo hasta después de entregar un sistema, y que los usuarios adquieran experiencia con éste, cuando se aclaren los requerimientos reales. Incluso, es probable que debido a factores externos, los requerimientos cambien rápida e impredeciblemente. En tal caso, el software podría ser obsoleto al momento de entregarse.

Los procesos de desarrollo de software que buscan especificar por completo los requerimientos y, luego, diseñar, construir y probar el sistema, no están orientados al desarrollo rápido de software. A medida que los requerimientos cambian, o se descubren problemas en los requerimientos, el diseño o la implementación del sistema tienen que reelaborarse y probarse de nuevo. En consecuencia, un proceso convencional en cascada o uno basado en especificación se prolongan con frecuencia, en tanto que el software final se entrega al cliente mucho después de lo que se especificó originalmente.

En algunos tipos de software, como los sistemas de control críticos para la seguridad, donde es esencial un análisis completo del sistema, resulta oportuno un enfoque basado en un plan. Sin embargo, en un ambiente empresarial de rápido movimiento, esto llega a causar verdaderos problemas. Al momento en que el software esté disponible para su uso, la razón original para su adquisición quizás haya variado tan radicalmente que el software sería inútil a todas luces. Por lo tanto, para sistemas empresariales, son esenciales en particular los procesos de diseño que se enfocan en el desarrollo y la entrega de software rápidos.

Durante algún tiempo, se reconoció la necesidad de desarrollo y de procesos de sistema rápidos que administraran los requerimientos cambiantes. IBM introdujo el desarrollo incremental en la década de 1980 (Mills *et al*., 1980). La entrada de los llamados lenguajes de cuarta generación, también en la misma década, apoyó la idea del software de desarrollo y entrega rápidos (Martin, 1981). Sin embargo, la noción prosperó realmente a finales de la década de 1990, con el desarrollo de la noción de enfoques ágiles como el DSDM (Stapleton, 1997), Scrum (Schwaber y Beedle, 2001) y la programación extrema (Beck, 1999; Beck, 2000).

Los procesos de desarrollo del software rápido se diseñan para producir rápidamente un software útil. El software no se desarrolla como una sola unidad, sino como una serie de incrementos, y cada uno de ellos incluye una nueva funcionalidad del sistema. Aun cuando existen muchos enfoques para el desarrollo de software rápido, comparten algunas características fundamentales:

1. Los procesos de especificación, diseño e implementación están entrelazados. No existe una especificación detallada del sistema, y la documentación del diseño se

minimiza o es generada automáticamente por el entorno de programación que se usa para implementar el sistema. El documento de requerimientos del usuario define sólo las características más importantes del sistema.

2. El sistema se desarrolla en diferentes versiones. Los usuarios finales y otros colaboradores del sistema intervienen en la especificación y evaluación de cada versión. Ellos podrían proponer cambios al software y nuevos requerimientos que se implementen en una versión posterior del sistema.

3. Las interfaces de usuario del sistema se desarrollan usando con frecuencia un sistema de elaboración interactivo, que permita que el diseño de la interfaz se cree rápidamente en cuanto se dibujan y colocan iconos en la interfaz. En tal situación, el sistema puede generar una interfaz basada en la Web para un navegador o una interfaz para una plataforma específica, como Microsoft Windows.

Los métodos ágiles son métodos de desarrollo incremental donde los incrementos son mínimos y, por lo general, se crean las nuevas liberaciones del sistema, y cada dos o tres semanas se ponen a disposición de los clientes. Involucran a los clientes en el proceso de desarrollo para conseguir una rápida retroalimentación sobre los requerimientos cambiantes. Minimizan la cantidad de documentación con el uso de comunicaciones informales, en vez de reuniones formales con documentos escritos.

3.1 Métodos ágiles

En la década de 1980 y a inicios de la siguiente, había una visión muy difundida de que la forma más adecuada para lograr un mejor software era mediante una cuidadosa planeación del proyecto, aseguramiento de calidad formalizada, el uso de métodos de análisis y el diseño apoyado por herramientas CASE, así como procesos de desarrollo de software rigurosos y controlados. Esta percepción proviene de la comunidad de ingeniería de software, responsable del desarrollo de grandes sistemas de software de larga duración, como los sistemas aeroespaciales y gubernamentales.

Este software lo desarrollaron grandes equipos que trabajaban para diferentes compañías. A menudo los equipos estaban geográficamente dispersos y laboraban por largos periodos en el software. Un ejemplo de este tipo de software es el sistema de control de una aeronave moderna, que puede tardar hasta 10 años desde la especificación inicial hasta la implementación. Estos enfoques basados en un plan incluyen costos operativos significativos en la planeación, el diseño y la documentación del sistema. Dichos gastos se justifican cuando debe coordinarse el trabajo de múltiples equipos de desarrollo, cuando el sistema es un sistema crítico y cuando numerosas personas intervendrán en el mantenimiento del software a lo largo de su vida.

Sin embargo, cuando este engorroso enfoque de desarrollo basado en la planeación se aplica a sistemas de negocios pequeños y medianos, los costos que se incluyen son tan grandes que dominan el proceso de desarrollo del software. Se invierte más tiempo en diseñar el sistema, que en el desarrollo y la prueba del programa. Conforme cambian los requerimientos del sistema, resulta esencial la reelaboración y, en principio al menos, la especificación y el diseño deben modificarse con el programa.

En la década de 1990 el descontento con estos enfoques engorrosos de la ingeniería de software condujo a algunos desarrolladores de software a proponer nuevos "métodos

ágiles", los cuales permitieron que el equipo de desarrollo se enfocara en el software en lugar del diseño y la documentación. Los métodos ágiles se apoyan universalmente en el enfoque incremental para la especificación, el desarrollo y la entrega del software. Son más adecuados para el diseño de aplicaciones en que los requerimientos del sistema cambian, por lo general, rápidamente durante el proceso de desarrollo. Tienen la intención de entregar con prontitud el software operativo a los clientes, quienes entonces propondrán requerimientos nuevos y variados para incluir en posteriores iteraciones del sistema. Se dirigen a simplificar el proceso burocrático al evitar trabajo con valor dudoso a largo plazo, y a eliminar documentación que quizá nunca se emplee.

La filosofía detrás de los métodos ágiles se refleja en el manifiesto ágil, que acordaron muchos de los desarrolladores líderes de estos métodos. Este manifiesto afirma:

> *Estamos descubriendo mejores formas para desarrollar software, al hacerlo y al ayudar a otros a hacerlo. Gracias a este trabajo llegamos a valorar:*
>
> *A los individuos y las interacciones sobre los procesos y las herramientas*
>
> *Al software operativo sobre la documentación exhaustiva*
>
> *La colaboración con el cliente sobre la negociación del contrato*
>
> *La respuesta al cambio sobre el seguimiento de un plan*
>
> *Esto es, aunque exista valor en los objetos a la derecha, valoraremos más los de la izquierda.*

Probablemente el método ágil más conocido sea la programación extrema (Beck, 1999; Beck, 2000), descrita más adelante en este capítulo. Otros enfoques ágiles incluyen los de Scrum (Cohn, 2009; Schwaber, 2004; Schwaber y Beedle, 2001), de Crystal (Cockburn, 2001; Cockburn, 2004), de desarrollo de software adaptativo (Highsmith, 2000), de DSDM (Stapleton, 1997; Stapleton, 2003) y el desarrollo dirigido por características (Palmer y Felsing, 2002). El éxito de dichos métodos condujo a cierta integración con métodos más tradicionales de desarrollo, basados en el modelado de sistemas, lo cual resulta en la noción de modelado ágil (Ambler y Jeffries, 2002) y ejemplificaciones ágiles del Proceso Racional Unificado (Larman, 2002).

Aunque todos esos métodos ágiles se basan en la noción del desarrollo y la entrega incrementales, proponen diferentes procesos para lograrlo. Sin embargo, comparten una serie de principios, según el manifiesto ágil y, por ende, tienen mucho en común. Dichos principios se muestran en la figura 3.1. Diferentes métodos ágiles ejemplifican esos principios en diversas formas; sin embargo, no se cuenta con espacio suficiente para discutir todos los métodos ágiles. En cambio, este texto se enfoca en dos de los métodos usados más ampliamente: programación extrema (sección 3.3) y de Scrum (sección 3.4).

Los métodos ágiles han tenido mucho éxito para ciertos tipos de desarrollo de sistemas:

1. Desarrollo del producto, donde una compañía de software elabora un producto pequeño o mediano para su venta.
2. Diseño de sistemas a la medida dentro de una organización, donde hay un claro compromiso del cliente por intervenir en el proceso de desarrollo, y donde no existen muchas reglas ni regulaciones externas que afecten el software.

Principio	Descripción
Participación del cliente	Los clientes deben intervenir estrechamente durante el proceso de desarrollo. Su función consiste en ofrecer y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del mismo.
Entrega incremental	El software se desarrolla en incrementos y el cliente especifica los requerimientos que se van a incluir en cada incremento.
Personas, no procesos	Tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
Adoptar el cambio	Esperar a que cambien los requerimientos del sistema y, de este modo, diseñar el sistema para adaptar dichos cambios.
Mantener simplicidad	Enfocarse en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Siempre que sea posible, trabajar de manera activa para eliminar la complejidad del sistema.

Figura 3.1 Los principios de los métodos ágiles

Como se analiza en la sección final de este capítulo, el éxito de los métodos ágiles se debe al interés considerable por usar dichos métodos para otros tipos de desarrollo del software. No obstante, dado su enfoque en equipos reducidos firmemente integrados, hay problemas en escalarlos hacia grandes sistemas. También se ha experimentado en el uso de enfoques ágiles para la ingeniería de sistemas críticos (Drobna *et al*., 2004). Sin embargo, a causa de las necesidades de seguridad, protección y análisis de confiabilidad en los sistemas críticos, los métodos ágiles requieren modificaciones significativas antes de usarse cotidianamente con la ingeniería de sistemas críticos.

En la práctica, los principios que subyacen a los métodos ágiles son a veces difíciles de cumplir:

1. Aunque es atractiva la idea del involucramiento del cliente en el proceso de desarrollo, su éxito radica en tener un cliente que desee y pueda pasar tiempo con el equipo de desarrollo, y éste represente a todos los participantes del sistema. Los representantes del cliente están comúnmente sujetos a otras presiones, así que no intervienen por completo en el desarrollo del software.
2. Quizás algunos miembros del equipo no cuenten con la personalidad adecuada para la participación intensa característica de los métodos ágiles y, en consecuencia, no podrán interactuar adecuadamente con los otros integrantes del equipo.
3. Priorizar los cambios sería extremadamente difícil, sobre todo en sistemas donde existen muchos participantes. Cada uno por lo general ofrece diversas prioridades a diferentes cambios.
4. Mantener la simplicidad requiere trabajo adicional. Bajo la presión de fechas de entrega, es posible que los miembros del equipo carezcan de tiempo para realizar las simplificaciones deseables al sistema.

5. Muchas organizaciones, especialmente las grandes compañías, pasan años cambiando su cultura, de tal modo que los procesos se definan y continúen. Para ellas, resulta difícil moverse hacia un modelo de trabajo donde los procesos sean informales y estén definidos por equipos de desarrollo.

Otro problema que no es técnico, es decir, que consiste en un problema general con el desarrollo y la entrega incremental, ocurre cuando el cliente del sistema acude a una organización externa para el desarrollo del sistema. Por lo general, el documento de requerimientos del software forma parte del contrato entre el cliente y el proveedor. Como la especificación incremental es inherente en los métodos ágiles, quizá sea difícil elaborar contratos para este tipo de desarrollo.

Como resultado, los métodos ágiles deben apoyarse en contratos, en los cuales el cliente pague por el tiempo requerido para el desarrollo del sistema, en vez de hacerlo por el desarrollo de un conjunto específico de requerimientos. En tanto todo marche bien, esto beneficia tanto al cliente como al desarrollador. No obstante, cuando surgen problemas, sería difícil discutir acerca de quién es culpable y quién debería pagar por el tiempo y los recursos adicionales requeridos para solucionar las dificultades.

La mayoría de los libros y ensayos que describen los métodos ágiles y las experiencias con éstos hablan del uso de dichos métodos para el desarrollo de nuevos sistemas. Sin embargo, como se explica en el capítulo 9, una enorme cantidad de esfuerzo en ingeniería de software se usa en el mantenimiento y la evolución de los sistemas de software existentes. Hay sólo un pequeño número de reportes de experiencia sobre el uso de métodos ágiles para el mantenimiento de software (Poole y Huisman, 2001). Se presentan entonces dos preguntas que deberían considerarse junto con los métodos y el mantenimiento ágiles:

1. ¿Los sistemas que se desarrollan usando un enfoque ágil se mantienen, a pesar del énfasis en el proceso de desarrollo de minimizar la documentación formal?
2. ¿Los métodos ágiles pueden usarse con efectividad para evolucionar un sistema como respuesta a requerimientos de cambio por parte del cliente?

Se estima que la documentación formal describe el sistema y, por lo tanto, facilita la comprensión a quienes cambian el sistema. Sin embargo, en la práctica, con frecuencia la documentación formal no se conserva actualizada y, por ende, no refleja con precisión el código del programa. Por esta razón, los apasionados de los métodos ágiles argumentan que escribir esta documentación es una pérdida de tiempo y que la clave para implementar software mantenible es producir un código legible de alta calidad. De esta manera, las prácticas ágiles enfatizan la importancia de escribir un código bien estructurado y destinar el esfuerzo en mejorar el código. En consecuencia, la falta de documentación no debe representar un problema para mantener los sistemas desarrollados con el uso de un enfoque ágil.

No obstante, según la experiencia del autor con el mantenimiento de sistemas, éste sugiere que el documento clave es el documento de requerimientos del sistema, el cual indica al ingeniero de software lo que se supone que debe hacer el sistema. Sin tal conocimiento, es difícil valorar el efecto de los cambios propuestos al sistema. Varios métodos ágiles recopilan los requerimientos de manera informal e incremental, aunque sin crear un documento coherente de requerimientos. A este respecto, es probable

que el uso de métodos ágiles haga más difícil y costoso el mantenimiento posterior del sistema.

Es factible que las prácticas ágiles, usadas en el proceso de mantenimiento en sí, resulten efectivas, ya sea que se utilice o no se utilice un enfoque ágil para el desarrollo del sistema. La entrega incremental, el diseño para el cambio y el mantenimiento de la simplicidad tienen sentido cuando se modifica el software. De hecho, se pensaría tanto en un proceso de desarrollo ágil como en un proceso de evolución del software.

Sin embargo, quizá la principal dificultad luego de entregar el software sea mantener al cliente interviniendo en el proceso. Aunque un cliente justifique la participación de tiempo completo de un representante durante el desarrollo del sistema, esto es menos probable en el mantenimiento, cuando los cambios no son continuos. Es posible que los representantes del cliente pierdan interés en el sistema. En consecuencia, es previsible que se requieran mecanismos alternativos, como las propuestas de cambio, descritas en el capítulo 25, para establecer los nuevos requerimientos del sistema.

El otro problema potencial tiene que ver con mantener la continuidad del equipo de desarrollo. Los métodos ágiles se apoyan en aquellos miembros del equipo que comprenden los aspectos del sistema sin que deban consultar la documentación. Si se separa un equipo de desarrollo ágil, entonces se pierde este conocimiento implícito y es difícil que los nuevos miembros del equipo acumulen la misma percepción del sistema y sus componentes.

Quienes apoyan los métodos ágiles han creído fielmente en la promoción de su uso y tienden a pasar por alto sus limitaciones. Esto alienta una respuesta igualmente extrema que, para el autor, exagera los problemas con este enfoque (Stephens y Rosenberg, 2003). Críticos más razonables como DeMarco y Boehm (DeMarco y Boehm, 2002) destacan tanto las ventajas como las desventajas de los métodos ágiles. Proponen un enfoque híbrido donde los métodos ágiles que incorporan algunas técnicas del desarrollo dirigido por un plan son la mejor forma de avanzar.

3.2 Desarrollo dirigido por un plan y desarrollo ágil

Los enfoques ágiles en el desarrollo de software consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades en el diseño y la implementación, como la adquisición de requerimientos y pruebas. En contraste, un enfoque basado en un plan para la ingeniería de software identifica etapas separadas en el proceso de software con salidas asociadas a cada etapa. Las salidas de una etapa se usan como base para planear la siguiente actividad del proceso. La figura 3.2 muestra las distinciones entre los enfoques ágil y el basado en un plan para la especificación de sistemas.

En un enfoque basado en un plan, la iteración ocurre dentro de las actividades con documentos formales usados para comunicarse entre etapas del proceso. Por ejemplo, los requerimientos evolucionarán y, a final de cuentas, se producirá una especificación de aquéllos. Esto entonces es una entrada al proceso de diseño y la implementación. En un enfoque ágil, la iteración ocurre a través de las actividades. Por lo tanto, los requerimientos y el diseño se desarrollan en conjunto, no por separado.

Un proceso de software dirigido por un plan soporta el desarrollo y la entrega incrementales. Es perfectamente factible asignar requerimientos y planear tanto la fase de diseño y desarrollo como una serie de incrementos. Un proceso ágil no está inevitable-

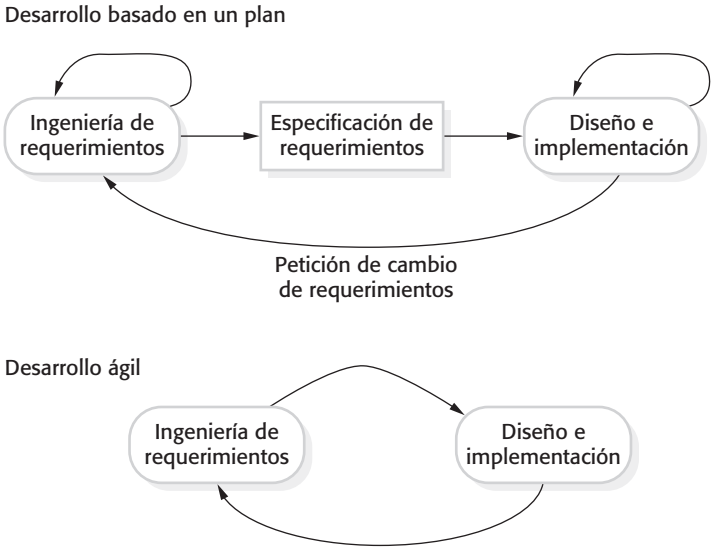


Figura 3.2 Especificación ágil y dirigida por un plan

mente enfocado al código y puede producir cierta documentación de diseño. Como se expone en la siguiente sección, el equipo de desarrollo ágil puede incluir un "pico" de documentación donde, en vez de producir una nueva versión de un sistema, el equipo generará documentación del sistema.

De hecho, la mayoría de los proyectos de software incluyen prácticas de los enfoques ágil y basado en un plan. Para decidir sobre el equilibrio entre un enfoque basado en un plan y uno ágil, se deben responder algunas preguntas técnicas, humanas y organizacionales:

1. ¿Es importante tener una especificación y un diseño muy detallados antes de dirigirse a la implementación? Siendo así, probablemente usted tenga que usar un enfoque basado en un plan.
2. ¿Es práctica una estrategia de entrega incremental, donde se dé el software a los clientes y se obtenga así una rápida retroalimentación de ellos? De ser el caso, considere el uso de métodos ágiles.
3. ¿Qué tan grande es el sistema que se desarrollará? Los métodos ágiles son más efectivos cuando el sistema logra diseñarse con un pequeño equipo asignado que se comunique de manera informal. Esto sería imposible para los grandes sistemas que precisan equipos de desarrollo más amplios, de manera que tal vez se utilice un enfoque basado en un plan.
4. ¿Qué tipo de sistema se desarrollará? Los sistemas que demandan mucho análisis antes de la implementación (por ejemplo, sistema en tiempo real con requerimientos de temporización compleja), por lo general, necesitan un diseño bastante detallado para realizar este análisis. En tales circunstancias, quizá sea mejor un enfoque basado en un plan.
5. ¿Cuál es el tiempo de vida que se espera del sistema? Los sistemas con lapsos de vida prolongados podrían requerir más documentación de diseño, para comunicar al equipo de apoyo los propósitos originales de los desarrolladores del sistema. Sin embargo,

los defensores de los métodos ágiles argumentan acertadamente que con frecuencia la documentación no se conserva actualizada, ni se usa mucho para el mantenimiento del sistema a largo plazo.

6. ¿Qué tecnologías se hallan disponibles para apoyar el desarrollo del sistema? Los métodos ágiles se auxilian a menudo de buenas herramientas para seguir la pista de un diseño en evolución. Si se desarrolla un sistema con un IDE sin contar con buenas herramientas para visualización y análisis de programas, entonces posiblemente se requiera más documentación de diseño.
7. ¿Cómo está organizado el equipo de desarrollo? Si el equipo de desarrollo está distribuido, o si parte del desarrollo se subcontrata, entonces tal vez se requiera elaborar documentos de diseño para comunicarse a través de los equipos de desarrollo. Quizá se necesite planear por adelantado cuáles son.
8. ¿Existen problemas culturales que afecten el desarrollo del sistema? Las organizaciones de ingeniería tradicionales presentan una cultura de desarrollo basada en un plan, pues es una norma en ingeniería. Esto requiere comúnmente una amplia documentación de diseño, en vez del conocimiento informal que se utiliza en los procesos ágiles.
9. ¿Qué tan buenos son los diseñadores y programadores en el equipo de desarrollo? Se argumenta en ocasiones que los métodos ágiles requieren niveles de habilidad superiores a los enfoques basados en un plan, en que los programadores simplemente traducen un diseño detallado en un código. Si usted tiene un equipo con niveles de habilidad relativamente bajos, es probable que necesite del mejor personal para desarrollar el diseño, siendo otros los responsables de la programación.
10. ¿El sistema está sujeto a regulación externa? Si un regulador externo tiene que aprobar el sistema (por ejemplo, la Agencia de Aviación Federal [FAA] estadounidense aprueba el software que es crítico para la operación de una aeronave), entonces, tal vez se le requerirá documentación detallada como parte del sistema de seguridad.

En realidad, es irrelevante el conflicto sobre si un proyecto puede considerarse dirigido por un plan o ágil. A final de cuentas, la principal inquietud de los compradores de un sistema de software es si cuentan o no con un sistema de software ejecutable, que cubra sus necesidades y realice funciones útiles para el usuario de manera individual o dentro de una organización. En la práctica, muchas compañías que afirman haber usado métodos ágiles adoptaron algunas habilidades ágiles y las integraron con sus procesos dirigidos por un plan.

3.3 Programación extrema

La programación extrema (XP) es quizás el método ágil mejor conocido y más ampliamente usado. El nombre lo acuñó Beck (2000) debido a que el enfoque se desarrolló llevando a niveles “extremos” las prácticas reconocidas, como el desarrollo iterativo. Por ejemplo, en la XP muchas versiones actuales de un sistema pueden desarrollarse mediante diferentes programadores, integrarse y ponerse a prueba en un solo día.

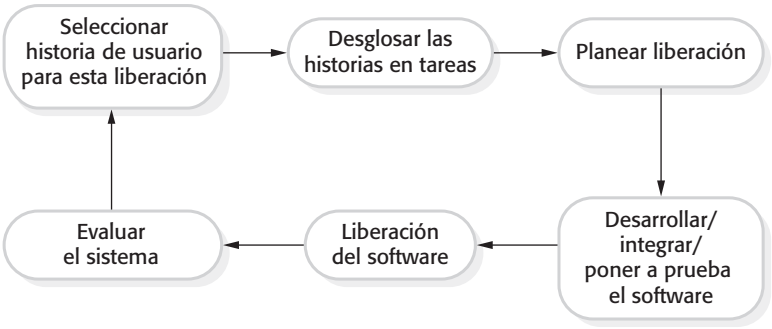


Figura 3.3 El ciclo de liberación de la programación extrema

En la programación extrema, los requerimientos se expresan como escenarios (llamados historias de usuario), que se implementan directamente como una serie de tareas. Los programadores trabajan en pares y antes de escribir el código desarrollan pruebas para cada tarea. Todas las pruebas deben ejecutarse con éxito una vez que el nuevo código se integre en el sistema. Entre las liberaciones del sistema existe un breve lapso. La figura 3.3 ilustra el proceso XP para producir un incremento del sistema por desarrollar.

La programación extrema incluye algunas prácticas, resumidas en la figura 3.4, las cuales reflejan los principios de los métodos ágiles:

1. El desarrollo incremental se apoya en pequeñas y frecuentes liberaciones del sistema. Los requerimientos se fundamentan en simples historias del cliente, o bien, en escenarios usados como base para decidir qué funcionalidad debe incluirse en un incremento del sistema.
2. La inclusión del cliente se apoya a través de un enlace continuo con el cliente en el equipo de desarrollo. El representante del cliente participa en el desarrollo y es responsable de definir las pruebas de aceptación para el sistema.
3. Las personas, no los procesos, se basan en la programación en pares, en la propiedad colectiva del código del sistema y en un proceso de desarrollo sustentable que no incluya jornadas de trabajo excesivamente largas.
4. El cambio se acepta mediante liberaciones regulares del sistema a los clientes, desarrollo de primera prueba, refactorización para evitar degeneración del código e integración continua de nueva funcionalidad.
5. Mantener la simplicidad se logra mediante la refactorización constante, que mejora la calidad del código, y con el uso de diseños simples que no anticipan innecesariamente futuros cambios al sistema.

En un proceso XP, los clientes intervienen estrechamente en la especificación y priorización de los requerimientos del sistema. Estos últimos no se especifican como listas de actividades requeridas del sistema. En cambio, el cliente del sistema forma parte del equipo de desarrollo y discute los escenarios con otros miembros del equipo. En conjunto, desarrollan una "tarjeta de historia" que encapsula las necesidades del cliente. Entonces, el equipo de desarrollo implementa dicho escenario en una liberación futura del software. En la figura 3.5 se muestra el ejemplo de una tarjeta de historia para el

Principio o práctica	Descripción
Planeación incremental	Los requerimientos se registran en tarjetas de historia (*story cards*) y las historias que se van a incluir en una liberación se determinan por el tiempo disponible y la prioridad relativa. Los desarrolladores desglosan dichas historias en "tareas" de desarrollo. Vea las figuras 3.5 y 3.6.
Liberaciones pequeñas	Al principio se desarrolla el conjunto mínimo de funcionalidad útil, que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.
Diseño simple	Se realiza un diseño suficiente para cubrir sólo aquellos requerimientos actuales.
Desarrollo de la primera prueba	Se usa un marco de referencia de prueba de unidad automatizada al escribir las pruebas para una nueva pieza de funcionalidad, antes de que esta última se implemente.
Refactorización	Se espera que todos los desarrolladores refactoricen de manera continua el código y, tan pronto como sea posible, se encuentren mejoras de éste. Lo anterior conserva el código simple y mantenible.
Programación en pares	Los desarrolladores trabajan en pares, y cada uno comprueba el trabajo del otro; además, ofrecen apoyo para que se realice siempre un buen trabajo.
Propiedad colectiva	Los desarrolladores en pares laboran en todas las áreas del sistema, de manera que no se desarrollan islas de experiencia, ya que todos los desarrolladores se responsabilizan por todo el código. Cualquiera puede cambiar cualquier función.
Integración continua	Tan pronto como esté completa una tarea, se integra en todo el sistema. Después de tal integración, deben aprobarse todas las pruebas de unidad en el sistema.
Ritmo sustentable	Grandes cantidades de tiempo extra no se consideran aceptables, pues el efecto neto de este tiempo libre con frecuencia es reducir la calidad del código y la productividad de término medio.
Cliente en sitio	Un representante del usuario final del sistema (el cliente) tiene que disponer de tiempo completo para formar parte del equipo XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y responsable de llevar los requerimientos del sistema al grupo para su implementación.

Figura 3.4 Prácticas de programación extrema

sistema de administración de pacientes en atención a la salud mental. Ésta es una breve descripción de un escenario para prescribir medicamentos a un paciente.

Las tarjetas de historia son las entradas principales al proceso de planeación XP o el "juego de planeación". Una vez diseñadas las tarjetas de historia, el equipo de desarrollo las descompone en tareas (figura 3.6) y estima el esfuerzo y los recursos requeridos para implementar cada tarea. Esto involucra por lo general discusiones con el cliente para refinar los requerimientos. Entonces, para su implementación, el cliente prioriza las historias y elige aquellas que pueden usarse inmediatamente para entregar apoyo empresarial útil. La intención es identificar funcionalidad útil que pueda implementarse en aproximadamente dos semanas, cuando la siguiente liberación del sistema esté disponible para el cliente.

Desde luego, conforme cambian los requerimientos, las historias no implementadas cambian o se desechan. Si se demandan cambios para un sistema que ya se entregó, se desarrollan nuevas tarjetas de historia y, otra vez, el cliente decide si dichos cambios tienen prioridad sobre la nueva función.

Prescripción de medicamentos

Kate es una médica que quiere prescribir fármacos a un paciente que se atiende en una clínica. El archivo del paciente ya se desplegó en su computadora, de manera que da clic en el campo del medicamento y luego puede seleccionar "medicamento actual", "medicamento nuevo" o "formulario".

Si selecciona "medicamento actual", el sistema le pide comprobar la dosis. Si quiere cambiar la dosis, ingresa la dosis y luego confirma la prescripción.

Si elige "medicamento nuevo", el sistema supone que Kate sabe cuál medicamento prescribir. Ella teclea las primeras letras del nombre del medicamento. El sistema muestra una lista de medicamentos posibles cuyo nombre inicia con dichas letras. Posteriormente elige el fármaco requerido y el sistema responde solicitándole que verifique que el medicamento seleccionado sea el correcto. Ella ingresa la dosis y luego confirma la prescripción.

Si Kate elige "formulario", el sistema muestra un recuadro de búsqueda para el formulario aprobado. Entonces busca el medicamento requerido. Ella selecciona un medicamento y el sistema le pide comprobar que éste sea el correcto. Luego ingresa la dosis y confirma la prescripción.

El sistema siempre verifica que la dosis esté dentro del rango aprobado. Si no es así, le pide a Kate que la modifique.

Después de que ella confirma la prescripción, se desplegará para su verificación. Kate hace clic o en "OK" o en "Cambiar". Si hace clic en "OK", la prescripción se registra en la base de datos de auditoría. Si hace clic en "Cambiar", reingresa al proceso de "prescripción de medicamento".

Figura 3.5 Una historia de la "prescripción de medicamento"

A veces, durante la planeación del juego, salen a la luz preguntas que no pueden responderse fácilmente y se requiere trabajo adicional para explorar posibles soluciones. El equipo puede elaborar algún prototipo o tratar de desarrollarlo para entender el problema y la solución. En términos XP, éste es un "pico" (*spike*), es decir, un incremento donde no se realiza programación. También suele haber "picos" para diseñar la arquitectura del sistema o desarrollar la documentación del sistema.

La programación extrema toma un enfoque "extremo" para el desarrollo incremental. Nuevas versiones del software se construyen varias veces al día y las versiones se entregan a los clientes aproximadamente cada dos semanas. Nunca se descuidan las fechas límite de las liberaciones; si hay problemas de desarrollo, se consulta al cliente y la funcionalidad se elimina de la liberación planeada.

Cuando un programador diseña el sistema para crear una nueva versión, debe correr todas las pruebas automatizadas existentes, así como las pruebas para la nueva funcionalidad. La nueva construcción del software se acepta siempre que todas las pruebas se ejecuten con éxito. Entonces esto se convierte en la base para la siguiente iteración del sistema.

Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad. Sin embargo, la programación extrema descartó este principio basada en el hecho de que al diseñar para el cambio con frecuencia se desperdicia esfuerzo. No vale la pena gastar tiempo en adicionar generalidad a un programa para enfrentar el cambio. Los cambios anticipados casi nunca se materializan y en realidad pueden hacerse peticiones de cambio diametralmente opuestas. Por lo tanto, el enfoque XP acepta que los cambios sucederán y cuando éstos ocurran realmente se reorganizará el software.

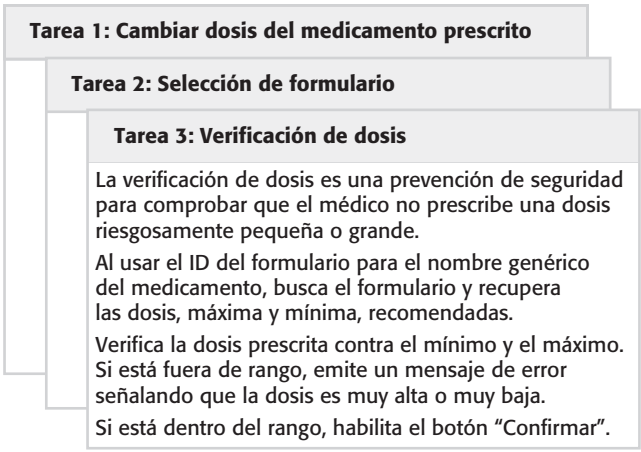


Figura 3.6 Ejemplos de tarjetas de tarea para prescripción de medicamentos.

Un problema general con el desarrollo incremental es que tiende a degradar la estructura del software, de modo que los cambios al software se vuelven cada vez más difíciles de implementar. En esencia, el desarrollo avanza al encontrar soluciones alternativas a los problemas, con el resultado de que el código se duplica con frecuencia, partes del software se reutilizan de forma inadecuada y la estructura global se degrada conforme el código se agrega al sistema.

La programación extrema aborda este problema al sugerir que el software debe refactorizarse continuamente. Esto significa que el equipo de programación busca posibles mejoras al software y las implementa de inmediato. Cuando un miembro del equipo observa un código que puede optimarse, realiza dichas mejoras, aun en situaciones donde no hay necesidad apremiante de ellas. Los ejemplos de refactorización incluyen la reorganización de una jerarquía de clases para remover un código duplicado, el ordenamiento y el cambio de nombre de atributos y métodos, y la sustitución de código con llamadas a métodos definidos en la librería de un programa. Los entornos de desarrollo del programa, como Eclipse (Carlson, 2005), incluyen herramientas para refactorizar, lo cual simplifica el proceso de encontrar dependencias entre secciones de código y realizar modificaciones globales al código.

Entonces, en principio, el software siempre debe ser de fácil comprensión y cambiar a medida que se implementen nuevas historias. En la práctica, no siempre es el caso. En ocasiones la presión del desarrollo significa que la refactorización se demora, porque se dedica el tiempo a la implementación de una nueva funcionalidad. Algunas características y cambios nuevos no pueden ajustarse con facilidad al refactorizar el nivel del código y al requerir modificar la arquitectura del sistema.

En la práctica, muchas compañías que adoptaron XP no usan todas las prácticas de programación extrema que se mencionan en la figura 3.4. Seleccionan según sus formas específicas de trabajar. Por ejemplo, algunas compañías encuentran útil la programación en pares; otras prefieren usar la programación y las revisiones individuales. Para acomodar diferentes niveles de habilidad, algunos programadores no hacen refactorización en partes del sistema que ellos no desarrollan, y pueden usarse requerimientos convencionales en vez de historias de usuario. Sin embargo, la mayoría de las compañías que adoptan una variante XP usan liberaciones pequeñas, desarrollo de primera prueba e integración continua.

3.3.1 Pruebas en XP

Como se indicó en la introducción de este capítulo, una de las diferencias importantes entre desarrollo incremental y desarrollo dirigido por un plan está en la forma en que el sistema se pone a prueba. Con el desarrollo incremental, no hay especificación de sistema que pueda usar un equipo de prueba externo para desarrollar pruebas del sistema. En consecuencia, algunos enfoques del desarrollo incremental tienen un proceso de pruebas muy informal, comparado con las pruebas dirigidas por un plan.

Para evitar varios de los problemas de prueba y validación del sistema, XP enfatiza la importancia de la prueba de programa. La XP incluye un enfoque para probar que reduce las posibilidades de introducir errores no detectados en la versión actual del sistema.

Las características clave de poner a prueba en XP son:

1. Desarrollo de primera prueba,
2. desarrollo de pruebas incrementales a partir de escenarios,
3. involucramiento del usuario en el desarrollo y la validación de pruebas, y
4. el uso de marcos de pruebas automatizadas.

El desarrollo de la primera prueba es una de las innovaciones más importantes en XP. En lugar de escribir algún código y luego las pruebas para dicho código, las pruebas se elaboran antes de escribir el código. Esto significa que la prueba puede correrse conforme se escribe el código y descubrir problemas durante el desarrollo.

Escribir pruebas implícitamente define tanto una interfaz como una especificación del comportamiento para la funcionalidad a desarrollar. Se reducen los problemas de la mala interpretación de los requerimientos y la interfaz. Este enfoque puede adoptarse en cualquier proceso donde haya una relación clara entre un requerimiento de sistema y el código que implementa dicho requerimiento. En la XP, siempre se observa este vínculo porque las tarjetas de historia que representan los requerimientos se descomponen en tareas, y éstas son la principal unidad de implementación. La adopción del desarrollo de primera prueba en XP condujo a enfoques de desarrollo basados en pruebas más generales (Astels, 2003). Éstas se estudian en el capítulo 8.

En el desarrollo de primera prueba, los implementadores de tarea deben comprender ampliamente la especificación, de modo que sean capaces de escribir pruebas para el sistema. Esto significa que las ambigüedades y omisiones en la especificación deben clarificarse antes de comenzar la implementación. Más aún, también evita el problema del "retraso en la prueba". Esto puede ocurrir cuando el desarrollador del sistema trabaja a un ritmo más rápido que el examinador. La implementación está cada vez más adelante de las pruebas y hay una tendencia a omitirlas, de modo que se mantenga la fecha de desarrollo.

Los requerimientos de usuario en XP se expresan como escenarios o historias, y el usuario los prioriza para su desarrollo. El equipo de desarrollo valora cada escenario y lo descompone en tareas. Por ejemplo, en la figura 3.6 se muestran algunas de las tarjetas de tarea desarrolladas a partir de la tarjeta de historia para la prescripción de medicamentos (figura 3.5). Cada tarea genera una o más pruebas de unidad, que verifican la implementación descrita en dicha tarea. La figura 3.7 es una descripción breve de un caso de prueba que se desarrolló para comprobar que la dosis prescrita de un medicamento no se halle fuera de los límites de seguridad conocidos.

Prueba 4: Comprobación de dosis
Entrada: 1. Un número en mg que represente una sola dosis del medicamento. 2. Un número que signifique el número de dosis individuales por día. **Pruebas:** 1. Probar las entradas donde la dosis individual sea correcta, pero la frecuencia muy elevada. 2. Probar las entradas donde la dosis individual sea muy alta y muy baja. 3. Probar las entradas donde la dosis individual × frecuencia sea muy alta y muy baja. 4. Probar las entradas donde la dosis individual × frecuencia esté en el rango permitido. **Salida:** OK o mensaje de error que indique que la dosis está fuera del rango de seguridad.

Figura 3.7
Descripción de caso de prueba para comprobar dosis

El papel del cliente en el proceso de pruebas es ayudar a desarrollar pruebas de aceptación para las historias, que deban implementarse en la siguiente liberación del sistema. Como se estudiará en el capítulo 8, las pruebas de aceptación son el proceso donde el sistema se pone a prueba usando datos del cliente para verificar que se cubren las necesidades reales de éste.

En XP, la prueba de aceptación, como el desarrollo, es incremental. El cliente que forma parte del equipo escribe pruebas conforme avanza el desarrollo. Por lo tanto, todo código nuevo se valida para garantizar que eso sea lo que necesita el cliente. Para la historia en la figura 3.5, la prueba de aceptación implicaría escenarios donde *a*) se cambió la dosis de un medicamento, *b*) se seleccionó un nuevo medicamento y *c*) se usó el formulario para encontrar un medicamento. En la práctica, se requiere por lo general una serie de pruebas de aceptación en vez de una sola prueba.

Contar con el cliente para apoyar el desarrollo de pruebas de aceptación en ocasiones es una gran dificultad en el proceso de pruebas XP. Quienes adoptan el rol del cliente tienen disponibilidad muy limitada, por lo que es probable que no trabajen a tiempo completo con el equipo de desarrollo. El cliente podría creer que brindar los requerimientos fue suficiente contribución y, por lo tanto, se mostrarían renuentes a intervenir en el proceso de pruebas.

La automatización de las pruebas es esencial para el desarrollo de la primera prueba. Las pruebas se escriben como componentes ejecutables antes de implementar la tarea. Dichos componentes de pruebas deben ser independientes, simular el envío de la entrada a probar y verificar que el resultado cumple con la especificación de salida. Un marco de pruebas automatizadas es un sistema que facilita la escritura de pruebas realizables y envía una serie de pruebas para su ejecución. Junit (Massol y Husted, 2003) es un ejemplo usado ampliamente de un marco de pruebas automatizadas.

Conforme se automatizan las pruebas, siempre hay una serie de pruebas que se ejecutan rápida y fácilmente. Cada vez que se agregue cualquier funcionalidad al sistema, pueden correrse las pruebas y conocerse de inmediato los problemas que introduce el nuevo código.

El desarrollo de la primera prueba y las pruebas automatizadas por lo general dan por resultado un gran número de pruebas que se escriben y ejecutan. Sin embargo, este enfoque no conduce necesariamente a pruebas minuciosas del programa. Existen tres razones para ello:

1. Los programadores prefieren programar que probar y, en ocasiones, toman atajos cuando escriben pruebas. Por ejemplo, escriben pruebas incompletas que no comprueban todas las posibles excepciones que quizás ocurran.

2. Algunas pruebas llegan a ser muy difíciles de escribir de manera incremental. Por ejemplo, en una interfaz de usuario compleja, suele ser complicado escribir pruebas de unidad para el código que implementa la "lógica de despliegue" y el flujo de trabajo entre pantallas.

3. Es difícil juzgar la totalidad de un conjunto de pruebas. Aunque tenga muchas pruebas de sistema, su conjunto de pruebas no ofrece cobertura completa. Partes críticas del sistema pueden no ejecutarse y, por ende, permanecerían sin probarse.

En consecuencia, aunque un gran conjunto de pruebas ejecutadas regularmente da la impresión de que el sistema está completo y es correcto, esto tal vez no sea el caso. Si las pruebas no se revisan y se escriben más pruebas después del desarrollo, entonces pueden entregarse *bugs* (problemas, errores en el programa) en la liberación del sistema.

3.3.2 Programación en pares

Otra práctica innovadora que se introdujo en XP es que los programadores trabajan en pares para desarrollar el software. En realidad, trabajan juntos en la misma estación de trabajo para desarrollar el software. Sin embargo, los mismos pares no siempre programan juntos. En vez de ello, los pares se crean dinámicamente, de manera que todos los miembros del equipo trabajen entre sí durante el proceso de desarrollo.

El uso de la programación en pares tiene algunas ventajas:

1. Apoya la idea de la propiedad y responsabilidad colectivas para el sistema. Esto refleja la idea de Weinberg (1971) sobre la programación sin ego, donde el software es propiedad del equipo como un todo y los individuos no son responsables por los problemas con el código. En cambio, el equipo tiene responsabilidad colectiva para resolver dichos problemas.

2. Actúa como un proceso de revisión informal, porque al menos dos personas observan cada línea de código. Las inspecciones y revisiones de código (que se explican en el capítulo 24) son muy eficientes para detectar un alto porcentaje de errores de software. Sin embargo, consumen tiempo en su organización y, usualmente, presentan demoras en el proceso de desarrollo. Aunque la programación en pares es un proceso menos formal que quizá no identifica tantos errores como las inspecciones de código, es un proceso de inspección mucho más económico que las inspecciones formales del programa.

3. Ayuda a la refactorización, que es un proceso de mejoramiento del software. La dificultad de implementarlo en un entorno de desarrollo normal es que el esfuerzo en la refactorización se utiliza para beneficio a largo plazo. Un individuo que practica la refactorización podría calificarse como menos eficiente que uno que simplemente realiza desarrollo del código. Donde se usan la programación en pares y la propiedad colectiva, otros se benefician inmediatamente de la refactorización, de modo que es probable que apoyen el proceso.

Al respecto, tal vez se pensaría que la programación en pares es menos eficiente que la programación individual. En un tiempo dado, un par de desarrolladores elaboraría

la mitad del código que dos individuos que trabajen solos. Hay varios estudios de la productividad de los programadores en pares con resultados mixtos. Al usar estudiantes voluntarios, Williams y sus colaboradores (Cockburn y Williams, 2001; Williams *et al*., 2000) descubrieron que la productividad con la programación en pares es comparable con la de dos individuos que trabajan de manera independiente. Las razones sugeridas son que los pares discuten el software antes de desarrollarlo, de modo que probablemente tengan menos salidas en falso y menos rediseño. Más aún, el número de errores que se evitan por la inspección informal es tal que se emplea menos tiempo en reparar los *bugs* descubiertos durante el proceso de pruebas.

Sin embargo, los estudios con programadores más experimentados (Arisholm *et al*., 2007; Parrish *et al*., 2004) no replican dichos resultados. Hallaron que había una pérdida de productividad significativa comparada con dos programadores que trabajan individualmente. Hubo algunos beneficios de calidad, pero no compensaron por completo los costos de la programación en pares. No obstante, el intercambio de conocimiento que ocurre durante la programación en pares es muy importante, pues reduce los riesgos globales de un proyecto cuando salen miembros del equipo. En sí mismo, esto hace que la programación de este tipo valga la pena.

3.4 Administración de un proyecto ágil

La responsabilidad principal de los administradores del proyecto de software es dirigir el proyecto, de modo que el software se entregue a tiempo y con el presupuesto planeado para ello. Supervisan el trabajo de los ingenieros de software y monitorizan el avance en el desarrollo del software.

El enfoque estándar de la administración de proyectos es el basado en un plan. Como se estudia en el capítulo 23, los administradores se apoyan en un plan para el proyecto que muestra lo que se debe entregar y cuándo, así como quién trabajará en el desarrollo de los entregables del proyecto. Un enfoque basado en un plan requiere en realidad que un administrador tenga una visión equilibrada de todo lo que debe diseñarse y de los procesos de desarrollo. Sin embargo, no funciona bien con los métodos ágiles, donde los requerimientos se desarrollan incrementalmente, donde el software se entrega en rápidos incrementos cortos, y donde los cambios a los requerimientos y el software son la norma.

Como cualquier otro proceso de diseño de software profesional, el desarrollo ágil tiene que administrarse de tal modo que se busque el mejor uso del tiempo y de los recursos disponibles para el equipo. Esto requiere un enfoque diferente a la administración del proyecto, que se adapte al desarrollo incremental y a las fortalezas particulares de los métodos ágiles.

Aunque el enfoque de Scrum (Schwaber, 2004; Schwaber y Beedle, 2001) es un método ágil general, su enfoque está en la administración iterativa del desarrollo, y no en enfoques técnicos específicos para la ingeniería de software ágil. La figura 3.8 representa un diagrama del proceso de administración de Scrum. Este proceso no prescribe el uso de prácticas de programación, como la programación en pares y el desarrollo de primera prueba. Por lo tanto, puede usarse con enfoques ágiles más técnicos, como XP, para ofrecer al proyecto un marco administrativo.

Existen tres fases con Scrum. La primera es la planeación del bosquejo, donde se establecen los objetivos generales del proyecto y el diseño de la arquitectura de software.

Figura 3.8 El proceso de Scrum

A esto le sigue una serie de ciclos *sprint*, donde cada ciclo desarrolla un incremento del sistema. Finalmente, la fase de cierre del proyecto concluye el proyecto, completa la documentación requerida, como los marcos de ayuda del sistema y los manuales del usuario, y valora las lecciones aprendidas en el proyecto.

La característica innovadora de Scrum es su fase central, a saber, los ciclos *sprint*. Un *sprint* de Scrum es una unidad de planeación en la que se valora el trabajo que se va a realizar, se seleccionan las particularidades por desarrollar y se implementa el software. Al final de un *sprint*, la funcionalidad completa se entrega a los participantes. Las características clave de este proceso son las siguientes:

1. Los *sprints* tienen longitud fija, por lo general de dos a cuatro semanas. Corresponden al desarrollo de una liberación del sistema en XP.
2. El punto de partida para la planeación es la cartera del producto, que es la lista de trabajo por realizar en el proyecto. Durante la fase de valoración del *sprint*, esto se revisa, y se asignan prioridades y riesgos. El cliente interviene estrechamente en este proceso y al comienzo de cada *sprint* puede introducir nuevos requerimientos o tareas.
3. La fase de selección incluye a todo el equipo del proyecto que trabaja con el cliente, con la finalidad de seleccionar las características y la funcionalidad a desarrollar durante el *sprint*.
4. Una vez acordado, el equipo se organiza para desarrollar el software. Con el objetivo de revisar el progreso y, si es necesario, volver a asignar prioridades al trabajo, se realizan reuniones diarias breves con todos los miembros del equipo. Durante esta etapa, el equipo se aísla del cliente y la organización, y todas las comunicaciones se canalizan a través del llamado "maestro de Scrum". El papel de este último es proteger al equipo de desarrollo de distracciones externas. La forma en que el trabajo se realiza depende del problema y del equipo. A diferencia de XP, Scrum no hace sugerencias específicas sobre cómo escribir requerimientos, desarrollar la primera prueba, etcétera. Sin embargo, dichas prácticas XP se usan cuando el equipo las considera adecuadas.
5. Al final del *sprint*, el trabajo hecho se revisa y se presenta a los participantes. Luego comienza el siguiente ciclo de *sprint*.

La idea detrás de Scrum es que debe autorizarse a todo el equipo para tomar decisiones, de modo que se evita deliberadamente el término "administrador del proyecto". En

lugar de ello, el "maestro de Scrum" es el facilitador que ordena las reuniones diarias, rastrea el atraso del trabajo a realizar, registra las decisiones, mide el progreso del atraso, y se comunica con los clientes y administradores fuera del equipo.

Todo el equipo asiste a las reuniones diarias, que en ocasiones son reuniones en las que los participantes no se sientan, para hacerlas breves y enfocadas. Durante la reunión, todos los miembros del equipo comparten información, describen sus avances desde la última reunión, los problemas que han surgido y los planes del día siguiente. Ello significa que todos en el equipo conocen lo que acontece y, si surgen problemas, replantean el trabajo en el corto plazo para enfrentarlo. Todos participan en esta planeación; no hay dirección descendente desde el maestro de Scrum.

En la Web existen muchos reportes anecdóticos del uso exitoso del Scrum. Rising y Janoff (2000) discuten su uso exitoso en un entorno de desarrollo de software para telecomunicaciones y mencionan sus ventajas del modo siguiente:

1. El producto se desglosa en un conjunto de piezas manejables y comprensibles.
2. Los requerimientos inestables no retrasan el progreso.
3. Todo el equipo tiene conocimiento de todo y, en consecuencia, se mejora la comunicación entre el equipo.
4. Los clientes observan la entrega a tiempo de los incrementos y obtienen retroalimentación sobre cómo funciona el producto.
5. Se establece la confianza entre clientes y desarrolladores, a la vez que se crea una cultura positiva donde todos esperan el triunfo del proyecto.

Scrum, como originalmente se designó, tenía la intención de usarse con equipos coasignados, donde todos los miembros del equipo pudieran congregarse a diario en reuniones breves. Sin embargo, mucho del desarrollo del software implica ahora equipos distribuidos con miembros del equipo ubicados en diferentes lugares alrededor del mundo. En consecuencia, hay varios experimentos en marcha con la finalidad de desarrollar el Scrum para entornos de desarrollo distribuidos (Smits y Pshigoda, 2007; Sutherland *et al*., 2007).

3.5 Escalamiento de métodos ágiles

Los métodos ágiles se desarrollaron para usarse en pequeños equipos de programación, que podían trabajar juntos en la misma habitación y comunicarse de manera informal. Por lo tanto, los métodos ágiles se emplean principalmente para el diseño de sistemas pequeños y medianos. Desde luego, la necesidad de entrega más rápida del software, que es más adecuada para las necesidades del cliente, se aplica también a sistemas más grandes. Por consiguiente, hay un enorme interés en escalar los métodos ágiles para enfrentar los sistemas de mayor dimensión, desarrollados por grandes organizaciones.

Denning y sus colaboradores (2008) argumentan que la única forma de evitar los problemas comunes de la ingeniería de software, como los sistemas que no cubren las necesidades del cliente y exceden el presupuesto, es encontrar maneras de hacer que los métodos ágiles funcionen para grandes sistemas. Leffingwell (2007) discute cuáles prácticas ágiles se escalan al desarrollo de grandes sistemas. Moore y Spens (2008) reportan su experiencia al usar un enfoque ágil para desarrollar un gran sistema médico, con 300 desarrolladores que trabajaban en equipos distribuidos geográficamente.

El desarrollo de grandes sistemas de software difiere en algunas formas del desarrollo de sistemas pequeños:

1. Los grandes sistemas son, por lo general, colecciones de sistemas separados en comunicación, donde equipos separados desarrollan cada sistema. Dichos equipos trabajan con frecuencia en diferentes lugares, en ocasiones en otras zonas horarias. Es prácticamente imposible que cada equipo tenga una visión de todo el sistema. En consecuencia, sus prioridades son generalmente completar la parte del sistema sin considerar asuntos de los sistemas más amplios.
2. Los grandes sistemas son "sistemas abandonados" (Hopkins y Jenkins, 2008); esto es, incluyen e interactúan con algunos sistemas existentes. Muchos de los requerimientos del sistema se interesan por su interacción y, por lo tanto, en realidad no se prestan a la flexibilidad y al desarrollo incremental. Aquí también podrían ser relevantes los conflictos políticos y a menudo la solución más sencilla a un problema es cambiar un sistema existente. Sin embargo, esto requiere negociar con los administradores de dicho sistema para convencerlos de que los cambios pueden implementarse sin riesgo para la operación del sistema.
3. Donde muchos sistemas se integran para crear un solo sistema, una fracción significativa del desarrollo se ocupa en la configuración del sistema, y no en el desarrollo del código original. Esto no necesariamente es compatible con el desarrollo incremental y la integración frecuente del sistema.
4. Los grandes sistemas y sus procesos de desarrollo por lo común están restringidos por reglas y regulaciones externas, que limitan la forma en que pueden desarrollarse, lo cual requiere de ciertos tipos de documentación del sistema que se va a producir, etcétera.
5. Los grandes sistemas tienen un tiempo prolongado de adquisición y desarrollo. Es difícil mantener equipos coherentes que conozcan el sistema durante dicho periodo, pues resulta inevitable que algunas personas se cambien a otros empleos y proyectos.
6. Los grandes sistemas tienen por lo general un conjunto variado de participantes. Por ejemplo, cuando enfermeras y administradores son los usuarios finales de un sistema médico, el personal médico ejecutivo, los administradores del hospital, etcétera, también son participantes en el sistema. En realidad es imposible involucrar a todos estos participantes en el proceso de desarrollo.

Existen dos perspectivas en el escalamiento de los métodos ágiles:

1. Una perspectiva de "expansión" (*scaling up*), que se interesa por el uso de dichos métodos para el desarrollo de grandes sistemas de software que no logran desarrollarse con equipos pequeños.

2. Una perspectiva de "ampliación" (*scaling out*), que se interesa por que los métodos ágiles se introduzcan en una organización grande con muchos años de experiencia en el desarrollo de software.

Los métodos ágiles tienen que adaptarse para enfrentar la ingeniería de los sistemas grandes. Leffingwell (2007) explica que es esencial mantener los fundamentos de los métodos ágiles: planeación flexible, liberación frecuente del sistema, integración continua, desarrollo dirigido por pruebas y buena comunicación del equipo. El autor considera que las siguientes adaptaciones son críticas y deben introducirse:

1. Para el desarrollo de grandes sistemas no es posible enfocarse sólo en el código del sistema. Es necesario hacer más diseño frontal y documentación del sistema. Debe diseñarse la arquitectura de software y producirse documentación para describir los aspectos críticos del sistema, como esquemas de bases de datos, división del trabajo entre los equipos, etcétera.
2. Tienen que diseñarse y usarse mecanismos de comunicación entre equipos. Esto debe incluir llamadas telefónicas regulares, videoconferencias entre los miembros del equipo y frecuentes reuniones electrónicas breves, para que los equipos se actualicen mutuamente del avance. Hay que ofrecer varios canales de comunicación (como correo electrónico, mensajería instantánea, wikis y sistemas de redes sociales) para facilitar las comunicaciones.
3. La integración continua, donde todo el sistema se construya cada vez que un desarrollador verifica un cambio, es prácticamente imposible cuando muchos programas separados deben integrarse para crear el sistema. Sin embargo, resulta esencial mantener construcciones del sistema frecuentes y liberaciones del sistema regulares. Esto podría significar la introducción de nuevas herramientas de gestión de configuración que soporten el desarrollo de software por parte de múltiples equipos.

Las compañías de software pequeñas que desarrollan productos de software están entre quienes adoptan con más entusiasmo los métodos ágiles. Dichas compañías no están restringidas por burocracias organizacionales o estándares de procesos, y son capaces de cambiar rápidamente para acoger nuevas ideas. Desde luego, las compañías más grandes también experimentan en proyectos específicos con los métodos ágiles; sin embargo, para ellas es mucho más difícil "ampliar" dichos métodos en toda la organización. Lindvall y sus colaboradores (2004) analizan algunos de los problemas al escalar los métodos ágiles en cuatro grandes compañías tecnológicas.

Es difícil introducir los métodos ágiles en las grandes compañías por algunas razones:

1. Los gerentes del proyecto carecen de experiencia con los métodos ágiles; pueden ser reticentes para aceptar el riesgo de un nuevo enfoque, pues no saben cómo afectará sus proyectos particulares.
2. Las grandes organizaciones tienen a menudo procedimientos y estándares de calidad que se espera sigan todos los proyectos y, dada su naturaleza burocrática, es probable que sean incompatibles con los métodos ágiles. En ocasiones, reciben apoyo de herramientas de software (por ejemplo, herramientas de gestión de requerimientos), y el uso de dichas herramientas es obligatorio para todos los proyectos.

3. Los métodos ágiles parecen funcionar mejor cuando los miembros del equipo tienen un nivel de habilidad relativamente elevado. Sin embargo, dentro de grandes organizaciones, probablemente haya una amplia gama de habilidades y destrezas, y los individuos con niveles de habilidad inferiores quizá no sean miembros de equipos efectivos en los procesos ágiles.
4. Quizás haya resistencia cultural contra los métodos ágiles, en especial en aquellas organizaciones con una larga historia de uso de procesos convencionales de ingeniería de sistemas.

Los procedimientos de gestión de cambio y de pruebas son ejemplos de procedimientos de la compañía que podrían no ser compatibles con los métodos ágiles. La administración del cambio es el proceso que controla los cambios a un sistema, de modo que el efecto de los cambios sea predecible y se controlen los costos. Antes de realizarse, todos los cambios deben aprobarse y esto entra en conflicto con la noción de refactorización. En XP, cualquier desarrollador puede mejorar cualquier código sin conseguir aprobación externa. Para sistemas grandes, también existen estándares de pruebas, donde una construcción del sistema se envía a un equipo de pruebas externo. Esto entraría en conflicto con los enfoques de primera prueba y prueba frecuente utilizados en XP.

Introducir y sostener el uso de los métodos ágiles a lo largo de una organización grande es un proceso de cambio cultural. El cambio cultural tarda mucho tiempo en implementarse y a menudo requiere un cambio de administración antes de llevarse a cabo. Las compañías que deseen usar métodos ágiles necesitan promotores para alentar el cambio. Tienen que dedicar recursos significativos para el proceso del cambio. Al momento de escribir este texto, unas cuantas compañías clasificadas como grandes han realizado una transición exitosa al desarrollo ágil a lo largo de la organización.

PUNTOS CLAVE

- Los métodos ágiles son métodos de desarrollo incremental que se enfocan en el diseño rápido, liberaciones frecuentes del software, reducción de gastos en el proceso y producción de código de alta calidad. Hacen que el cliente intervenga directamente en el proceso de desarrollo.
- La decisión acerca de si se usa un enfoque de desarrollo ágil o uno basado en un plan depende del tipo de software que se va a elaborar, las capacidades del equipo de desarrollo y la cultura de la compañía que diseña el sistema.
- La programación extrema es un método ágil bien conocido que integra un rango de buenas prácticas de programación, como las liberaciones frecuentes del software, el mejoramiento continuo del software y la participación del cliente en el equipo de desarrollo.
- Una fortaleza particular de la programación extrema, antes de crear una característica del programa, es el desarrollo de pruebas automatizadas. Todas las pruebas deben ejecutarse con éxito cuando un incremento se integra en un sistema.

- El método de Scrum es un método ágil que ofrece un marco de referencia para la administración del proyecto. Se centra alrededor de un conjunto de *sprints*, que son periodos fijos cuando se desarrolla un incremento de sistema. La planeación se basa en priorizar un atraso de trabajo y seleccionar las tareas de importancia más alta para un *sprint*.
- Resulta difícil el escalamiento de los métodos ágiles para sistemas grandes, ya que éstos necesitan diseño frontal y cierta documentación. La integración continua es prácticamente imposible cuando existen muchos equipos de desarrollo separados que trabajan en un proyecto.

LECTURAS SUGERIDAS

Extreme Programming Explained. Éste fue el primer libro sobre XP y todavía es, quizá, el más legible. Explica el enfoque desde la perspectiva de uno de sus inventores y el entusiasmo se evidencia claramente en el libro. (Kent Beck, Addison-Wesley, 2000.)

"Get Ready for Agile Methods, With Care". Una crítica detallada de los métodos ágiles, que examina sus fortalezas y debilidades; está escrito por un ingeniero de software con vasta experiencia. (B. Boehm, *IEEE Computer*, enero de 2002.) http://doi.ieeecomputersociety.org/10.1109/2.976920.

Scaling Software Agility: Best Practices for Large Enterprises. Aunque se enfoca en los conflictos del escalamiento de los métodos ágiles, este libro también incluye un resumen de los principales métodos ágiles, como XP, Scrum y Crystal. (D. Leffingwell, Addison-Wesley, 2007.)

Running an Agile Software Development Project. La mayoría de los libros acerca de los métodos ágiles se enfocan en un método específico, pero este texto toma un enfoque diferente y analiza cómo poner en práctica XP en un proyecto. Buen consejo práctico. (M. Holcombe, John Wiley and Sons, 2008.)

EJERCICIOS

3.1. Explique por qué la entrega e implementación rápidas de nuevos sistemas es con frecuencia más importante para las empresas que la funcionalidad detallada de dichos sistemas.

3.2. Señale cómo los principios subyacentes a los métodos ágiles conducen al acelerado desarrollo e implementación del software.

3.3. ¿Cuándo desaconsejaría el uso de un método ágil para desarrollar un sistema de software?

3.4. La programación extrema expresa los requerimientos del usuario como historias, y cada historia se escribe en una tarjeta. Analice las ventajas y desventajas de este enfoque para la descripción de requerimientos.

3.5. Explique por qué el desarrollo de la primera prueba ayuda al programador a diseñar una mejor comprensión de los requerimientos del sistema. ¿Cuáles son las dificultades potenciales con el desarrollo de la primera prueba?

3.6. Sugiera cuatro razones por las que la tasa de productividad de los programadores que trabajan en pares llega a ser más de la mitad que la de dos programadores que trabajan individualmente.

3.7. Compare y contraste el enfoque de Scrum para la administración de proyectos con enfoques convencionales basados en un plan, estudiados en el capítulo 23. Las comparaciones deben basarse en la efectividad de cada enfoque para planear la asignación de personal a los proyectos, estimar el costo de los mismos, mantener la cohesión del equipo y administrar los cambios en la conformación del equipo del proyecto.

3.8. Usted es el administrador de software en una compañía que desarrolla software de control crítico para una aeronave. Es el responsable de la elaboración de un sistema de apoyo al diseño de software, que ayude a la traducción de los requerimientos de software a una especificación formal del software (que se estudia en el capítulo 13). Comente acerca de las ventajas y las desventajas de las siguientes estrategias de desarrollo:

a) Recopile los requerimientos para tal sistema con los ingenieros de software y los participantes externos (como la autoridad de certificación reguladora), y desarrolle el sistema usando un enfoque basado en un plan.

b) Diseñe un prototipo usando un lenguaje de script, como Ruby o Python, evalúe este prototipo con los ingenieros de software y otros participantes; luego, revise los requerimientos del sistema. Vuelva a desarrollar el sistema final con Java.

c) Desarrolle el sistema en Java usando un enfoque ágil, con un usuario involucrado en el equipo de diseño.

3.9. Se ha sugerido que uno de los problemas de tener un usuario estrechamente involucrado con un equipo de desarrollo de software es que “se vuelve nativo”; esto es, adopta el punto de vista del equipo de desarrollo y pierde la visión de las necesidades de sus colegas usuarios. Sugiera tres formas en que se podría evitar este problema y discuta las ventajas y desventajas de cada enfoque.

3.10. Con la finalidad de reducir costos y el impacto ambiental del cambio, su compañía decide cerrar algunas oficinas y ofrecer apoyo al personal para trabajar desde casa. Sin embargo, el gerente que introdujo la política no está consciente de que el software se desarrolla usando métodos ágiles, que se apoya en el trabajo cercano del equipo y de la programación en pares. Analice las dificultades que causaría esta nueva política y cómo podría solventar estos problemas.

REFERENCIAS

Ambler, S. W. y Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.

Arisholm, E., Gallis, H., Dyba, T. y Sjoberg, D. I. K. (2007). "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise". *IEEE Trans. on Software Eng.,* **33** (2), 65–86.

Astels, D. (2003). *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall.

Beck, K. (1999). "Embracing Change with Extreme Programming". *IEEE Computer,* **32** (10), 70–8.

Beck, K. (2000). *Extreme Programming explained*. Reading, Mass.: Addison-Wesley.

Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.

Cockburn, A. (2001). *Agile Software Development*. Reading, Mass.: Addison-Wesley.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley.

Cockburn, A. y Williams, L. (2001). "The costs and benefits of pair programming". *In Extreme programming examined. (ed.)*. Boston: Addison-Wesley.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley.

Demarco, T. y Boehm, B. (2002). "The Agile Methods Fray". *IEEE Computer,* **35** (6), 90–2.

Denning, P. J., Gunderson, C. y Hayes-Roth, R. (2008). "Evolutionary System Development". *Comm. ACM,* **51** (12), 29–31.

Drobna, J., Noftz, D. y Raghu, R. (2004). "Piloting XP on Four Mission-Critical Projects". *IEEE Software,* **21** (6), 70–5.

Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.

Hopkins, R. y Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield. Boston,* Mass.: IBM Press.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall.

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. y Kahkonen, T. (2004). "Agile Software Development in Large Organizations". *IEEE Computer,* **37** (12), 26–34.

Martin, J. (1981). *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall.

Massol, V. y Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. y Quinnan, R. E. (1980). "The Management of Software Engineering". *IBM Systems*. J., **19** (4), 414–77.

Moore, E. y Spens, J. (2008). "Scaling Agile: Finding your Agile Tribe". *Proc. Agile 2008 Conference,* Toronto: IEEE Computer Society. 121–124.

Palmer, S. R. y Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall.

Parrish, A., Smith, R., Hale, D. y Hale, J. (2004). "A Field Study of Developer Pairs: Productivity Impacts and Implications". *IEEE Software,* **21** (5), 76–9.

Poole, C. y Huisman, J. W. (2001). "Using Extreme Programming in a Maintenance Environment". *IEEE Software,* **18** (6), 42–50.

Rising, L. y Janoff, N. S. (2000). "The Scrum Software Development Process for Small Teams". *IEEE Software,* **17** (4), 26–32.

Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.

Schwaber, K. y Beedle, M. (2001). *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall.

Smits, H. y Pshigoda, G. (2007). "Implementing Scrum in a Distributed Software Development Organization". Agile 2007, Washington, DC: IEEE Computer Society.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method*. Harlow, UK: Addison-Wesley.

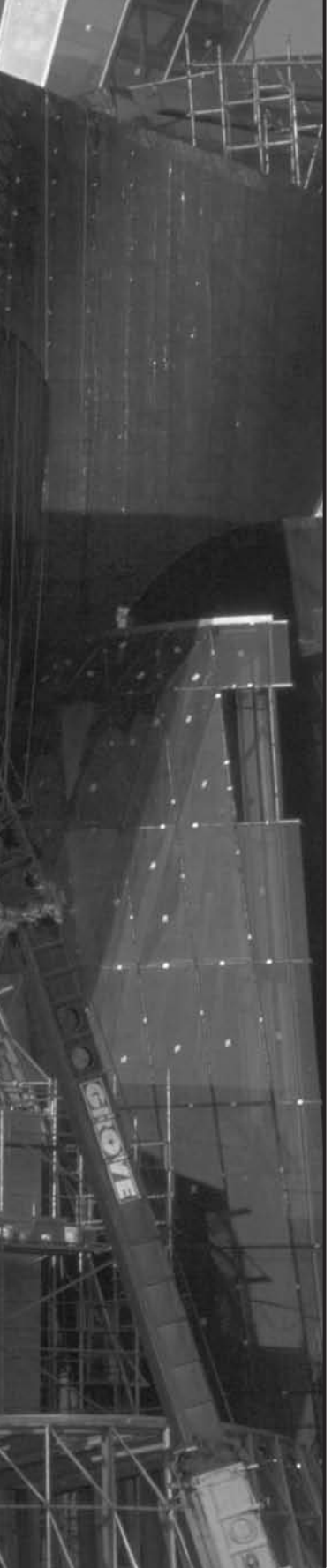
Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed*. Harlow, UK: Pearson Education.

Stephens, M. y Rosenberg, D. (2003). *Extreme Programming Refactored*. Berkley, Calif.: Apress.

Sutherland, J., Viktorov, A., Blount, J. y Puntikov, N. (2007). "Distributed Scrum: Agile Project Management with Outsourced Development Teams". 40th Hawaii Int. Conf. on System Sciences, Hawaii: IEEE Computer Society.

Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand.

Williams, L., Kessler, R. R., Cunningham, W. y Jeffries, R. (2000). "Strengthening the Case for Pair Programming". *IEEE Software,* **17** (4), 19–25.



4

Ingeniería de requerimientos

Objetivos

El objetivo de este capítulo es introducir los requerimientos de software y discutir los procesos que hay en el descubrimiento y la documentación de tales requerimientos. Al estudiar este capítulo:

- entenderá los conceptos de requerimientos del usuario y del sistema, así como por qué tales requerimientos se deben escribir en diferentes formas;
- comprenderá las diferencias entre requerimientos de software funcionales y no funcionales;
- reconocerá cómo se organizan los requerimientos dentro de un documento de requerimientos de software;
- conocerá las principales actividades de la ingeniería de requerimientos: adquisición, análisis y validación, así como las relaciones entre dichas actividades;
- analizará por qué es necesaria la administración de requerimientos y cómo ésta apoya otras actividades de la ingeniería de requerimientos.

Contenido

4.1 Requerimientos funcionales y no funcionales
4.2 El documento de requerimientos de software
4.3 Especificación de requerimientos
4.4 Procesos de ingeniería de requerimientos
4.5 Adquisición y análisis de requerimientos
4.6 Validación de requerimientos
4.7 Administración de requerimientos

Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito, como sería controlar un dispositivo, colocar un pedido o buscar información. Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requerimientos (IR).

El término "requerimiento" no se usa de manera continua en la industria del software. En algunos casos, un requerimiento es simplemente un enunciado abstracto de alto nivel en un servicio que debe proporcionar un sistema, o bien, una restricción sobre un sistema. En el otro extremo, consiste en una definición detallada y formal de una función del sistema. Davis (1993) explica por qué existen esas diferencias:

> *Si una compañía desea otorgar un contrato para un gran proyecto de desarrollo de software, tiene que definir sus necesidades de una forma suficientemente abstracta para que una solución no esté predefinida. Los requerimientos deben redactarse de tal forma que muchos proveedores liciten en pos del contrato, ofreciendo, tal vez, diferentes maneras de cubrir las necesidades de organización del cliente. Una vez otorgado el contrato, el proveedor tiene que escribir con más detalle una definición del sistema para el cliente, de modo que éste comprenda y valide lo que hará el software. Estos documentos suelen nombrarse documentos de requerimientos para el sistema.*

Algunos de los problemas que surgen durante el proceso de ingeniería de requerimientos son resultado del fracaso de hacer una separación clara entre esos diferentes niveles de descripción. En este texto se distinguen con el uso del término "requerimientos del usuario" para representar los requerimientos abstractos de alto nivel; y "requerimientos del sistema" para caracterizar la descripción detallada de lo que el sistema debe hacer. Los requerimientos del usuario y los requerimientos del sistema se definen del siguiente modo:

1. Los requerimientos del usuario son enunciados, en un lenguaje natural junto con diagramas, acerca de qué servicios esperan los usuarios del sistema, y de las restricciones con las cuales éste debe operar.
2. Los requerimientos del sistema son descripciones más detalladas de las funciones, los servicios y las restricciones operacionales del sistema de software. El documento de requerimientos del sistema (llamado en ocasiones especificación funcional) tiene que definir con exactitud lo que se implementará. Puede formar parte del contrato entre el comprador del sistema y los desarrolladores del software.

Los diferentes niveles de requerimientos son útiles debido a que informan sobre el sistema a distintos tipos de lector. La figura 4.1 ilustra la diferencia entre los requerimientos del usuario y del sistema. Este ejemplo de un sistema de administración de pacientes para apoyar la atención a la salud mental (MHC-PMS) muestra cómo los requerimientos del usuario se extienden hacia varios requerimientos del sistema. En la figura 4.1 se observa que el requerimiento del usuario es muy general. Los requerimientos del sistema ofrecen información más específica sobre los servicios y las funciones del sistema que se implementará.

Definición del requerimiento del usuario

1. El MHC-PMS elaborará mensualmente informes administrativos que revelen el costo de los medicamentos prescritos por cada clínica durante ese mes.

Especificación de los requerimientos del sistema

1.1 En el último día laboral de cada mes se redactará un resumen de los medicamentos prescritos, su costo y las clínicas que los prescriben.
1.2 El sistema elaborará automáticamente el informe que se imprimirá después de las 17:30 del último día laboral del mes.
1.3 Se realizará un reporte para cada clínica junto con los nombres de cada medicamento, el número de prescripciones, las dosis prescritas y el costo total de los medicamentos prescritos.
1.4 Si los medicamentos están disponibles en diferentes unidades de dosis (por ejemplo, 10 mg, 20 mg) se harán informes por separado para cada unidad de dosis.
1.5 El acceso a los informes de costos se restringirá a usuarios autorizados en la lista de control de acceso administrativo.

Figura 4.1 Requerimientos del usuario y requerimientos del sistema

Es necesario escribir los requerimientos con diferentes niveles de detalle, ya que varios lectores los usarán de distintas formas. La figura 4.2 muestra los posibles lectores de los requerimientos del usuario y los del sistema. De éstos, los primeros por lo general no están interesados en la manera en que se implementará el sistema, y quizá sean administradores a quienes no les atraigan las facilidades detalladas del sistema. Mientras que los segundos necesitan conocer con más precisión qué hará el sistema, ya que están preocupados sobre cómo apoyará los procesos de negocios o porque están inmersos en la implementación del sistema.

En este capítulo se presenta un panorama "tradicional" de los requerimientos, más que de los requerimientos en los procesos ágiles. Para la mayoría de los sistemas grandes, todavía se presenta una fase de ingeniería de requerimientos claramente identificable, antes de comenzar la implementación del sistema. El resultado es un documento de requerimientos que puede formar parte del contrato de desarrollo del sistema. Desde luego, por lo común hay cambios posteriores a los requerimientos, en tanto que los requerimientos del usuario podrían extenderse como requerimientos de sistema más detallados. Sin embargo, el enfoque ágil para alcanzar, al mismo tiempo, los requerimientos a medida que el sistema se desarrolla rara vez se utiliza en el diseño de sistemas grandes.

4.1 Requerimientos funcionales y no funcionales

A menudo, los requerimientos del sistema de software se clasifican como requerimientos funcionales o requerimientos no funcionales:

1. *Requerimientos funcionales* Son enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo

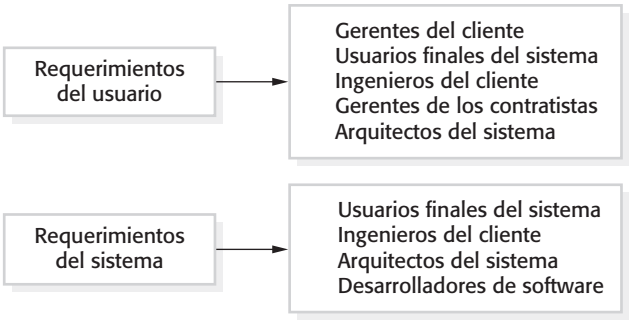


Figura 4.2 Lectores de diferentes tipos de especificación de requerimientos

debería comportarse el sistema en situaciones específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema.

2. *Requerimientos no funcionales* Son limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto de temporización y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema.

En realidad, la distinción entre los diferentes tipos de requerimientos no es tan clara como sugieren estas definiciones sencillas. Un requerimiento de un usuario interesado por la seguridad, como el enunciado que limita el acceso a usuarios autorizados, parecería un requerimiento no funcional. Sin embargo, cuando se desarrolla con más detalle, este requerimiento puede generar otros requerimientos que son evidentemente funcionales, como la necesidad de incluir facilidades de autenticación en el sistema.

Esto muestra que los requerimientos no son independientes y que un requerimiento genera o restringe normalmente otros requerimientos. Por lo tanto, los requerimientos del sistema no sólo detallan los servicios o las características que se requieren del mismo, sino también especifican la funcionalidad necesaria para asegurar que estos servicios y características se entreguen de manera adecuada.

4.1.1 Requerimientos funcionales

Los requerimientos funcionales para un sistema refieren lo que el sistema debe hacer. Tales requerimientos dependen del tipo de software que se esté desarrollando, de los usuarios esperados del software y del enfoque general que adopta la organización cuando se escriben los requerimientos. Al expresarse como requerimientos del usuario, los requerimientos funcionales se describen por lo general de forma abstracta que entiendan los usuarios del sistema. Sin embargo, requerimientos funcionales más específicos del sistema detallan las funciones del sistema, sus entradas y salidas, sus excepciones, etcétera.

Los requerimientos funcionales del sistema varían desde requerimientos generales que cubren lo que tiene que hacer el sistema, hasta requerimientos muy específicos que reflejan maneras locales de trabajar o los sistemas existentes de una organización. Por ejemplo, veamos algunos casos de requerimientos funcionales para el sistema MHC-PMS, que

Requerimientos de dominio

Los requerimientos de dominio se derivan del dominio de aplicación del sistema, más que a partir de las necesidades específicas de los usuarios del sistema. Pueden ser requerimientos funcionales nuevos por derecho propio, restricciones a los requerimientos funcionales existentes o formas en que deben realizarse cálculos particulares.

El problema con los requerimientos de dominio es que los ingenieros de software no pueden entender las características del dominio en que opera el sistema. Por lo común, no pueden indicar si un requerimiento de dominio se perdió o entró en conflicto con otros requerimientos.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

se usan para mantener información de pacientes que reciben tratamiento por problemas de salud mental:

1. Un usuario podrá buscar en todas las clínicas las listas de citas.
2. El sistema elaborará diariamente, para cada clínica, una lista de pacientes que se espera que asistan a cita ese día.
3. Cada miembro del personal que usa el sistema debe identificarse de manera individual con su número de ocho dígitos.

Estos requerimientos funcionales del usuario definen las actividades específicas que debe proporcionar el sistema. Se tomaron del documento de requerimientos del usuario y muestran que los requerimientos funcionales pueden escribirse con diferentes niveles de detalle (contraste los requerimientos 1 y 3).

La inexactitud en la especificación de requerimientos causa muchos problemas en la ingeniería de software. Es natural que un desarrollador de sistemas interprete un requerimiento ambiguo de forma que simplifique su implementación. Sin embargo, con frecuencia, esto no es lo que desea el cliente. Tienen que establecerse nuevos requerimientos y efectuar cambios al sistema. Desde luego, esto aplaza la entrega del sistema y aumenta los costos.

Es el caso del primer ejemplo de requerimiento para el MHC-PMS que establece que un usuario podrá buscar las listas de citas en todas las clínicas. El motivo para este requerimiento es que los pacientes con problemas de salud mental en ocasiones están confundidos. Quizá tengan una cita en una clínica y en realidad acudan a una diferente. De ahí que si tienen una cita, se registrará que asistieron, sin importar la clínica.

Los miembros del personal médico que especifican esto quizás esperen que "buscar" significa que, dado el nombre de un paciente, el sistema busca dicho nombre en las citas de todas las clínicas. Sin embargo, esto no es claro en el requerimiento. Los desarrolladores del sistema pueden interpretar el requerimiento de forma diferente e implementar una búsqueda, de tal modo que el usuario deba elegir una clínica y luego realizar la búsqueda. Evidentemente, esto implicará más entradas del usuario y tomará más tiempo.

En principio, la especificación de los requerimientos funcionales de un sistema debe ser completa y consistente. Totalidad significa que deben definirse todos los servicios requeridos por el usuario. Consistencia quiere decir que los requerimientos tienen que evitar definiciones contradictorias. En la práctica, para sistemas complejos grandes, es

casi imposible lograr la consistencia y la totalidad de los requerimientos. Una causa para ello es la facilidad con que se cometen errores y omisiones al escribir especificaciones para sistemas complejos. Otra es que hay muchos participantes en un sistema grande. Un participante es un individuo o una función que se ve afectado de alguna forma por el sistema. Los participantes tienen diferentes necesidades, pero con frecuencia son inconsistentes. Tales inconsistencias tal vez no sean evidentes cuando se especifican por primera vez los requerimientos, de modo que en la especificación se incluyen requerimientos inconsistentes. Los problemas suelen surgir sólo después de un análisis en profundidad o después de que se entregó el sistema al cliente.

4.1.2 Requerimientos no funcionales

Los requerimientos no funcionales, como indica su nombre, son requerimientos que no se relacionan directamente con los servicios específicos que el sistema entrega a sus usuarios. Pueden relacionarse con propiedades emergentes del sistema, como fiabilidad, tiempo de respuesta y uso de almacenamiento. De forma alternativa, pueden definir restricciones sobre la implementación del sistema, como las capacidades de los dispositivos I/O o las representaciones de datos usados en las interfaces con otros sistemas.

Los requerimientos no funcionales, como el rendimiento, la seguridad o la disponibilidad, especifican o restringen por lo general características del sistema como un todo. Los requerimientos no funcionales a menudo son más significativos que los requerimientos funcionales individuales. Es común que los usuarios del sistema encuentren formas para trabajar en torno a una función del sistema que realmente no cubre sus necesidades. No obstante, el fracaso para cubrir los requerimientos no funcionales haría que todo el sistema fuera inútil. Por ejemplo, si un sistema de aeronave no cubre sus requerimientos de fiabilidad, no será certificado para su operación como dispositivo seguro; si un sistema de control embebido fracasa para cubrir sus requerimientos de rendimiento, no operarán correctamente las funciones de control.

Aunque es posible identificar con regularidad cuáles componentes de sistema implementan requerimientos funcionales específicos (por ejemplo, hay componentes de formateo que implementan requerimientos de informe), por lo general es más difícil relacionar componentes con requerimientos no funcionales. La implementación de dichos requerimientos puede propagarse a lo largo del sistema. Para esto existen dos razones:

1. Los requerimientos no funcionales afectan más la arquitectura global de un sistema que los componentes individuales. Por ejemplo, para garantizar que se cumplan los requerimientos de rendimiento, quizá se deba organizar el sistema para minimizar las comunicaciones entre componentes.
2. Un requerimiento no funcional individual, como un requerimiento de seguridad, podría generar algunos requerimientos funcionales relacionados que definan nuevos servicios del sistema que se requieran. Además, también podría generar requerimientos que restrinjan los requerimientos ya existentes.

Los requerimientos no funcionales surgen a través de necesidades del usuario, debido a restricciones presupuestales, políticas de la organización, necesidad de interoperabilidad con otro software o sistemas de hardware, o factores externos como regulaciones de seguridad o legislación sobre privacidad. La figura 4.3 es una clasificación de requerimientos

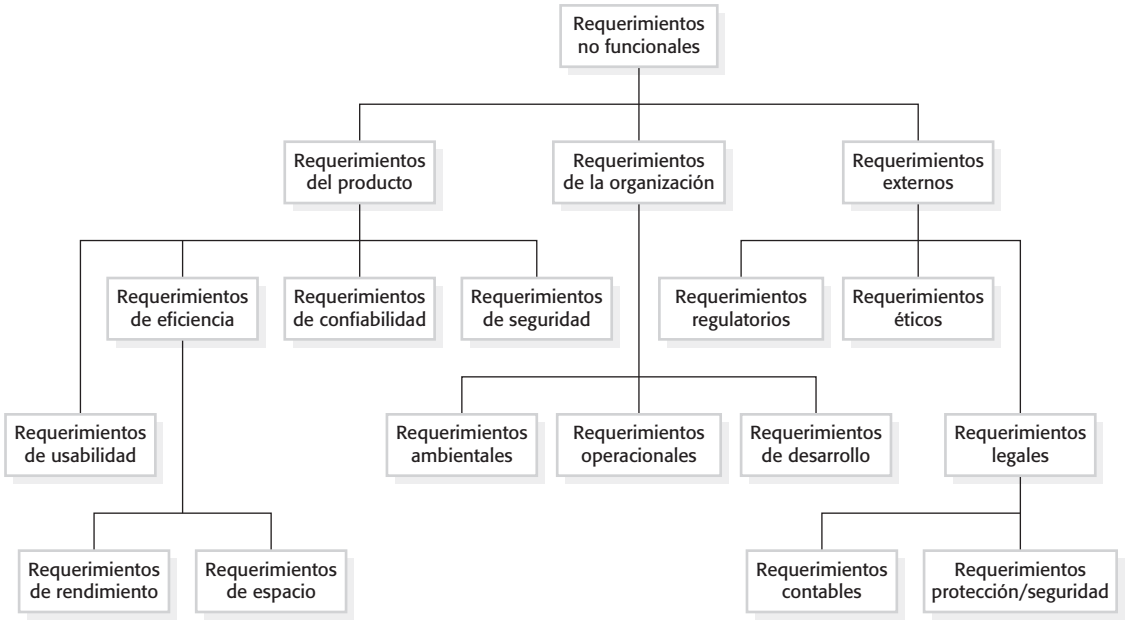


Figura 4.3 Tipos de requerimientos no funcionales

no funcionales. Observe a partir de este diagrama que los requerimientos no funcionales provienen de características requeridas del software (requerimientos del producto), la organización que desarrolla el software (requerimientos de la organización) o de fuentes externas:

1. *Requerimientos del producto* Estos requerimientos especifican o restringen el comportamiento del software. Los ejemplos incluyen requerimientos de rendimiento sobre qué tan rápido se debe ejecutar el sistema y cuánta memoria requiere, requerimientos de fiabilidad que establecen la tasa aceptable de fallas, requerimientos de seguridad y requerimientos de usabilidad.
2. *Requerimientos de la organización* Son requerimientos de sistemas amplios, derivados de políticas y procedimientos en la organización del cliente y del desarrollador. Los ejemplos incluyen requerimientos del proceso operacional que definen cómo se usará el sistema, requerimientos del proceso de desarrollo que especifican el lenguaje de programación, estándares del entorno o el proceso de desarrollo a utilizar, y requerimientos ambientales que definen el entorno de operación del sistema.
3. *Requerimientos externos* Este término cubre todos los requerimientos derivados de factores externos al sistema y su proceso de desarrollo. En ellos se incluyen requerimientos regulatorios que establecen lo que debe hacer el sistema para ser aprobado en su uso por un regulador, como sería un banco central; requerimientos legislativos que tienen que seguirse para garantizar que el sistema opere conforme a la ley, y requerimientos éticos que garanticen que el sistema será aceptable para sus usuarios y el público en general.

La figura 4.4 muestra ejemplos de requerimientos del producto, de la organización y requerimientos externos tomados del MHC-PMS, cuyos requerimientos de usuario se

REQUERIMIENTO DEL PRODUCTO
El MHC-PMS estará disponible en todas las clínicas durante las horas de trabajo normales (lunes a viernes, de 8:30 a 17:30). En cualquier día, los tiempos muertos dentro de las horas laborales normales no rebasarán los cinco segundos.

REQUERIMIENTOS DE LA ORGANIZACIÓN
Los usuarios del sistema MHC-PMS se acreditarán a sí mismos con el uso de la tarjeta de identidad de la autoridad sanitaria.

REQUERIMIENTOS EXTERNOS
Como establece la HStan-03-2006-priv, el sistema implementará provisiones para la privacidad del paciente.

Figura 4.4 Ejemplos de requerimientos no funcionales en el MHC-PMS

introdujeron en la sección 4.1.1. El requerimiento del producto es un requerimiento de disponibilidad que define cuándo estará disponible el sistema y el tiempo muerto permitido cada día. No dice algo sobre la funcionalidad del MHC-PMS e identifica con claridad una restricción que deben considerar los diseñadores del sistema.

El requerimiento de la organización especifica cómo se autentican los usuarios en el sistema. La autoridad sanitaria que opera el sistema se mueve hacia un procedimiento de autenticación estándar para cualquier software donde, en vez de que los usuarios tengan un nombre de conexión (login), pasan su tarjeta de identidad por un lector para identificarse a sí mismos. El requerimiento externo se deriva de la necesidad de que el sistema esté conforme con la legislación de privacidad. Evidentemente, la privacidad es un asunto muy importante en los sistemas de atención a la salud, y el requerimiento especifica que el sistema debe desarrollarse conforme a un estándar de privacidad nacional.

Un problema común con requerimientos no funcionales es que los usuarios o clientes con frecuencia proponen estos requerimientos como metas generales, como facilidad de uso, capacidad de que el sistema se recupere de fallas, o rapidez de respuesta al usuario. Las metas establecen buenas intenciones; no obstante, ocasionan problemas a los desarrolladores del sistema, pues dejan espacio para la interpretación y la disputa posterior una vez que se entregue el sistema. Por ejemplo, la siguiente meta del sistema es típica de cómo un administrador expresa los requerimientos de usabilidad:

> *Para el personal médico debe ser fácil usar el sistema, y este último debe organizarse de tal forma que minimice los errores del usuario.*

Lo anterior se escribió para mostrar cómo podría expresarse la meta como un requerimiento no funcional "comprobable". Aun cuando es imposible comprobar de manera objetiva la meta del sistema, en la siguiente descripción se puede incluir, al menos, la instrumentación de software para contar los errores cometidos por los usuarios cuando prueban el sistema.

> *Después de cuatro horas de capacitación, el personal médico usará todas las funciones del sistema. Después de esta capacitación, los usuarios experimentados no deberán superar el promedio de dos errores cometidos por hora de uso del sistema.*

Siempre que sea posible, se deberán escribir de manera cuantitativa los requerimientos no funcionales, de manera que puedan ponerse objetivamente a prueba. La figura 4.5 muestra las métricas que se utilizan para especificar propiedades no funcionales del sistema.

Propiedad	Medida
Rapidez	Transacciones/segundo procesadas Tiempo de respuesta usuario/evento Tiempo de regeneración de pantalla
Tamaño	Mbytes Número de chips ROM
Facilidad de uso	Tiempo de capacitación Número de cuadros de ayuda
Fiabilidad	Tiempo medio para falla Probabilidad de indisponibilidad Tasa de ocurrencia de falla Disponibilidad
Robustez	Tiempo de reinicio después de falla Porcentaje de eventos que causan falla Probabilidad de corrupción de datos en falla
Portabilidad	Porcentaje de enunciados dependientes de objetivo Número de sistemas objetivo

Figura 4.5 Métricas para especificar requerimientos no funcionales

Usted puede medir dichas características cuando el sistema se pone a prueba para comprobar si éste cumple o no cumple con sus requerimientos no funcionales.

En la práctica, los usuarios de un sistema suelen encontrar difícil traducir sus metas en requerimientos mensurables. Para algunas metas, como la mantenibilidad, no hay métricas para usarse. En otros casos, incluso cuando sea posible la especificación cuantitativa, los clientes no logran relacionar sus necesidades con dichas especificaciones. No comprenden qué significa algún número que define la fiabilidad requerida (por así decirlo), en términos de su experiencia cotidiana con los sistemas de cómputo. Más aún, el costo por verificar objetivamente los requerimientos no funcionales mensurables suele ser muy elevado, y los clientes que pagan por el sistema quizá piensen que dichos costos no están justificados.

Los requerimientos no funcionales entran a menudo en conflicto e interactúan con otros requerimientos funcionales o no funcionales. Por ejemplo, el requerimiento de autenticación en la figura 4.4 requiere, indiscutiblemente, la instalación de un lector de tarjetas en cada computadora unida al sistema. Sin embargo, podría haber otro requerimiento que solicite acceso móvil al sistema desde las computadoras portátiles de médicos o enfermeras. Por lo general, las computadoras portátiles no están equipadas con lectores de tarjeta, de modo que, ante tales circunstancias, probablemente deba permitirse algún método de autenticación alternativo.

En la práctica, en el documento de requerimientos, resulta difícil separar los requerimientos funcionales de los no funcionales. Si los requerimientos no funcionales se expresan por separado de los requerimientos funcionales, las relaciones entre ambos serían difíciles de entender. No obstante, se deben destacar de manera explícita los requerimientos que están claramente relacionados con las propiedades emergentes del sistema, como el rendimiento o la fiabilidad. Esto se logra al ponerlos en una sección separada del documento de requerimientos o al distinguirlos, en alguna forma, de otros requerimientos del sistema.

Estándares del documento de requerimientos

Algunas organizaciones grandes, como el Departamento de Defensa estadounidense y el Institute of Electrical and Electronic Engineers (IEEE), definieron estándares para los documentos de requerimientos. Comúnmente son muy genéricos, pero útiles como base para desarrollar estándares organizativos más detallados. El IEEE es uno de los proveedores de estándares mejor conocidos y desarrolló un estándar para la estructura de documentos de requerimientos. Este estándar es más adecuado para sistemas como comando militar y sistemas de control que tienen un largo tiempo de vida y, por lo general, los diseña un grupo de organizaciones.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Los requerimientos no funcionales, como los requerimientos de fiabilidad, protección y confidencialidad, son en particular importantes para los sistemas fundamentales. En el capítulo 12 se incluyen estos requerimientos, donde se describen técnicas específicas para definir requerimientos de confiabilidad y seguridad.

4.2 El documento de requerimientos de software

El documento de requerimientos de software (llamado algunas veces especificación de requerimientos de software o SRS) es un comunicado oficial de lo que deben implementar los desarrolladores del sistema. Incluye tanto los requerimientos del usuario para un sistema, como una especificación detallada de los requerimientos del sistema. En ocasiones, los requerimientos del usuario y del sistema se integran en una sola descripción. En otros casos, los requerimientos del usuario se definen en una introducción a la especificación de requerimientos del sistema. Si hay un gran número de requerimientos, los requerimientos del sistema detallados podrían presentarse en un documento aparte.

Son esenciales los documentos de requerimientos cuando un contratista externo diseña el sistema de software. Sin embargo, los métodos de desarrollo ágiles argumentan que los requerimientos cambian tan rápidamente que un documento de requerimientos se vuelve obsoleto tan pronto como se escribe, así que el esfuerzo se desperdicia en gran medida. En lugar de un documento formal, los enfoques como la programación extrema (Beck, 1999) recopilan de manera incremental requerimientos del usuario y los escriben en tarjetas como historias de usuario. De esa manera, el usuario da prioridad a los requerimientos para su implementación en el siguiente incremento del sistema.

Este enfoque es adecuado para sistemas empresariales donde los requerimientos son inestables. Sin embargo, aún resulta útil escribir un breve documento de apoyo que defina los requerimientos de la empresa y los requerimientos de confiabilidad para el sistema; es fácil olvidar los requerimientos que se aplican al sistema como un todo, cuando uno se enfoca en los requerimientos funcionales para la siguiente liberación del sistema.

El documento de requerimientos tiene un conjunto variado de usuarios, desde el administrador ejecutivo de la organización que paga por el sistema, hasta los ingenieros responsables del desarrollo del software. La figura 4.6, tomada del libro del autor con Gerald Kotonya sobre ingeniería de requerimientos (Kotonya y Sommerville, 1998), muestra a los posibles usuarios del documento y cómo ellos lo utilizan.

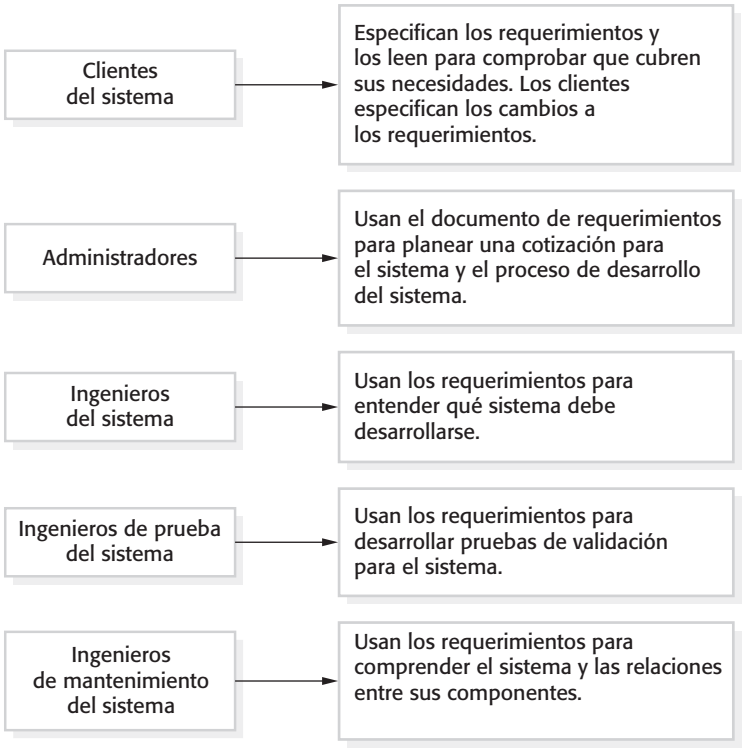


Figura 4.6 Usuarios de un documento de requerimientos

La diversidad de posibles usuarios significa que el documento de requerimientos debe ser un compromiso entre la comunicación de los requerimientos a los clientes, la definición de los requerimientos con detalle preciso para desarrolladores y examinadores, y la inclusión de información sobre la posible evolución del sistema. La información de cambios anticipados ayuda tanto a los diseñadores del sistema a evitar decisiones de diseño restrictivas, como a los ingenieros de mantenimiento del sistema que deben adaptar el sistema a los nuevos requerimientos.

El nivel de detalle que se incluya en un documento de requerimientos depende del tipo de sistema a diseñar y el proceso de desarrollo utilizado. Los sistemas críticos necesitan tener requerimientos detallados porque la seguridad y la protección también deben analizarse de forma pormenorizada. Cuando el sistema lo desarrolla una compañía independiente (por ejemplo, mediante la subcontratación), deben detallarse y precisarse las especificaciones del sistema. Si se utiliza un proceso de desarrollo iterativo interno, entonces el documento de requerimientos suele ser mucho menos detallado y cualquier ambigüedad puede resolverse durante el desarrollo del sistema.

La figura 4.7 indica una posible organización para un documento de requerimientos basada en un estándar del IEEE para documentos de requerimientos (IEEE, 1998). Este estándar es genérico y se adapta a usos específicos. En este caso, el estándar se extendió para incluir información de la evolución prevista del sistema. Esta información ayuda a los encargados del sistema y permite a los diseñadores incluir soporte para características futuras del sistema.

Naturalmente, la información que se incluya en un documento de requerimientos depende del tipo de software que se va a desarrollar y del enfoque para el desarrollo que se use. Si se adopta un enfoque evolutivo para un producto de software (por ejemplo), el

Capítulo	Descripción
Prefacio	Debe definir el número esperado de lectores del documento, así como describir su historia de versiones, incluidas las causas para la creación de una nueva versión y un resumen de los cambios realizados en cada versión.
Introducción	Describe la necesidad para el sistema. Debe detallar brevemente las funciones del sistema y explicar cómo funcionará con otros sistemas. También tiene que indicar cómo se ajusta el sistema en los objetivos empresariales o estratégicos globales de la organización que comisiona el software.
Glosario	Define los términos técnicos usados en el documento. No debe hacer conjeturas sobre la experiencia o la habilidad del lector.
Definición de requerimientos del usuario	Aquí se representan los servicios que ofrecen al usuario. También, en esta sección se describen los requerimientos no funcionales del sistema. Esta descripción puede usar lenguaje natural, diagramas u otras observaciones que sean comprensibles para los clientes. Deben especificarse los estándares de producto y proceso que tienen que seguirse.
Arquitectura del sistema	Este capítulo presenta un panorama de alto nivel de la arquitectura anticipada del sistema, que muestra la distribución de funciones a través de los módulos del sistema. Hay que destacar los componentes arquitectónicos que sean de reutilización.
Especificación de requerimientos del sistema	Debe representar los requerimientos funcionales y no funcionales con más detalle. Si es preciso, también pueden detallarse más los requerimientos no funcionales. Pueden definirse las interfaces a otros sistemas.
Modelos del sistema	Pueden incluir modelos gráficos del sistema que muestren las relaciones entre componentes del sistema, el sistema y su entorno. Ejemplos de posibles modelos son los modelos de objeto, modelos de flujo de datos o modelos de datos semánticos.
Evolución del sistema	Describe los supuestos fundamentales sobre los que se basa el sistema, y cualquier cambio anticipado debido a evolución de hardware, cambio en las necesidades del usuario, etc. Esta sección es útil para los diseñadores del sistema, pues los ayuda a evitar decisiones de diseño que restringirían probablemente futuros cambios al sistema.
Apéndices	Brindan información específica y detallada que se relaciona con la aplicación a desarrollar; por ejemplo, descripciones de hardware y bases de datos. Los requerimientos de hardware definen las configuraciones, mínima y óptima, del sistema. Los requerimientos de base de datos delimitan la organización lógica de los datos usados por el sistema y las relaciones entre datos.
Índice	Pueden incluirse en el documento varios índices. Así como un índice alfabético normal, uno de diagramas, un índice de funciones, etcétera.

Figura 4.7 Estructura de un documento de requerimientos

documento de requerimientos dejará fuera muchos de los capítulos detallados que se sugirieron anteriormente. El enfoque estará en especificar los requerimientos del usuario y los requerimientos no funcionales de alto nivel del sistema. En este caso, diseñadores y programadores usan su criterio para decidir cómo cubrir los requerimientos establecidos del usuario para el sistema.

Sin embargo, cuando el software sea parte de un proyecto de sistema grande que incluya la interacción de sistemas de hardware y software, será necesario por lo general

Problemas con el uso de lenguaje natural para la especificación de requerimientos

La flexibilidad del lenguaje natural, que es tan útil para la especificación, causa problemas frecuentemente. Hay espacio para escribir requerimientos poco claros, y los lectores (los diseñadores) pueden malinterpretar los requerimientos porque tienen un antecedente diferente al del usuario. Es fácil mezclar muchos requerimientos en una sola oración y quizá sea difícil estructurar los requerimientos en lenguaje natural.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

definir los requerimientos a un nivel detallado. Esto significa que es probable que los documentos de requerimientos sean muy largos y deban incluir la mayoría, si no es que todos, los capítulos que se muestran en la figura 4.7. Para documentos extensos, es muy importante incluir una tabla de contenido global y un índice del documento, de manera que los lectores encuentren con facilidad la información que necesitan.

4.3 Especificación de requerimientos

La especificación de requerimientos es el proceso de escribir, en un documento de requerimientos, los requerimientos del usuario y del sistema. De manera ideal, los requerimientos del usuario y del sistema deben ser claros, sin ambigüedades, fáciles de entender, completos y consistentes. Esto en la práctica es difícil de lograr, pues los participantes interpretan los requerimientos de formas diferentes y con frecuencia en los requerimientos hay conflictos e inconsistencias inherentes.

Los requerimientos del usuario para un sistema deben describir los requerimientos funcionales y no funcionales, de forma que sean comprensibles para los usuarios del sistema que no cuentan con un conocimiento técnico detallado. De manera ideal, deberían especificar sólo el comportamiento externo del sistema. El documento de requerimientos no debe incluir detalles de la arquitectura o el diseño del sistema. En consecuencia, si usted escribe los requerimientos del usuario, no tiene que usar jerga de software, anotaciones estructuradas o formales. Debe escribir los requerimientos del usuario en lenguaje natural, con tablas y formas sencillas, así como diagramas intuitivos.

Los requerimientos del sistema son versiones extendidas de los requerimientos del usuario que los ingenieros de software usan como punto de partida para el diseño del sistema. Añaden detalles y explican cómo el sistema debe brindar los requerimientos del usuario. Se pueden usar como parte del contrato para la implementación del sistema y, por lo tanto, deben ser una especificación completa y detallada de todo el sistema.

Idealmente, los requerimientos del sistema deben describir de manera simple el comportamiento externo del sistema y sus restricciones operacionales. No tienen que ocuparse de cómo se diseña o implementa el sistema. Sin embargo, al nivel de detalle requerido para especificar por completo un sistema de software complejo, es prácticamente imposible excluir toda la información de diseño. Para ello existen varias razones:

1. Tal vez se tenga que diseñar una arquitectura inicial del sistema para ayudar a estructurar la especificación de requerimientos. Los requerimientos del sistema se organizan

Notación	Descripción
Enunciados en lenguaje natural	Los requerimientos se escriben al usar enunciados numerados en lenguaje natural. Cada enunciado debe expresar un requerimiento.
Lenguaje natural estructurado	Los requerimientos se escriben en lenguaje natural en una forma o plantilla estándar. Cada campo ofrece información de un aspecto del requerimiento.
Lenguajes de descripción de diseño	Este enfoque usa un lenguaje como un lenguaje de programación, pero con características más abstractas para especificar los requerimientos al definir un modelo operacional del sistema. Aunque en la actualidad este enfoque se usa raras veces, aún tiene utilidad para especificaciones de interfaz.
Anotaciones gráficas	Los modelos gráficos, complementados con anotaciones de texto, sirven para definir los requerimientos funcionales del sistema; los casos de uso del UML y los diagramas de secuencia se emplean de forma común.
Especificaciones matemáticas	Dichas anotaciones se basan en conceptos matemáticos como máquinas o conjuntos de estado finito. Aunque tales especificaciones sin ambigüedades pueden reducir la imprecisión en un documento de requerimientos, la mayoría de los clientes no comprenden una especificación formal. No pueden comprobar que representa lo que quieren y por ello tienen reticencia para aceptarlo como un contrato de sistema.

Figura 4.8 Formas de escribir una especificación de requerimientos del sistema

de acuerdo con los diferentes subsistemas que constituyen el sistema. Como veremos en los capítulos 6 y 18, esta definición arquitectónica es esencial si usted quiere reutilizar componentes de software al implementar el sistema.

2. En la mayoría de los casos, los sistemas deben interoperar con los sistemas existentes, lo cual restringe el diseño e impone requerimientos sobre el nuevo sistema.
3. Quizá sea necesario el uso de una arquitectura específica para cubrir los requerimientos no funcionales (como la programación N-versión para lograr fiabilidad, que se estudia en el capítulo 13). Un regulador externo, que precise certificar que dicho sistema es seguro, puede especificar que se utilice un diseño arquitectónico ya avalado.

Los requerimientos del usuario se escriben casi siempre en lenguaje natural, complementado con diagramas y tablas adecuados en el documento de requerimientos. Los requerimientos del sistema se escriben también en lenguaje natural, pero de igual modo se utilizan otras notaciones basadas en formas, modelos gráficos del sistema o modelos matemáticos del sistema. La figura 4.8 resume las posibles anotaciones que podrían usarse para escribir requerimientos del sistema.

Los modelos gráficos son más útiles cuando es necesario mostrar cómo cambia un estado o al describir una secuencia de acciones. Los gráficos de secuencia UML y los gráficos de estado, que se explican en el capítulo 5, exponen la secuencia de acciones que ocurren en respuesta a cierto mensaje o evento. En ocasiones, se usan especificaciones matemáticas formales con la finalidad de describir los requerimientos para sistemas de protección o seguridad críticos, aunque rara vez se usan en otras circunstancias. Este enfoque para escribir especificaciones se explica en el capítulo 12.

3.2 Si se requiere, cada 10 minutos el sistema medirá el azúcar en la sangre y administrará insulina. (*Los cambios de azúcar en la sangre son relativamente lentos, de manera que no son necesarias mediciones más frecuentes; la medición menos periódica podría conducir a niveles de azúcar innecesariamente elevados.*)

3.6 Cada minuto, el sistema debe correr una rutina de autoevaluación, con las condiciones a probar y las acciones asociadas definidas en la tabla 1. (*Una rutina de autoevaluación puede detectar problemas de hardware y software, y prevenir al usuario sobre el hecho de que la operación normal puede ser imposible.*)

Figura 4.9 Ejemplo de requerimientos para el sistema de software de la bomba de insulina

4.3.1 Especificación en lenguaje natural

Desde los albores de la ingeniería de software, el lenguaje natural se usa para escribir los requerimientos de software. Es expresivo, intuitivo y universal. También es potencialmente vago, ambiguo y su significado depende de los antecedentes del lector. Como resultado, hay muchas propuestas para formas alternativas de escribir los requerimientos. Sin embargo, ninguna se ha adoptado de manera amplia, por lo que el lenguaje natural seguirá siendo la forma más usada para especificar los requerimientos del sistema y del software.

Para minimizar la interpretación errónea al escribir los requerimientos en lenguaje natural, se recomienda seguir algunos lineamientos sencillos:

1. Elabore un formato estándar y asegúrese de que todas las definiciones de requerimientos se adhieran a dicho formato. Al estandarizar el formato es menos probable cometer omisiones y más sencillo comprobar los requerimientos. El formato que usa el autor expresa el requerimiento en una sola oración. A cada requerimiento de usuario se asocia un enunciado de razones para explicar por qué se propuso el requerimiento. Las razones también pueden incluir información sobre quién planteó el requerimiento (la fuente del requerimiento), de modo que usted conozca a quién consultar en caso de que cambie el requerimiento.
2. Utilice el lenguaje de manera clara para distinguir entre requerimientos obligatorios y deseables. Los primeros son requerimientos que el sistema debe soportar y, por lo general, se escriben en futuro "debe ser". En tanto que los requerimientos deseables no son necesarios y se escriben en tiempo pospretérito o como condicional "debería ser".
3. Use texto resaltado (negrilla, cursiva o color) para seleccionar partes clave del requerimiento.
4. No deduzca que los lectores entienden el lenguaje técnico de la ingeniería de software. Es fácil que se malinterpreten palabras como "arquitectura" y "módulo". Por lo tanto, debe evitar el uso de jerga, abreviaturas y acrónimos.
5. Siempre que sea posible, asocie una razón con cada requerimiento de usuario. La razón debe explicar por qué se incluyó el requerimiento. Es particularmente útil cuando los requerimientos cambian, pues ayuda a decidir cuáles cambios serían indeseables.

La figura 4.9 ilustra cómo se usan dichos lineamientos. Incluye dos requerimientos para el software embebido para la bomba de insulina automatizada, que se introdujo en el capítulo 1. Usted puede descargar la especificación completa de los requerimientos de la bomba de insulina en las páginas Web del libro.

Bomba de insulina/Software de control/SRS/3.3.2

Función	Calcula dosis de insulina: nivel seguro de azúcar.
Descripción	Calcula la dosis de insulina que se va a suministrar cuando la medición del nivel de azúcar actual esté en zona segura entre 3 y 7 unidades.
Entradas	Lectura del azúcar actual (r2), las dos lecturas previas (r0 y r1).
Fuente	Lectura del azúcar actual del sensor. Otras lecturas de la memoria.
Salidas	CompDose: la dosis de insulina a administrar.
Destino	Ciclo de control principal.
Acción	CompDose es cero si es estable el nivel de azúcar, o cae o si aumenta el nivel pero disminuye la tasa de aumento. Si el nivel se eleva y la tasa de aumento crece, CompDose se calcula entonces al dividir la diferencia entre el nivel de azúcar actual y el nivel previo entre 4 y redondear el resultado. Si la suma se redondea a cero, en tal caso CompDose se establece en la dosis mínima que puede entregarse.
Requerimientos	Dos lecturas previas, de modo que puede calcularse la tasa de cambio del nivel de azúcar.
Precondición	El depósito de insulina contiene al menos la dosis individual de insulina máxima permitida.
Postcondición	r0 se sustituye con r1, luego r1 se sustituye con r2.
Efectos colaterales	Ninguno.

Figura 4.10 Especificación estructurada de un requerimiento para una bomba de insulina

4.3.2 Especificaciones estructuradas

El lenguaje natural estructurado es una manera de escribir requerimientos del sistema, donde está limitada la libertad del escritor de requerimientos y todos éstos se anotan en una forma estándar. Aunque este enfoque conserva la mayoría de la expresividad y comprensibilidad del lenguaje natural, asegura que haya cierta uniformidad sobre la especificación. Las anotaciones en lenguaje estructurado emplean plantillas para especificar requerimientos del sistema. La especificación utiliza constructos de lenguaje de programación para mostrar alternativas e iteración, y destaca elementos clave con el uso de sombreado o de fuentes distintas.

Los Robertson (Robertson y Robertson, 1999), en su libro del método de ingeniería de requerimientos VOLERE, recomiendan que se escriban los requerimientos del usuario inicialmente en tarjetas, un requerimiento por tarjeta. Proponen algunos campos en cada tarjeta, tales como razones de los requerimientos, dependencias en otros requerimientos, fuente de los requerimientos, materiales de apoyo, etcétera. Lo anterior es similar al enfoque utilizado en el ejemplo de la especificación estructurada que se muestra en la figura 4.10.

Para usar un enfoque estructurado que especifique los requerimientos de sistema, hay que definir una o más plantillas estándar para requerimientos, y representar dichas plantillas como formas estructuradas. La especificación puede estructurarse sobre los objetos manipulados por el sistema, las funciones que el sistema realiza o los eventos procesados por el sistema. En la figura 4.10 se muestra un ejemplo de una especificación basada en la forma, en este caso, una que define cómo calcular la dosis de insulina a administrar cuando el azúcar en la sangre está dentro de una banda segura.

Condición	Acción
Nivel de azúcar en descenso (r2 < r1)	CompDose = 0
Nivel de azúcar estable (r2 = r1)	CompDose = 0
Nivel de azúcar creciente y tasa de incremento decreciente ((r2 − r1) < (r1 − r0))	CompDose = 0
Nivel de azúcar creciente y tasa de incremento estable o creciente ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If resultado redondeado = 0 then CompDose = MinimumDose

Figura 4.11 Especificación tabular del cálculo para una bomba de insulina

Cuando use una forma estándar para especificar requerimientos funcionales, debe incluir la siguiente información:

1. Una descripción de la función o entidad a especificar.
2. Una descripción de sus entradas y sus procedencias.
3. Una descripción de sus salidas y a dónde se dirigen.
4. Información sobre los datos requeridos para el cálculo u otras entidades en el sistema que se utilizan (la parte "requiere").
5. Una descripción de la acción que se va a tomar.
6. Si se usa un enfoque funcional, una precondición establece lo que debe ser verdadero antes de llamar a la función, y una postcondición especifica lo que es verdadero después de llamar a la función.
7. Una descripción de los efectos colaterales (si acaso hay alguno) de la operación.

Al usar especificaciones estructuradas se eliminan algunos de los problemas de la especificación en lenguaje natural. La variabilidad en la especificación se reduce y los requerimientos se organizan de forma más efectiva. Sin embargo, en ocasiones todavía es difícil escribir requerimientos sin ambigüedades, en particular cuando deben especificarse cálculos complejos (por ejemplo, cómo calcular la dosis de insulina).

Para enfrentar este problema se puede agregar información extra a los requerimientos en lenguaje natural, por ejemplo, con el uso de tablas o modelos gráficos del sistema. Éstos pueden mostrar cómo proceden los cálculos, cambia el estado del sistema, interactúan los usuarios con el sistema y se realizan las secuencias de acciones.

Las tablas son particularmente útiles cuando hay algunas posibles situaciones alternas y se necesita describir las acciones a tomar en cada una de ellas. La bomba de insulina fundamenta sus cálculos del requerimiento de insulina, en la tasa de cambio de los niveles de azúcar en la sangre. Las tasas de cambio se calculan con las lecturas, actual y anterior. La figura 4.11 es una descripción tabular de cómo se usa la tasa de cambio del azúcar en la sangre, para calcular la cantidad de insulina por suministrar.

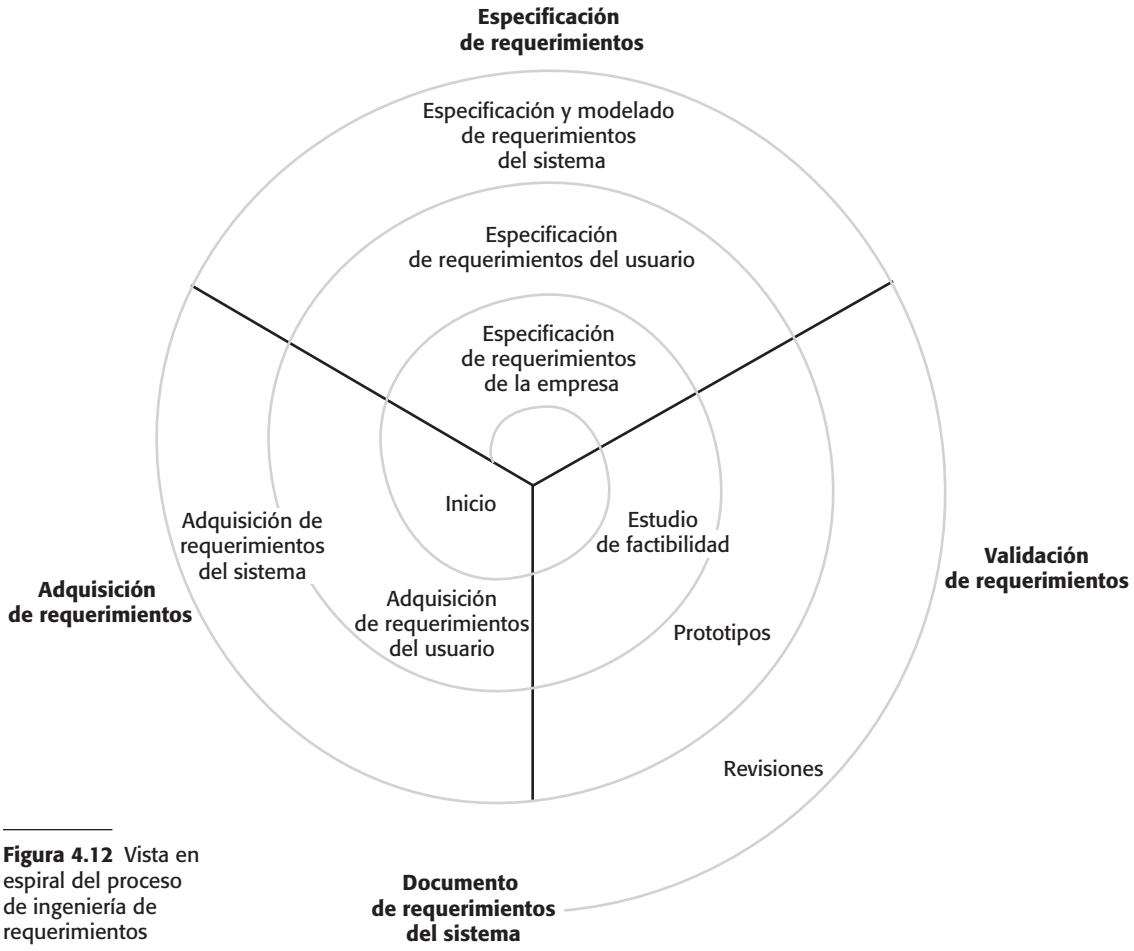


Figura 4.12 Vista en espiral del proceso de ingeniería de requerimientos

4.4 Procesos de ingeniería de requerimientos

Como vimos en el capítulo 2, los procesos de ingeniería de requerimientos incluyen cuatro actividades de alto nivel. Éstas se enfocan en valorar si el sistema es útil para la empresa (estudio de factibilidad), descubrir requerimientos (adquisición y análisis), convertir dichos requerimientos en alguna forma estándar (especificación) y comprobar que los requerimientos definan realmente el sistema que quiere el cliente (validación). En la figura 2.6 se mostró esto como proceso secuencial; sin embargo, en la práctica, la ingeniería de requerimientos es un proceso iterativo donde las actividades están entrelazadas.

La figura 4.12 presenta este entrelazamiento. Las actividades están organizadas como un proceso iterativo alrededor de una espiral, y la salida es un documento de requerimientos del sistema. La cantidad de tiempo y esfuerzo dedicados a cada actividad en cada iteración depende de la etapa del proceso global y el tipo de sistema que está siendo desarrollado. En el inicio del proceso, se empleará más esfuerzo para comprender los requerimientos empresariales de alto nivel y los no funcionales, así como los requerimientos del

Estudios de factibilidad

Un estudio de factibilidad es un breve estudio enfocado que debe realizarse con oportunidad en el proceso de IR. Debe responder tres preguntas clave: *a*) ¿El sistema contribuye con los objetivos globales de la organización? *b*) ¿El sistema puede implementarse dentro de la fecha y el presupuesto usando la tecnología actual? *c*) ¿El sistema puede integrarse con otros sistemas que se utilicen?

Si la respuesta a cualquiera de estas preguntas es negativa, probablemente no sea conveniente continuar con el proyecto.

http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html

usuario para el sistema. Más adelante en el proceso, en los anillos exteriores de la espiral, se dedicará más esfuerzo a la adquisición y comprensión de los requerimientos detallados del sistema.

Este modelo en espiral acomoda enfoques al desarrollo, donde los requerimientos se elaboraron con diferentes niveles de detalle. El número de iteraciones de la espiral tiende a variar, de modo que la espiral terminará después de adquirir algunos o todos los requerimientos del usuario. Se puede usar el desarrollo ágil en vez de la creación de prototipos, de manera que se diseñen en conjunto los requerimientos y la implementación del sistema.

Algunas personas consideran la ingeniería de requerimientos como el proceso de aplicar un método de análisis estructurado, tal como el análisis orientado a objetos (Larman, 2002). Esto implica analizar el sistema y desarrollar un conjunto de modelos gráficos del sistema, como los modelos de caso de uso, que luego sirven como especificación del sistema. El conjunto de modelos describe el comportamiento del sistema y se anota con información adicional que describe, por ejemplo, el rendimiento o la fiabilidad requeridos del sistema.

Aunque los métodos estructurados desempeñan un papel en el proceso de ingeniería de requerimientos, hay mucho más ingeniería de requerimientos de la que se cubre con dichos métodos. La adquisición de requerimientos, en particular, es una actividad centrada en la gente, y a las personas no les gustan las restricciones impuestas por modelos de sistema rígidos.

Prácticamente en todos los sistemas cambian los requerimientos. Las personas implicadas desarrollan una mejor comprensión de qué quieren que haga el software; la organización que compra el sistema cambia; se hacen modificaciones al hardware, al software y al entorno organizacional del sistema. Al proceso de administrar tales requerimientos cambiantes se le llama administración de requerimientos, tema que se trata en la sección 4.7.

4.5 Adquisición y análisis de requerimientos

Después de un estudio de factibilidad inicial, la siguiente etapa del proceso de ingeniería de requerimientos es la adquisición y el análisis de requerimientos. En esta actividad, los ingenieros de software trabajan con clientes y usuarios finales del sistema para descubrir el dominio de aplicación, qué servicios debe proporcionar el sistema, el desempeño requerido de éste, las restricciones de hardware, etcétera.

Figura 4.13 El proceso de adquisición y análisis de requerimientos

En una organización, la adquisición y el análisis de requerimientos pueden involucrar a diversas clases de personas. Un participante en el sistema es quien debe tener alguna influencia directa o indirecta sobre los requerimientos del mismo. Los participantes incluyen a usuarios finales que interactuarán con el sistema, y a cualquiera en una organización que resultará afectada por él. Otros participantes del sistema pueden ser los ingenieros que desarrollan o mantienen otros sistemas relacionados, administradores de negocios, expertos de dominio y representantes de asociaciones sindicales.

En la figura 4.13 se muestra un modelo del proceso de adquisición y análisis. Cada organización tendrá su versión o ejemplificación de este modelo general, dependiendo de factores locales, tales como experiencia del personal, tipo de sistema a desarrollar, estándares usados, etcétera.

Las actividades del proceso son:

1. *Descubrimiento de requerimientos* Éste es el proceso de interactuar con los participantes del sistema para descubrir sus requerimientos. También los requerimientos de dominio de los participantes y la documentación se descubren durante esta actividad. Existen numerosas técnicas complementarias que pueden usarse para el descubrimiento de requerimientos, las cuales se estudian más adelante en esta sección.

2. *Clasificación y organización de requerimientos* Esta actividad toma la compilación no estructurada de requerimientos, agrupa requerimientos relacionados y los organiza en grupos coherentes. La forma más común de agrupar requerimientos es usar un modelo de la arquitectura del sistema, para identificar subsistemas y asociar los requerimientos con cada subsistema. En la práctica, la ingeniería de requerimientos y el diseño arquitectónico no son actividades separadas completamente.

3. *Priorización y negociación de requerimientos* Inevitablemente, cuando intervienen diversos participantes, los requerimientos entrarán en conflicto. Esta actividad se preocupa por priorizar los requerimientos, así como por encontrar y resolver conflictos de requerimientos mediante la negociación. Por lo general, los participantes tienen que reunirse para resolver las diferencias y estar de acuerdo con el compromiso de los requerimientos.

4. *Especificación de requerimientos* Los requerimientos se documentan e ingresan en la siguiente ronda de la espiral. Pueden producirse documentos de requerimientos formales o informales, como se estudia en la sección 4.3.

La figura 4.13 muestra que la adquisición y el análisis de requerimientos es un proceso iterativo con retroalimentación continua de cada actividad a otras actividades. El ciclo del proceso comienza con el descubrimiento de requerimientos y termina con la documentación de los requerimientos. La comprensión de los requerimientos por parte del analista mejora con cada ronda del ciclo. El ciclo concluye cuando está completo el documento de requerimientos.

La adquisición y la comprensión de los requerimientos por parte de los participantes del sistema es un proceso difícil por diferentes razones:

1. Los participantes con frecuencia no saben lo que quieren de un sistema de cómputo, excepto en términos muy generales; pueden encontrar difícil articular qué quieren que haga el sistema; pueden hacer peticiones inalcanzables porque no saben qué es factible y qué no lo es.
2. Los participantes en un sistema expresan naturalmente los requerimientos con sus términos y conocimientos implícitos de su trabajo. Los ingenieros de requerimientos, sin experiencia en el dominio del cliente, podrían no entender dichos requerimientos.
3. Diferentes participantes tienen distintos requerimientos y pueden expresarlos en variadas formas. Los ingenieros de requerimientos deben descubrir todas las fuentes potenciales de requerimientos e identificar similitudes y conflictos.
4. Factores políticos llegan a influir en los requerimientos de un sistema. Los administradores pueden solicitar requerimientos específicos del sistema, porque éstos les permitirán aumentar su influencia en la organización.
5. El ambiente económico y empresarial donde ocurre el análisis es dinámico. Inevitablemente cambia durante el proceso de análisis. Puede cambiar la importancia de requerimientos particulares; o bien, tal vez surjan nuevos requerimientos de nuevos participantes a quienes no se consultó originalmente.

Resulta ineludible que diferentes participantes tengan diversas visiones de la importancia y prioridad de los requerimientos y, algunas veces, dichas visiones están en conflicto. Durante el proceso, usted deberá organizar negociaciones regulares con los participantes, de forma que se alcancen compromisos. Es imposible complacer por completo a cada participante, pero, si algunos suponen que sus visiones no se consideraron de forma adecuada, quizás intenten deliberadamente socavar el proceso de IR.

En la etapa de especificación de requerimientos, los requerimientos adquiridos hasta el momento se documentan de tal forma que puedan usarse para ayudar al hallazgo de requerimientos. En esta etapa, podría generarse una primera versión del documento de requerimientos del sistema, con secciones faltantes y requerimientos incompletos. De modo alternativo, los requerimientos pueden documentarse en una forma completamente diferente (por ejemplo, en una hoja de cálculo o en tarjetas). Escribir requerimientos en tarjetas suele ser muy efectivo, ya que los participantes las administran, cambian y organizan con facilidad.

Puntos de vista

Un punto de vista es una forma de recopilar y organizar un conjunto de requerimientos de un grupo de participantes que cuentan con algo en común. Por lo tanto, cada punto de vista incluye una serie de requerimientos del sistema. Los puntos de vista pueden provenir de usuarios finales, administradores, etcétera. Ayudan a identificar a los individuos que brindan información sobre sus requerimientos y a estructurar los requerimientos para análisis.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

4.5.1 Descubrimiento de requerimientos

El descubrimiento de requerimientos (llamado a veces adquisición de requerimientos) es el proceso de recopilar información sobre el sistema requerido y los sistemas existentes, así como de separar, a partir de esta información, los requerimientos del usuario y del sistema. Las fuentes de información durante la fase de descubrimiento de requerimientos incluyen documentación, participantes del sistema y especificaciones de sistemas similares. La interacción con los participantes es a través de entrevistas y observaciones, y pueden usarse escenarios y prototipos para ayudar a los participantes a entender cómo será el sistema.

Los participantes varían desde administradores y usuarios finales de un sistema hasta participantes externos como los reguladores, quienes certifican la aceptabilidad del sistema. Por ejemplo, los participantes que se incluyen para el sistema de información de pacientes en atención a la salud mental son:

1. Pacientes cuya información se registra en el sistema.
2. Médicos que son responsables de valorar y tratar a los pacientes.
3. Enfermeros que coordinan, junto con los médicos, las consultas y suministran algunos tratamientos.
4. Recepcionistas que administran las citas médicas de los pacientes.
5. Personal de TI que es responsable de instalar y mantener el sistema.
6. Un director de ética médica que debe garantizar que el sistema cumpla con los lineamientos éticos actuales de la atención al paciente.
7. Encargados de atención a la salud que obtienen información administrativa del sistema.
8. Personal de archivo médico que es responsable de garantizar que la información del sistema se conserve, y se implementen de manera adecuada los procedimientos de mantenimiento del archivo.

Además de los participantes del sistema, se observa que los requerimientos también pueden venir del dominio de aplicación y de otros sistemas que interactúan con el sistema a especificar. Todos ellos deben considerarse durante el proceso de adquisición de requerimientos.

Todas estas diferentes fuentes de requerimientos (participantes, dominio, sistemas) se representan como puntos de vista del sistema, y cada visión muestra un subconjunto de los

requerimientos para el sistema. Diferentes puntos de vista de un problema enfocan el problema de diferentes formas. Sin embargo, sus perspectivas no son totalmente independientes, sino que por lo general se traslapan, de manera que tienen requerimientos comunes. Usted puede usar estos puntos de vista para estructurar tanto el descubrimiento como la documentación de los requerimientos del sistema.

4.5.2 Entrevistas

Las entrevistas formales o informales con participantes del sistema son una parte de la mayoría de los procesos de ingeniería de requerimientos. En estas entrevistas, el equipo de ingeniería de requerimientos formula preguntas a los participantes sobre el sistema que actualmente usan y el sistema que se va a desarrollar. Los requerimientos se derivan de las respuestas a dichas preguntas. Las entrevistas son de dos tipos:

1. Entrevistas cerradas, donde los participantes responden a un conjunto de preguntas preestablecidas.
2. Entrevistas abiertas, en las cuales no hay agenda predefinida. El equipo de ingeniería de requerimientos explora un rango de conflictos con los participantes del sistema y, como resultado, desarrolla una mejor comprensión de sus necesidades.

En la práctica, las entrevistas con los participantes son por lo general una combinación de ambas. Quizá se deba obtener la respuesta a ciertas preguntas, pero eso a menudo conduce a otros temas que se discuten en una forma menos estructurada. Rara vez funcionan bien las discusiones completamente abiertas. Con frecuencia debe plantear algunas preguntas para comenzar y mantener la entrevista enfocada en el sistema que se va a desarrollar.

Las entrevistas son valiosas para lograr una comprensión global sobre qué hacen los participantes, cómo pueden interactuar con el nuevo sistema y las dificultades que enfrentan con los sistemas actuales. A las personas les gusta hablar acerca de sus trabajos, así que por lo general están muy dispuestas a participar en entrevistas. Sin embargo, las entrevistas no son tan útiles para comprender los requerimientos desde el dominio de la aplicación.

Por dos razones resulta difícil asimilar el conocimiento de dominio a través de entrevistas:

1. Todos los especialistas en la aplicación usan terminología y jerga que son específicos de un dominio. Es imposible que ellos discutan los requerimientos de dominio sin usar este tipo de lenguaje. Por lo general, usan la terminología en una forma precisa y sutil, que para los ingenieros de requerimientos es fácil de malinterpretar.
2. Cierto conocimiento del dominio es tan familiar a los participantes que encuentran difícil de explicarlo, o bien, creen que es tan fundamental que no vale la pena mencionarlo. Por ejemplo, para un bibliotecario no es necesario decir que todas las adquisiciones deben catalogarse antes de agregarlas al acervo. Sin embargo, esto quizá no sea obvio para el entrevistador y, por lo tanto, es posible que no lo tome en cuenta en los requerimientos.

Las entrevistas tampoco son una técnica efectiva para adquirir conocimiento sobre los requerimientos y las restricciones de la organización, porque existen relaciones sutiles de poder entre los diferentes miembros en la organización. Las estructuras publicadas de

la organización rara vez coinciden con la realidad de la toma de decisiones en una organización, pero los entrevistados quizá no deseen revelar a un extraño la estructura real, sino la teórica. En general, la mayoría de las personas se muestran renuentes a discutir los conflictos políticos y organizacionales que afecten los requerimientos.

Los entrevistadores efectivos poseen dos características:

1. Tienen mentalidad abierta, evitan ideas preconcebidas sobre los requerimientos y escuchan a los participantes. Si el participante aparece con requerimientos sorprendentes, entonces tienen disposición para cambiar su mentalidad acerca del sistema.
2. Instan al entrevistado con una pregunta de trampolín para continuar la plática, dar una propuesta de requerimientos o trabajar juntos en un sistema de prototipo. Cuando se pregunta al individuo "dime qué quieres" es improbable que alguien consiga información útil. Encuentran mucho más sencillo hablar en un contexto definido que en términos generales.

La información de las entrevistas se complementa con otra información del sistema de documentación que describe los procesos empresariales o los sistemas existentes, las observaciones del usuario, etcétera. En ocasiones, además de los documentos del sistema, la información de la entrevista puede ser la única fuente de datos sobre los requerimientos del sistema. Sin embargo, la entrevista por sí misma está expuesta a perder información esencial y, por consiguiente, debe usarse junto con otras técnicas de adquisición de requerimientos.

4.5.3 Escenarios

Por lo general, las personas encuentran más sencillo vincularse con ejemplos reales que con descripciones abstractas. Pueden comprender y criticar un escenario sobre cómo interactuar con un sistema de software. Los ingenieros de requerimientos usan la información obtenida de esta discusión para formular los verdaderos requerimientos del sistema.

Los escenarios son particularmente útiles para detallar un bosquejo de descripción de requerimientos. Se trata de ejemplos sobre descripciones de sesiones de interacción. Cada escenario abarca comúnmente una interacción o un número pequeño de interacciones posibles. Se desarrollan diferentes formas de escenarios y se ofrecen varios tipos de información con diversos niveles de detalle acerca del sistema. Las historias que se usan en programación extrema, estudiadas en el capítulo 3, son un tipo de escenario de requerimientos.

Un escenario comienza con un bosquejo de la interacción. Durante el proceso de adquisición, se suman detalles a éste para crear una representación completa de dicha interacción. En su forma más general, un escenario puede incluir:

1. Una descripción de qué esperan el sistema y los usuarios cuando inicia el escenario.
2. Una descripción en el escenario del flujo normal de los eventos.
3. Una descripción de qué puede salir mal y cómo se manejaría.
4. Información de otras actividades que estén en marcha al mismo tiempo.
5. Una descripción del estado del sistema cuando termina el escenario.

SUPOSICIÓN INICIAL:
El paciente observa a un auxiliar médico que elabora un registro en el sistema y recaba información personal de aquél (nombre, dirección, edad, etcétera). Una enfermera ingresa en el sistema y obtiene la historia médica.

NORMAL:
La enfermera busca al paciente por su nombre completo. Si hay más de un paciente con el mismo apellido, para identificarlo se usa el nombre y la fecha de nacimiento.

La enfermera elige la opción de menú y añade la historia médica.

Inmediatamente la enfermera sigue una serie de indicadores (*prompt*) del sistema para ingresar información de consultas en otras instituciones, sobre problemas de salud mental (entrada libre de texto), condiciones médicas existentes (la enfermera selecciona las condiciones del menú), medicamentos administrados actualmente (seleccionados del menú), alergias (texto libre) y vida familiar (formato).

QUÉ PUEDE SALIR MAL:
Si no existe el registro del paciente o no puede encontrarse, la enfermera debe crear un nuevo registro e ingresar información personal.

Las condiciones o los medicamentos del paciente no se ingresan en el menú. La enfermera debe elegir la opción "otro" e ingresar texto libre que describa la condición/medicamento.

El paciente no puede/no proporciona información acerca de su historia médica. La enfermera tiene que ingresar a texto libre que registre la incapacidad/renuencia a brindar información. El sistema debe imprimir el formato de exclusión estándar que menciona que la falta de información podría significar que el tratamiento esté limitado o demorado. Esto tiene que firmarlo el paciente.

OTRAS ACTIVIDADES:
Mientras se ingresa la información, otros miembros del personal pueden consultar los registros, pero no editarlos.

ESTADO DEL SISTEMA A COMPLETAR:
Ingreso del usuario. El registro del paciente, incluida su historia médica, se integra en la base de datos, se agrega un registro a la bitácora (log) del sistema que indica el tiempo de inicio y terminación de la sesión y la enfermera a cargo.

Figura 4.14 Escenario para recabar historia médica en MHC-PMS

La adquisición basada en escenario implica trabajar con los participantes para identificar escenarios y captar detalles a incluir en dichos escenarios. Estos últimos pueden escribirse como texto, complementarse con diagramas, tomas de pantallas, etcétera. De forma alternativa, es posible usar un enfoque más estructurado, como los escenarios de evento o casos de uso.

Como ejemplo de un simple escenario de texto, considere cómo usaría el MHC-PMS para ingresar datos de un nuevo paciente (figura 4.14). Cuando un nuevo paciente asiste a una clínica, un auxiliar médico crea un nuevo registro y agrega información personal (nombre, edad, etcétera). Después, una enfermera entrevista al paciente y recaba su historia médica. Luego, el paciente tiene una consulta inicial con un médico que lo diagnostica y, si es adecuado, recomienda un tratamiento. El escenario muestra lo que sucede cuando se recaba la historia médica.

4.5.4 Casos de uso

Los casos de uso son una técnica de descubrimiento de requerimientos que se introdujo por primera vez en el método Objectory (Jacobson *et al*., 1993). Ahora se ha convertido

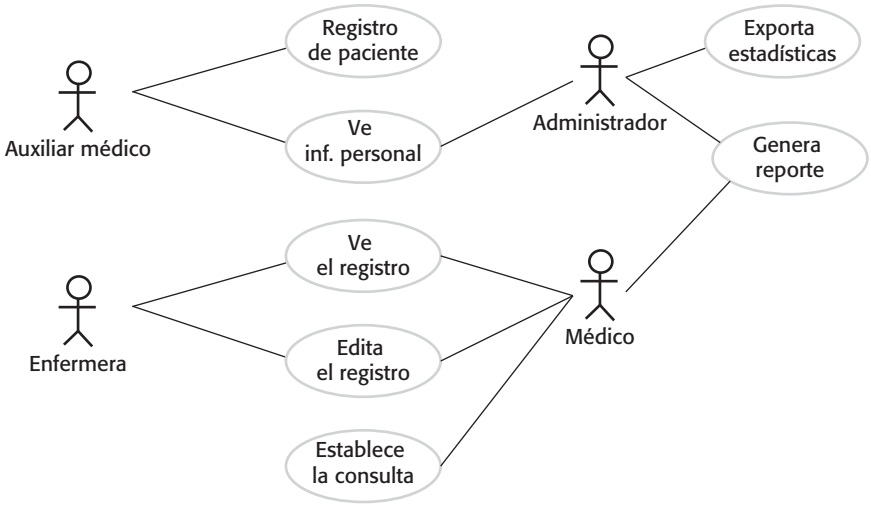


Figura 4.15 Casos de uso para el MHC-PMS

en una característica fundamental del modelado de lenguaje unificado. En su forma más sencilla, un caso de uso identifica a los actores implicados en una interacción, y nombra el tipo de interacción. Entonces, esto se complementa con información adicional que describe la interacción con el sistema. La información adicional puede ser una descripción textual, o bien, uno o más modelos gráficos como una secuencia UML o un gráfico de estado.

Los casos de uso se documentan con el empleo de un diagrama de caso de uso de alto nivel. El conjunto de casos de uso representa todas las interacciones posibles que se describirán en los requerimientos del sistema. Los actores en el proceso, que pueden ser individuos u otros sistemas, se representan como figuras sencillas. Cada clase de interacción se constituye como una elipse con etiqueta. Líneas vinculan a los actores con la interacción. De manera opcional, se agregan puntas de flecha a las líneas para mostrar cómo se inicia la interacción. Esto se ilustra en la figura 4.15, que presenta algunos de los casos de uso para el sistema de información del paciente.

No hay distinción tajante y rápida entre escenarios y casos de uso. Algunas personas consideran que cada caso de uso es un solo escenario; otras, como sugieren Stevens y Pooley (2006), encapsulan un conjunto de escenarios en un solo caso de uso. Cada escenario es un solo hilo a través del caso de uso. Por lo tanto, habría un escenario para la interacción normal, más escenarios para cada posible excepción. En la práctica, es posible usarlos en cualquier forma.

Los casos de uso identifican las interacciones individuales entre el sistema y sus usuarios u otros sistemas. Cada caso de uso debe documentarse con una descripción textual. Entonces pueden vincularse con otros modelos en el UML que desarrollará el escenario con más detalle. Por ejemplo, una breve descripción del caso de uso *Establece la consulta* de la figura 4.15 sería:

> *El establecimiento de consulta permite que dos o más médicos, que trabajan en diferentes consultorios, vean el mismo registro simultáneamente. Un médico inicia la consulta al elegir al individuo involucrado de un menú desplegable de médicos que estén en línea. Entonces el registro del paciente se despliega en sus pantallas,*

pero sólo el médico que inicia puede editar el registro. Además, se crea una ventana de chat de texto para ayudar a coordinar las acciones. Se supone que, de manera separada, se establecerá una conferencia telefónica para comunicación por voz.

Los escenarios y los casos de uso son técnicas efectivas para adquirir requerimientos de los participantes que interactúan directamente con el sistema. Cada tipo de interacción puede representarse como caso de uso. Sin embargo, debido a que se enfocan en interacciones con el sistema, no son tan efectivas para adquirir restricciones o requerimientos empresariales y no funcionales de alto nivel, ni para descubrir requerimientos de dominio.

El UML es un estándar *de facto* para modelado orientado a objetos, así que los casos de uso y la adquisición basada en casos ahora se utilizan ampliamente para adquisición de requerimientos. Los casos de uso se estudian en el capítulo 5, y se muestra cómo se emplean junto con otros modelos del sistema para documentar el diseño de un sistema.

4.5.5 Etnografía

Los sistemas de software no existen aislados. Se usan en un contexto social y organizacional, y dicho escenario podría derivar o restringir los requerimientos del sistema de software. A menudo satisfacer dichos requerimientos sociales y organizacionales es crítico para el éxito del sistema. Una razón por la que muchos sistemas de software se entregan, y nunca se utilizan, es que sus requerimientos no consideran de manera adecuada cómo afectaría el contexto social y organizacional la operación práctica del sistema.

La etnografía es una técnica de observación que se usa para entender los procesos operacionales y ayudar a derivar requerimientos de apoyo para dichos procesos. Un analista se adentra en el ambiente laboral donde se usará el sistema. Observa el trabajo diario y toma notas acerca de las tareas existentes en que intervienen los participantes. El valor de la etnografía es que ayuda a descubrir requerimientos implícitos del sistema que reflejan las formas actuales en que trabaja la gente, en vez de los procesos formales definidos por la organización.

Las personas con frecuencia encuentran muy difícil articular los detalles de su trabajo, porque es una segunda forma de vida para ellas. Entienden su trabajo, pero tal vez no su relación con otras funciones en la organización. Los factores sociales y organizacionales que afectan el trabajo, que no son evidentes para los individuos, sólo se vuelven claros cuando los percibe un observador sin prejuicios. Por ejemplo, un grupo de trabajo puede organizarse de modo que sus miembros conozcan el trabajo de los demás y se suplan entre sí cuando alguien se ausenta. Es probable que esto no se mencione durante una entrevista, pues el grupo podría no verlo como una parte integral de su función.

Suchman (1987) fue una de las primeras en usar la etnografía para estudiar el trabajo en la oficina. Ella descubrió que las prácticas reales del trabajo son más ricas, más complejas y más dinámicas que los modelos simples supuestos por los sistemas de automatización administrativa. La diferencia entre el trabajo supuesto y el real fue la razón más importante por la que dichos sistemas de oficina no tenían un efecto significativo sobre la productividad. Crabtree (2003) analiza desde entonces una amplia gama de estudios, y describe, en

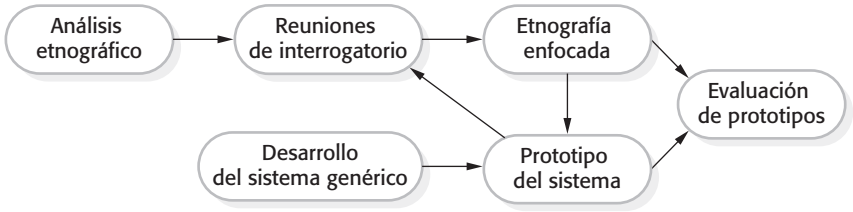


Figura 4.16 Etnografía y creación de prototipos para análisis de requerimientos

general, el uso de la etnografía en el diseño de sistemas. El autor ha investigado métodos para integrar la etnografía en el proceso de ingeniería de software, mediante su vinculación con los métodos de la ingeniería de requerimientos (Viller y Sommerville, 1999; Viller y Sommerville, 2000) y patrones para documentar la interacción en sistemas cooperativos (Martin *et al*., 2001; Martin *et al*., 2002; Martin y Sommerville, 2004).

La etnografía es muy efectiva para descubrir dos tipos de requerimientos:

1. Requerimientos que se derivan de la forma en que realmente trabaja la gente, en vez de la forma en la cual las definiciones del proceso indican que debería trabajar. Por ejemplo, los controladores de tráfico aéreo pueden desactivar un sistema de alerta de conflicto que detecte una aeronave con trayectoria de vuelo que se cruza, aun cuando los procedimientos de control normales especifiquen que es obligatorio usar tal sistema. Ellos deliberadamente dejan a la aeronave sobre la ruta de conflicto durante breves momentos, para ayudarse a dirigir el espacio aéreo. Su estrategia de control está diseñada para garantizar que dichas aeronaves se desvíen antes de que haya problemas, y consideran que la alarma de alerta de conflicto los distrae de su trabajo.

2. Requerimientos que se derivan de la cooperación y el conocimiento de las actividades de otras personas. Por ejemplo, los controladores de tráfico aéreo pueden usar el conocimiento del trabajo de otros controladores para predecir el número de aeronaves que entrarán a su sector de control. Entonces, modifican sus estrategias de control dependiendo de dicha carga de trabajo prevista. Por lo tanto, un sistema ATC automatizado debería permitir a los controladores en un sector tener cierta visibilidad del trabajo en sectores adyacentes.

La etnografía puede combinarse con la creación de prototipos (figura 4.16). La etnografía informa del desarrollo del prototipo, de modo que se requieren menos ciclos de refinamiento del prototipo. Más aún, la creación de prototipos se enfoca en la etnografía al identificar problemas y preguntas que entonces pueden discutirse con el etnógrafo. Siendo así, éste debe buscar las respuestas a dichas preguntas durante la siguiente fase de estudio del sistema (Sommerville *et al*., 1993).

Los estudios etnográficos pueden revelar detalles críticos de procesos, que con frecuencia se pierden con otras técnicas de adquisición de requerimientos. Sin embargo, debido a su enfoque en el usuario final, no siempre es adecuado para descubrir requerimientos de la organización o de dominio. No en todos los casos se identifican nuevas características que deben agregarse a un sistema. En consecuencia, la etnografía no es un enfoque completo para la adquisición por sí misma, y debe usarse para complementar otros enfoques, como el análisis de casos de uso.

Revisiones de requerimientos

Una revisión de requerimientos es un proceso donde un grupo de personas del cliente del sistema y el desarrollador del sistema leen con detalle el documento de requerimientos y buscan errores, anomalías e inconsistencias. Una vez detectados y registrados, recae en el cliente y el desarrollador la labor de negociar cómo resolver los problemas identificados.

http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html

4.6 Validación de requerimientos

La validación de requerimientos es el proceso de verificar que los requerimientos definan realmente el sistema que en verdad quiere el cliente. Se traslapa con el análisis, ya que se interesa por encontrar problemas con los requerimientos. La validación de requerimientos es importante porque los errores en un documento de requerimientos pueden conducir a grandes costos por tener que rehacer, cuando dichos problemas se descubren durante el desarrollo del sistema o después de que éste se halla en servicio.

En general, el costo por corregir un problema de requerimientos al hacer un cambio en el sistema es mucho mayor que reparar los errores de diseño o codificación. La razón es que un cambio a los requerimientos significa generalmente que también deben cambiar el diseño y la implementación del sistema. Más aún, el sistema debe entonces ponerse a prueba de nuevo.

Durante el proceso de validación de requerimientos, tienen que realizarse diferentes tipos de comprobaciones sobre los requerimientos contenidos en el documento de requerimientos. Dichas comprobaciones incluyen:

1. *Comprobaciones de validez* Un usuario quizá crea que necesita un sistema para realizar ciertas funciones. Sin embargo, con mayor consideración y análisis se logra identificar las funciones adicionales o diferentes que se requieran. Los sistemas tienen diversos participantes con diferentes necesidades, y cualquier conjunto de requerimientos es inevitablemente un compromiso a través de la comunidad de participantes.
2. *Comprobaciones de consistencia* Los requerimientos en el documento no deben estar en conflicto. Esto es, no debe haber restricciones contradictorias o descripciones diferentes de la misma función del sistema.
3. *Comprobaciones de totalidad* El documento de requerimientos debe incluir requerimientos que definan todas las funciones y las restricciones pretendidas por el usuario del sistema.
4. *Comprobaciones de realismo* Al usar el conocimiento de la tecnología existente, los requerimientos deben comprobarse para garantizar que en realidad pueden implementarse. Dichas comprobaciones también tienen que considerar el presupuesto y la fecha para el desarrollo del sistema.
5. *Verificabilidad* Para reducir el potencial de disputas entre cliente y contratista, los requerimientos del sistema deben escribirse siempre de manera que sean verificables. Esto significa que usted debe ser capaz de escribir un conjunto de pruebas que demuestren que el sistema entregado cumpla cada requerimiento especificado.

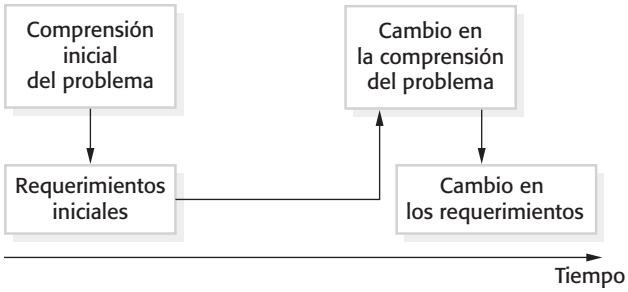


Figura 4.17
Evolución de los requerimientos

Hay algunas técnicas de validación de requerimientos que se usan individualmente o en conjunto con otras:

1. *Revisiones de requerimientos* Los requerimientos se analizan sistemáticamente usando un equipo de revisores que verifican errores e inconsistencias.
2. *Creación de prototipos* En esta aproximación a la validación, se muestra un modelo ejecutable del sistema en cuestión a los usuarios finales y clientes. Así, ellos podrán experimentar con este modelo para constatar si cubre sus necesidades reales.
3. *Generación de casos de prueba* Los requerimientos deben ser comprobables. Si las pruebas para los requerimientos se diseñan como parte del proceso de validación, esto revela con frecuencia problemas en los requerimientos. Si una prueba es difícil o imposible de diseñar, esto generalmente significa que los requerimientos serán difíciles de implementar, por lo que deberían reconsiderarse. El desarrollo de pruebas a partir de los requerimientos del usuario antes de escribir cualquier código es una pieza integral de la programación extrema.

No hay que subestimar los problemas incluidos en la validación de requerimientos. A final de cuentas, es difícil demostrar que un conjunto de requerimientos, de hecho, no cubre las necesidades de los usuarios. Estos últimos necesitan una imagen del sistema en operación, así como comprender la forma en que dicho sistema se ajustará a su trabajo. Es difícil, incluso para profesionales de la computación experimentados, realizar este tipo de análisis abstracto, y más aún para los usuarios del sistema. Como resultado, rara vez usted encontrará todos los problemas de requerimientos durante el proceso de validación de requerimientos. Es inevitable que haya más cambios en los requerimientos para corregir omisiones y malas interpretaciones, después de acordar el documento de requerimientos.

4.7 Administración de requerimientos

Los requerimientos para los grandes sistemas de software siempre cambian. Una razón es que dichos sistemas se desarrollaron por lo general para resolver problemas "horrorosos": aquellos problemas que no se pueden definir por completo. Como el problema no se logra definir por completo, los requerimientos del software están condenados también a estar incompletos. Durante el proceso de software, la comprensión que los participantes tienen de los problemas cambia constantemente (figura 4.17). Entonces, los requerimientos del sistema también deben evolucionar para reflejar esa visión cambiante del problema.

Requerimientos duraderos y volátiles

Algunos requerimientos son más susceptibles a cambiar que otros. Los requerimientos duraderos son los requerimientos que se asocian con las actividades centrales, de lento cambio, de una organización. También estos requerimientos se relacionan con actividades laborales fundamentales. Por el contrario, los requerimientos volátiles tienen más probabilidad de cambio. Se asocian por lo general con actividades de apoyo que reflejan cómo la organización hace su trabajo más que el trabajo en sí.

http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html

Una vez que se instala un sistema, y se utiliza con regularidad, surgirán inevitablemente nuevos requerimientos. Es difícil que los usuarios y clientes del sistema anticipen qué efectos tendrá el nuevo sistema sobre sus procesos de negocios y la forma en que se hace el trabajo. Una vez que los usuarios finales experimentan el sistema, descubrirán nuevas necesidades y prioridades. Existen muchas razones por las que es inevitable el cambio:

1. Los ambientes empresarial y técnico del sistema siempre cambian después de la instalación. Puede introducirse nuevo hardware, y quizá sea necesario poner en interfaz el sistema con otros sistemas, cambiar las prioridades de la empresa (con los consecuentes cambios en el sistema de apoyo requerido) e introducir nuevas leyes y regulaciones que el sistema deba cumplir cabalmente.
2. Los individuos que pagan por un sistema y los usuarios de dicho sistema, por lo general, no son los mismos. Los clientes del sistema imponen requerimientos debido a restricciones organizativas y presupuestales. Esto podría estar en conflicto con los requerimientos del usuario final y, después de la entrega, probablemente deban agregarse nuevas características para apoyar al usuario, si el sistema debe cubrir sus metas.
3. Los sistemas grandes tienen regularmente una comunidad de usuarios diversa, en la cual muchos individuos tienen diferentes requerimientos y prioridades que quizás estén en conflicto o sean contradictorios. Los requerimientos finales del sistema inevitablemente tienen un compromiso entre sí y, con la experiencia, a menudo se descubre que el equilibrio de apoyo brindado a diferentes usuarios tiene que cambiar.

La administración de requerimientos es el proceso de comprender y controlar los cambios en los requerimientos del sistema. Es necesario seguir la pista de requerimientos individuales y mantener los vínculos entre los requerimientos dependientes, de manera que pueda valorarse el efecto del cambio en los requerimientos. También es preciso establecer un proceso formal para hacer cambios a las propuestas y vincular éstos con los requerimientos del sistema. El proceso formal de la administración de requerimientos debe comenzar tan pronto como esté disponible un borrador del documento de requerimientos. Sin embargo, hay que empezar a planear cómo administrar el cambio en los requerimientos durante el proceso de adquisición de los mismos.

4.7.1 Planeación de la administración de requerimientos

La planeación es una primera etapa esencial en el proceso de administración de requerimientos. Esta etapa establece el nivel de detalle que se requiere en la administración de requerimientos. Durante la etapa de administración de requerimientos, usted tiene que decidir sobre:

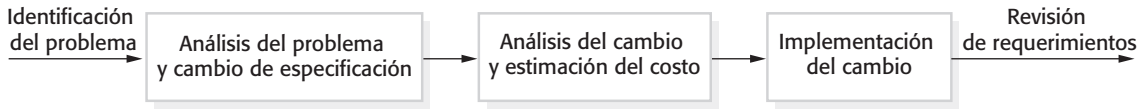


Figura 4.18 Administración del cambio de requerimientos

1. *Identificación de requerimientos* Cada requerimiento debe identificarse de manera exclusiva, de forma que pueda tener referencia cruzada con otros requerimientos y usarse en las evaluaciones de seguimiento.
2. *Un proceso de administración del cambio* Éste es el conjunto de actividades que valoran el efecto y costo de los cambios. En la siguiente sección se estudia con más detalle este proceso.
3. *Políticas de seguimiento* Dichas políticas definen las relaciones entre cada requerimiento, así como entre los requerimientos y el diseño del sistema que debe registrarse. La política de seguimiento también tiene que definir cómo mantener dichos registros.
4. *Herramientas de apoyo* La administración de requerimientos incluye el procesamiento de grandes cantidades de información acerca de los requerimientos. Las herramientas disponibles varían desde sistemas especializados de administración de requerimientos, hasta hojas de cálculo y sistemas de bases de datos simples.

La administración de requerimientos necesita apoyo automatizado y herramientas de software, para lo cual deben seleccionarse durante la fase de planeación. Se necesitan herramientas de apoyo para:

1. *Almacenamiento de requerimientos* Los requerimientos tienen que mantenerse en un almacén de datos administrado y seguro, que sea accesible para todos quienes intervienen en el proceso de ingeniería de requerimientos.
2. *Administración del cambio* El proceso de administración del cambio (figura 4.18) se simplifica si está disponible la herramienta de apoyo activa.
3. *Administración del seguimiento* Como se estudió anteriormente, la herramienta de apoyo para el seguimiento permite la identificación de requerimientos relacionados. Algunas herramientas que están disponibles usan técnicas de procesamiento en lenguaje natural, para ayudar a descubrir posibles relaciones entre los requerimientos.

Para sistemas pequeños, quizá no sea necesario usar herramientas especializadas de administración de requerimientos. El proceso de administración de requerimientos puede apoyarse con el uso de funciones disponibles en procesadores de texto, hojas de cálculo y bases de datos de PC. Sin embargo, para sistemas más grandes se requieren herramientas de apoyo más especializadas. En las páginas Web del libro se incluyen vínculos a información acerca de herramientas de administración de requerimientos.

4.7.2 Administración del cambio en los requerimientos

La administración del cambio en los requerimientos (figura 4.18) debe aplicarse a todos los cambios propuestos a los requerimientos de un sistema, después de aprobarse el documento de requerimientos. La administración del cambio es esencial porque es necesario determinar si los beneficios de implementar nuevos requerimientos están justificados por

Seguimiento de requerimientos

Es necesario seguir la huella de las relaciones entre requerimientos, sus fuentes y el diseño del sistema, de modo que usted pueda analizar las razones para los cambios propuestos, así como el efecto que dichos cambios tengan probablemente sobre otras partes del sistema. Es necesario poder seguir la pista de cómo un cambio se propaga hacia el sistema. ¿Por qué?

http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html

los costos de la implementación. La ventaja de usar un proceso formal para la administración del cambio es que todas las propuestas de cambio se tratan de manera consistente y los cambios al documento de requerimientos se realizan en una forma controlada.

Existen tres etapas principales de un proceso de administración del cambio:

1. *Análisis del problema y especificación del cambio* El proceso comienza con la identificación de un problema en los requerimientos o, en ocasiones, con una propuesta de cambio específica. Durante esta etapa, el problema o la propuesta de cambio se analizan para comprobar que es válida. Este análisis retroalimenta al solicitante del cambio, quien responderá con una propuesta de cambio de requerimientos más específica, o decidirá retirar la petición.

2. *Análisis del cambio y estimación del costo* El efecto del cambio propuesto se valora usando información de seguimiento y conocimiento general de los requerimientos del sistema. El costo por realizar el cambio se estima en términos de modificaciones al documento de requerimientos y, si es adecuado, al diseño y la implementación del sistema. Una vez completado este análisis, se toma una decisión acerca de si se procede o no con el cambio de requerimientos.

3. *Implementación del cambio* Se modifican el documento de requerimientos y, donde sea necesario, el diseño y la implementación del sistema. Hay que organizar el documento de requerimientos de forma que sea posible realizar cambios sin reescritura o reorganización extensos. Conforme a los programas, la variabilidad en los documentos se logra al minimizar las referencias externas y al hacer las secciones del documento tan modulares como sea posible. De esta manera, secciones individuales pueden modificarse y sustituirse sin afectar otras partes del documento.

Si un nuevo requerimiento tiene que implementarse urgentemente, siempre existe la tentación para cambiar el sistema y luego modificar de manera retrospectiva el documento de requerimientos. Hay que tratar de evitar esto, pues casi siempre conducirá a que la especificación de requerimientos y la implementación del sistema se salgan de ritmo. Una vez realizados los cambios al sistema, es fácil olvidar la inclusión de dichos cambios en el documento de requerimientos, o bien, agregar información al documento de requerimientos que sea inconsistente con la implementación.

Los procesos de desarrollo ágil, como la programación extrema, se diseñaron para enfrentar los requerimientos que cambian durante el proceso de desarrollo. En dichos procesos, cuando un usuario propone un cambio de requerimientos, éste no pasa por un proceso de administración del cambio formal. En vez de ello, el usuario tiene que priorizar dicho cambio y, si es de alta prioridad, decidir qué características del sistema planeadas para la siguiente iteración pueden eliminarse.

PUNTOS CLAVE

- Los requerimientos para un sistema de software establecen lo que debe hacer el sistema y definen las restricciones sobre su operación e implementación.
- Los requerimientos funcionales son enunciados de los servicios que debe proporcionar el sistema, o descripciones de cómo deben realizarse algunos cálculos.
- Los requerimientos no funcionales restringen con frecuencia el sistema que se va a desarrollar y el proceso de desarrollo a usar. Éstos pueden ser requerimientos del producto, requerimientos organizacionales o requerimientos externos. A menudo se relacionan con las propiedades emergentes del sistema y, por lo tanto, se aplican al sistema en su conjunto.
- El documento de requerimientos de software es un enunciado acordado sobre los requerimientos del sistema. Debe organizarse de forma que puedan usarlo tanto los clientes del sistema como los desarrolladores del software.
- El proceso de ingeniería de requerimientos incluye un estudio de factibilidad, adquisición y análisis de requerimientos, especificación de requerimientos, validación de requerimientos y administración de requerimientos.
- La adquisición y el análisis de requerimientos es un proceso iterativo que se representa como una espiral de actividades: descubrimiento de requerimientos, clasificación y organización de requerimientos, negociación de requerimientos y documentación de requerimientos.
- La validación de requerimientos es el proceso de comprobar la validez, la consistencia, la totalidad, el realismo y la verificabilidad de los requerimientos.
- Los cambios empresariales, organizacionales y técnicos conducen inevitablemente a cambios en los requerimientos para un sistema de software. La administración de requerimientos es el proceso de gestionar y controlar dichos cambios.

LECTURAS SUGERIDAS

Software Requirements, 2nd edition. Este libro, diseñado para escritores y usuarios de requerimientos, analiza las buenas prácticas en la ingeniería de requerimientos. (K. M. Weigers, 2003, Microsoft Press.)

"Integrated requirements engineering: A tutorial". Se trata de un ensayo de tutoría en el que se analizan las actividades de la ingeniería de requerimientos y cómo pueden adaptarse para ajustarse a las prácticas modernas de la ingeniería de software. (I. Sommerville, IEEE Software, 22(1), Jan–Feb 2005.) http://dx.doi.org/10.1109/MS.2005.13.

Mastering the Requirements Process, 2nd edition. Un libro bien escrito, fácil de leer, que se basa en un método particular (VOLERE), pero que también incluye múltiples buenos consejos generales acerca de la ingeniería de requerimientos. (S. Robertson y J. Robertson, 2006, Addison-Wesley.)

"Research Directions in Requirements Engineering". Un buen estudio de la investigación en ingeniería de requerimientos que destaca los futuros retos en la investigación en el área, con la finalidad de enfrentar conflictos como la escala y la agilidad. (B. H. C. Cheng y J. M. Atlee, Proc. Conf on Future of Software Engineering, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.17.

EJERCICIOS

4.1. Identifique y describa brevemente cuatro tipos de requerimientos que puedan definirse para un sistema basado en computadora.

4.2. Descubra las ambigüedades u omisiones en el siguiente enunciado de requerimientos de un sistema de emisión de boletos:

Un sistema automatizado de emisión de boletos vende boletos de ferrocarril. Los usuarios seleccionan su destino e ingresan un número de tarjeta de crédito y uno de identificación personal. El boleto de ferrocarril se emite y se carga en su cuenta de tarjeta de crédito. Cuando el usuario oprime el botón *start*, se activa una pantalla de menú con los posibles destinos, junto con un mensaje que pide al usuario seleccionar un destino. Una vez seleccionado el destino, se solicita a los usuarios ingresar su tarjeta de crédito. Se comprueba su validez y luego se pide al usuario ingresar un identificador personal. Cuando se valida la transacción crediticia, se emite el boleto.

4.3. Vuelva a escribir la descripción anterior usando el enfoque estructurado referido en este capítulo. Resuelva las ambigüedades identificadas de forma adecuada.

4.4. Escriba un conjunto de requerimientos no funcionales para el sistema de emisión de boletos, y establezca su fiabilidad y tiempo de respuesta esperados.

4.5. Con la técnica aquí sugerida, en que las descripciones en lenguaje natural se presentan en formato estándar, escriba requerimientos de usuario plausibles para las siguientes funciones:

- Un sistema de bombeo de petróleo (gasolina) no asistido que incluya un lector de tarjeta de crédito. El cliente pasa la tarjeta en el lector, luego especifica la cantidad de combustible requerido. Se suministra el combustible y se deduce de la cuenta del cliente.
- La función de dispensar efectivo en un cajero automático.
- La función de revisión y corrección ortográfica en un procesador de textos.

4.6. Sugiera cómo un ingeniero responsable de redactar una especificación de requerimientos de sistema puede seguir la huella de las relaciones entre requerimientos funcionales y no funcionales.

4.7. Con su conocimiento de cómo se usa un cajero automático, desarrolle un conjunto de casos de uso que pudieran servir como base para comprender los requerimientos para el sistema de un cajero automático.

4.8. ¿Quién debería involucrarse en una revisión de requerimientos? Dibuje un modelo del proceso que muestre cómo podría organizarse una revisión de requerimientos.

4.9. Cuando tienen que hacerse cambios de emergencia a los sistemas, es posible que deba modificarse el software del sistema antes de aprobar los cambios a los requerimientos. Sugiera un modelo de un proceso para realizar dichas modificaciones, que garantice que el documento de requerimientos y la implementación del sistema no serán inconsistentes.

4.10. Usted acepta un empleo con un usuario de software, quien contrató a su empleador anterior con la finalidad de desarrollar un sistema para ellos. Usted descubre que la interpretación de los requerimientos de su compañía es diferente de la interpretación tomada por su antiguo empleador. Discuta qué haría en tal situación. Usted sabe que los costos para su actual empleador aumentarán si no se resuelven las ambigüedades. Sin embargo, también tiene una responsabilidad de confidencialidad con su empleador anterior.

REFERENCIAS

Beck, K. (1999). "Embracing Change with Extreme Programming". *IEEE Computer,* **32** (10), 70–8.

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography.* London: Springer-Verlag.

Davis, A. M. (1993). *Software Requirements: Objects, Functions and States.* Englewood Cliffs, NJ: Prentice Hall.

IEEE. (1998). "IEEE Recommended Practice for Software Requirements Specifications". En *IEEE Software Engineering Standards Collection.* Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. y Overgaard, G. (1993). *Object-Oriented Software Engineering.* Wokingham: Addison-Wesley.

Kotonya, G. y Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* Chichester, UK: John Wiley and Sons.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process.* Englewood Cliff, NJ: Prentice Hall.

Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. y Viller, S. (2001). "Finding Patterns in the Fieldwork". *Proc. ECSCW' 01.* Bonn: Kluwer. 39–58.

Martin, D., Rouncefield, M. y Sommerville, I. (2002). "Applying patterns of interaction to work (re) design: E-government and planning". *Proc. ACM CHI' 2002,* ACM Press. 235–42.

Martin, D. y Sommerville, I. (2004). "Patterns of interaction: Linking ethnomethodology and design". *ACM Trans. on Computer-Human Interaction,* **11** (1), 59–89.

Robertson, S. y Robertson, J. (1999). *Mastering the Requirements Process.* Harlow, UK: Addison-Wesley.

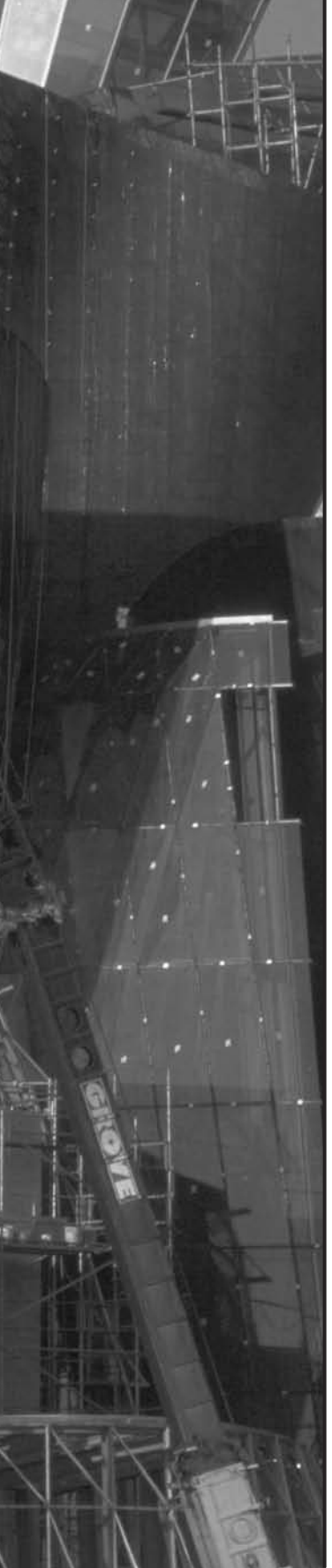
Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. y Twidale, M. (1993). "Integrating ethnography into the requirements engineering process". *Proc. RE' 93,* San Diego CA.: IEEE Computer Society Press. 165–73.

Stevens, P. y Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison Wesley.

Suchman, L. (1987). *Plans and Situated Actions.* Cambridge: Cambridge University Press.

Viller, S. y Sommerville, I. (1999). "Coherence: An Approach to Representing Ethnographic Analyses in Systems Design". *Human-Computer Interaction,* **14** (1 & 2), 9–41.

Viller, S. y Sommerville, I. (2000). "Ethnographically informed analysis for software engineers". *Int. J. of Human-Computer Studies,* **53** (1), 169–96.



5

Modelado del sistema

Objetivos

El objetivo de este capítulo es introducir algunos tipos de modelo de sistema que pueden desarrollarse, como parte de la ingeniería de requerimientos y los procesos de diseño del sistema. Al estudiar este capítulo:

- comprenderá cómo usar los modelos gráficos para representar los sistemas de software;
- entenderá por qué se requieren diferentes tipos de modelo, así como las perspectivas fundamentales de contexto, interacción, estructura y comportamiento del modelado de sistemas;
- accederá a algunos de los tipos de diagrama en el Lenguaje de Modelado Unificado (UML) y conocerá cómo se utilizan dichos diagramas en el modelado del sistema;
- estará al tanto de las ideas que subyacen en la ingeniería dirigida por modelo, donde un sistema se genera automáticamente a partir de modelos estructurales y de comportamiento.

Contenido

5.1 Modelos de contexto
5.2 Modelos de interacción
5.3 Modelos estructurales
5.4 Modelos de comportamiento
5.5 Ingeniería dirigida por modelo

El modelado de sistemas es el proceso para desarrollar modelos abstractos de un sistema, donde cada modelo presenta una visión o perspectiva diferente de dicho sistema. En general, el modelado de sistemas se ha convertido en un medio para representar el sistema usando algún tipo de notación gráfica, que ahora casi siempre se basa en notaciones en el Lenguaje de Modelado Unificado (UML). Sin embargo, también es posible desarrollar modelos formales (matemáticos) de un sistema, generalmente como una especificación detallada del sistema. En este capítulo se estudia el modelado gráfico utilizando el UML, y en el capítulo 12, el modelado formal.

Los modelos se usan durante el proceso de ingeniería de requerimientos para ayudar a derivar los requerimientos de un sistema, durante el proceso de diseño para describir el sistema a los ingenieros que implementan el sistema, y después de la implementación para documentar la estructura y la operación del sistema. Es posible desarrollar modelos tanto del sistema existente como del sistema a diseñar:

1. Los modelos del sistema existente se usan durante la ingeniería de requerimientos. Ayudan a aclarar lo que hace el sistema existente y pueden utilizarse como base para discutir sus fortalezas y debilidades. Posteriormente, conducen a los requerimientos para el nuevo sistema.
2. Los modelos del sistema nuevo se emplean durante la ingeniería de requerimientos para ayudar a explicar los requerimientos propuestos a otros participantes del sistema. Los ingenieros usan tales modelos para discutir las propuestas de diseño y documentar el sistema para la implementación. En un proceso de ingeniería dirigido por modelo, es posible generar una implementación de sistema completa o parcial a partir del modelo del sistema.

El aspecto más importante de un modelo del sistema es que deja fuera los detalles. Un modelo es una abstracción del sistema a estudiar, y no una representación alternativa de dicho sistema. De manera ideal, una representación de un sistema debe mantener toda la información sobre la entidad a representar. Una abstracción simplifica y recoge deliberadamente las características más destacadas. Por ejemplo, en el muy improbable caso de que este libro se entregue por capítulos en un periódico, la presentación sería una abstracción de los puntos clave del libro. Si se tradujera del inglés al italiano, sería una representación alternativa. La intención del traductor sería mantener toda la información como se presenta en inglés.

Desde diferentes perspectivas, usted puede desarrollar diferentes modelos para representar el sistema. Por ejemplo:

1. Una perspectiva externa, donde se modelen el contexto o entorno del sistema.
2. Una perspectiva de interacción, donde se modele la interacción entre un sistema y su entorno, o entre los componentes de un sistema.
3. Una perspectiva estructural, donde se modelen la organización de un sistema o la estructura de datos que procese el sistema.
4. Una perspectiva de comportamiento, donde se modele el comportamiento dinámico del sistema y cómo responde ante ciertos eventos.

Estas perspectivas tienen mucho en común con la visión 4 + 1 de arquitectura del sistema de Kruchten (Kruchten, 1995), la cual sugiere que la arquitectura y la organización de un sistema deben documentarse desde diferentes perspectivas. En el capítulo 6 se estudia este enfoque 4 + 1.

En este capítulo se usan diagramas definidos en UML (Booch *et al*., 2005; Rumbaugh *et al*., 2004), que se han convertido en un lenguaje de modelado estándar para modelado orientado a objetos. El UML tiene numerosos tipos de diagramas y, por lo tanto, soporta la creación de muchos diferentes tipos de modelo de sistema. Sin embargo, un estudio en 2007 (Erickson y Siau, 2007) mostró que la mayoría de los usuarios del UML consideraban que cinco tipos de diagrama podrían representar lo esencial de un sistema.

1. Diagramas de actividad, que muestran las actividades incluidas en un proceso o en el procesamiento de datos.
2. Diagramas de caso de uso, que exponen las interacciones entre un sistema y su entorno.
3. Diagramas de secuencias, que muestran las interacciones entre los actores y el sistema, y entre los componentes del sistema.
4. Diagramas de clase, que revelan las clases de objeto en el sistema y las asociaciones entre estas clases.
5. Diagramas de estado, que explican cómo reacciona el sistema frente a eventos internos y externos.

Como aquí no hay espacio para discutir todos los tipos de diagramas UML, el enfoque se centrará en cómo estos cinco tipos clave de diagramas se usan en el modelado del sistema.

Cuando desarrolle modelos de sistema, sea flexible en la forma en que use la notación gráfica. No siempre necesitará apegarse rigurosamente a los detalles de una notación. El detalle y el rigor de un modelo dependen de cómo lo use. Hay tres formas en que los modelos gráficos se emplean con frecuencia:

1. Como medio para facilitar la discusión sobre un sistema existente o propuesto.
2. Como una forma de documentar un sistema existente.
3. Como una descripción detallada del sistema que sirve para generar una implementación de sistema.

En el primer caso, el propósito del modelo es estimular la discusión entre los ingenieros de software que intervienen en el desarrollo del sistema. Los modelos pueden ser incompletos (siempre que cubran los puntos clave de la discusión) y utilizar de manera informal la notación de modelado. Así es como se utilizan en general los modelos en el llamado "modelado ágil" (Ambler y Jeffries, 2002). Cuando los modelos se usan como documentación, no tienen que estar completos, pues quizás usted sólo desee desarrollar modelos para algunas partes de un sistema. Sin embargo, estos modelos deben ser correctos: tienen que usar adecuadamente la notación y ser una descripción precisa del sistema.

El Lenguaje de Modelado Unificado

El Lenguaje de Modelado Unificado es un conjunto compuesto por 13 diferentes tipos de diagrama para modelar sistemas de software. Surgió del trabajo en la década de 1990 sobre el modelado orientado a objetos, cuando anotaciones similares, orientadas a objetos, se integraron para crear el UML. Una amplia revisión (UML 2) se finalizó en 2004. El UML es aceptado universalmente como el enfoque estándar al desarrollo de modelos de sistemas de software. Se han propuesto variantes más generales para el modelado de sistemas.

http://www.SoftwareEngineering-9.com/Web/UML/

En el tercer caso, en que los modelos se usan como parte de un proceso de desarrollo basado en modelo, los modelos de sistema deben ser completos y correctos. La razón para esto es que se usan como base para generar el código fuente del sistema. Por lo tanto, debe ser muy cuidadoso de no confundir símbolos equivalentes, como las flechas de palo y las de bloque, que tienen significados diferentes.

5.1 Modelos de contexto

En una primera etapa en la especificación de un sistema, debe decidir sobre las fronteras del sistema. Esto implica trabajar con los participantes del sistema para determinar cuál funcionalidad se incluirá en el sistema y cuál la ofrece el entorno del sistema. Tal vez decida que el apoyo automatizado para algunos procesos empresariales deba implementarse, pero otros deben ser procesos manuales o soportados por sistemas diferentes. Debe buscar posibles traslapes en la funcionalidad con los sistemas existentes y determinar dónde tiene que implementarse nueva funcionalidad. Estas decisiones deben hacerse oportunamente durante el proceso, para limitar los costos del sistema, así como el tiempo necesario para comprender los requerimientos y el diseño del sistema.

En algunos casos, la frontera entre un sistema y su entorno es relativamente clara. Por ejemplo, donde un sistema automático sustituye un sistema manual o computarizado, el entorno del nuevo sistema, por lo general, es el mismo que el entorno del sistema existente. En otros casos, existe más flexibilidad y usted es quien decide qué constituye la frontera entre el sistema y su entorno, durante el proceso de ingeniería de requerimientos.

Por ejemplo, imagine que desarrolla la especificación para el sistema de información de pacientes para atención a la salud mental. Este sistema intenta manejar la información sobre los pacientes que asisten a clínicas de salud mental y los tratamientos que les prescriben. Al desarrollar la especificación para este sistema, debe decidir si el sistema tiene que enfocarse exclusivamente en reunir información de las consultas (junto con otros sistemas para recopilar información personal acerca de los pacientes), o si también es necesario que recopile datos personales acerca del paciente. La ventaja de apoyarse en otros sistemas para la información del paciente es que evita duplicar datos. Sin embargo, la principal desventaja es que usar otros sistemas haría más lento el acceso a la información. Si estos sistemas no están disponibles, entonces no pueden usarse en MHC-PMS.

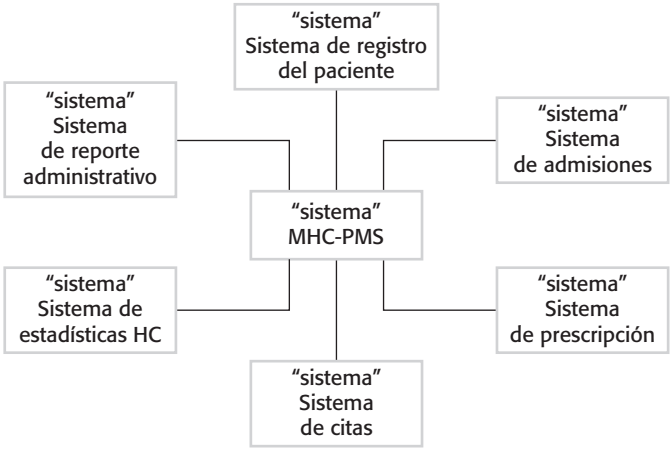


Figura 5.1 El contexto del MHC-PMS

La definición de frontera de un sistema no es un juicio libre de valor. Las preocupaciones sociales y organizacionales pueden significar que la posición de la frontera de un sistema se determine considerando factores no técnicos. Por ejemplo, una frontera de sistema puede colocarse deliberadamente, de modo que todo el proceso de análisis se realice en un sitio; puede elegirse de forma que sea innecesario consultar a un administrador particularmente difícil; puede situarse de manera que el costo del sistema aumente y la división de desarrollo del sistema deba, por lo tanto, expandirse al diseño y la implementación del sistema.

Una vez tomadas algunas decisiones sobre las fronteras del sistema, parte de la actividad de análisis es la definición de dicho contexto y las dependencias que un sistema tiene con su entorno. Normalmente, producir un modelo arquitectónico simple es el primer paso en esta actividad.

La figura 5.1 es un modelo de contexto simple que muestra el sistema de información del paciente y otros sistemas en su entorno. A partir de la figura 5.1, se observa que el MHC-PMS está conectado con un sistema de citas y un sistema más general de registro de pacientes, con el cual comparte datos. El sistema también está conectado a sistemas para manejo de reportes y asignación de camas de hospital, y un sistema de estadísticas que recopila información para la investigación. Finalmente, utiliza un sistema de prescripción que elabora recetas para la medicación de los pacientes.

Los modelos de contexto, por lo general, muestran que el entorno incluye varios sistemas automatizados. Sin embargo, no presentan los tipos de relaciones entre los sistemas en el entorno y el sistema que se especifica. Los sistemas externos generan datos para el sistema o consumen datos del sistema. Pueden compartir datos con el sistema, conectarse directamente, a través de una red, o no conectarse en absoluto. Pueden estar físicamente juntos o ubicados en edificios separados. Todas estas relaciones llegan a afectar los requerimientos y el diseño del sistema a definir, por lo que deben tomarse en cuenta.

Por consiguiente, los modelos de contexto simples se usan junto con otros modelos, como los modelos de proceso empresarial. Éstos describen procesos humanos y automatizados que se usan en sistemas particulares de software.

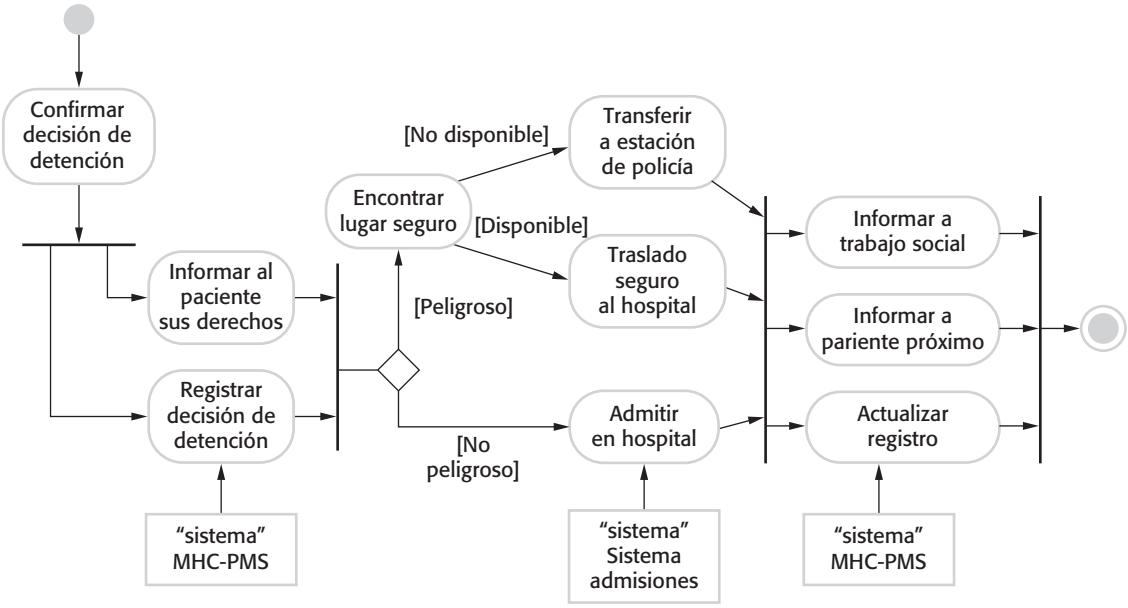


Figura 5.2 Modelo del proceso de detención involuntaria

La figura 5.2 es un modelo de un importante proceso de sistema que muestra los procesos en que se utiliza el MHC-PMS. En ocasiones, los pacientes que sufren de problemas de salud mental son un riesgo para otros o para sí mismos. Por ello, es posible que en un hospital deban mantenerse contra su voluntad para que se les suministre el tratamiento. Tal detención está sujeta a estrictas protecciones legales, por ejemplo, la decisión de detener a un paciente tiene que revisarse con regularidad, para que no se detenga a la persona indefinidamente sin una buena razón. Una de las funciones del MHC-PMS es garantizar que se implementen dichas protecciones.

La figura 5.2 es un diagrama de actividad UML. Los diagramas de actividad intentan mostrar las actividades que incluyen un proceso de sistema, así como el flujo de control de una actividad a otra. El inicio de un proceso se indica con un círculo lleno; el fin, mediante un círculo lleno dentro de otro círculo. Los rectángulos con esquinas redondeadas representan actividades, esto es, los subprocesos específicos que hay que realizar. Puede incluir objetos en los gráficos de actividad. En la figura 5.2 se muestran los sistemas que sirven para apoyar diferentes procesos. Se indicó que éstos son sistemas separados al usar la característica de estereotipo UML.

En un diagrama de actividad UML, las flechas representan el flujo de trabajo de una actividad a otra. Una barra sólida se emplea para indicar coordinación de actividades. Cuando el flujo de más de una actividad se dirige a una barra sólida, entonces todas esas actividades deben completarse antes del posible avance. Cuando el flujo de una barra sólida conduzca a algunas actividades, éstas pueden ejecutarse en forma paralela. Por consiguiente, en la figura 5.2, las actividades para informar a trabajo social y al familiar cercano del paciente, así como para actualizar el registro de detención, pueden ser concurrentes.

Las flechas pueden anotarse con guardas que indiquen la condición al tomar dicho flujo. En la figura 5.2 se observan guardas que muestran los flujos para pacientes que son

un riesgo para la sociedad y quienes no lo son. Los pacientes peligrosos deben mantenerse en una instalación segura. No obstante, los pacientes suicidas que, por lo tanto, representan un riesgo para sí mismos, se detendrían en un pabellón hospitalario adecuado.

5.2 Modelos de interacción

Todos los sistemas incluyen interacciones de algún tipo. Éstas son interacciones del usuario, que implican entradas y salidas del usuario; interacciones entre el sistema a desarrollar y otros sistemas; o interacciones entre los componentes del sistema. El modelado de interacción del usuario es importante, pues ayuda a identificar los requerimientos del usuario. El modelado de la interacción sistema a sistema destaca los problemas de comunicación que se lleguen a presentar. El modelado de interacción de componentes ayuda a entender si es probable que una estructura de un sistema propuesto obtenga el rendimiento y la confiabilidad requeridos por el sistema.

En esta sección se cubren dos enfoques relacionados con el modelado de interacción:

1. Modelado de caso de uso, que se utiliza principalmente para modelar interacciones entre un sistema y actores externos (usuarios u otros sistemas).
2. Diagramas de secuencia, que se emplean para modelar interacciones entre componentes del sistema, aunque también pueden incluirse agentes externos.

Los modelos de caso de uso y los diagramas de secuencia presentan la interacción a diferentes niveles de detalle y, por lo tanto, es posible utilizarlos juntos. Los detalles de las interacciones que hay en un caso de uso de alto nivel se documentan en un diagrama de secuencia. El UML también incluye diagramas de comunicación usados para modelar interacciones. Aquí no se analiza esto, ya que se trata de representaciones alternativas de gráficos de secuencia. De hecho, algunas herramientas pueden generar un diagrama de comunicación a partir de un diagrama de secuencia.

5.2.1 Modelado de casos de uso

El modelado de casos de uso fue desarrollado originalmente por Jacobson y sus colaboradores (1993) en la década de 1990, y se incorporó en el primer lanzamiento del UML (Rumbaugh *et al*., 1999). Como se estudió en el capítulo 4, el modelado de casos de uso se utiliza ampliamente para apoyar la adquisición de requerimientos. Un caso de uso puede tomarse como un simple escenario que describa lo que espera el usuario de un sistema.

Cada caso de uso representa una tarea discreta que implica interacción externa con un sistema. En su forma más simple, un caso de uso se muestra como una elipse, con los actores que intervienen en el caso de uso representados como figuras humanas. La figura 5.3 presenta un caso de uso del MHC-PMS que implica la tarea de subir datos desde el MHC-PMS hasta un sistema más general de registro de pacientes. Este sistema más general mantiene un resumen de datos sobre el paciente, en vez de los datos sobre cada consulta, que se registran en el MHC-PMS.

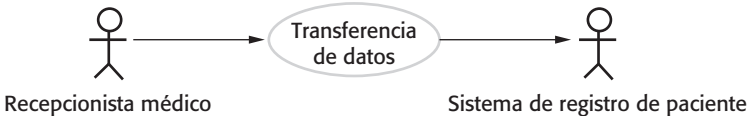


Figura 5.3 Caso de uso de transferencia de datos

Observe que en este caso de uso hay dos actores: el operador que transfiere los datos y el sistema de registro de pacientes. La notación con figura humana se desarrolló originalmente para cubrir la interacción entre individuos, pero también se usa ahora para representar otros sistemas externos y el hardware. De manera formal, los diagramas de caso de uso deben emplear líneas sin flechas; las flechas en el UML indican la dirección del flujo de mensajes. Evidentemente, en un caso de uso los mensajes pasan en ambas direcciones. Sin embargo, las flechas en la figura 5.3 se usan de manera informal para indicar que la recepcionista médica inicia la transacción y los datos se transfieren al sistema de registro de pacientes.

Los diagramas de caso de uso brindan un panorama bastante sencillo de una interacción, de modo que usted tiene que ofrecer más detalle para entender lo que está implicado. Este detalle puede ser una simple descripción textual, o una descripción estructurada en una tabla o un diagrama de secuencia, como se discute a continuación. Es posible elegir el formato más adecuado, dependiendo del caso de uso y del nivel de detalle que usted considere se requiera en el modelo. Para el autor, el formato más útil es un formato tabular estándar. La figura 5.4 ilustra una descripción tabular del caso de uso "transferencia de datos".

Como vimos en el capítulo 4, los diagramas de caso de uso compuestos indican un número de casos de uso diferentes. En ocasiones, se incluyen todas las interacciones posibles con un sistema en un solo diagrama de caso de uso compuesto. Sin embargo, esto quizá sea imposible debido a la cantidad de casos de uso. En tales situaciones, puede desarrollar varios diagramas, cada uno de los cuales exponga casos de uso relacionados. Por ejemplo, la figura 5.5 presenta todos los casos de uso en el MHC-PMS, en los cuales interviene el actor "recepcionista médico".

Figura 5.4 Descripción tabular del caso de uso "transferencia de datos"

MHC-PMS: Transferencia de datos	
Actores	Recepcionista médico, sistema de registros de paciente (PRS).
Descripción	Un recepcionista puede transferir datos del MHC-PMS a una base de datos general de registro de pacientes, mantenida por una autoridad sanitaria. La información transferida puede ser información personal actualizada (dirección, número telefónico, etc.) o un resumen del diagnóstico y tratamiento del paciente.
Datos	Información personal del paciente, resumen de tratamiento.
Estímulo	Comando de usuario emitido por recepcionista médico.
Respuesta	Confirmación de que el PRS se actualizó.
Comentarios	El recepcionista debe tener permisos de seguridad adecuados para acceder a la información del paciente y al PRS.

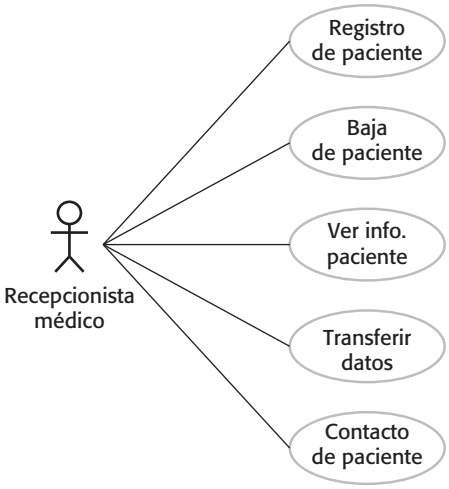


Figura 5.5 Casos de uso que involucran el papel "recepcionista médico"

5.2.2 Diagramas de secuencia

Los diagramas de secuencia en el UML se usan principalmente para modelar las interacciones entre los actores y los objetos en un sistema, así como las interacciones entre los objetos en sí. El UML tiene una amplia sintaxis para diagramas de secuencia, lo cual permite muchos tipos diferentes de interacción a modelar. Como aquí no hay espacio para cubrir todas las posibilidades, sólo nos enfocaremos en lo básico de este tipo de diagrama.

Como sugiere el nombre, un diagrama de secuencia muestra la sucesión de interacciones que ocurre durante un caso de uso particular o una instancia de caso de uso. La figura 5.6 es un ejemplo de un diagrama de secuencia que ilustra los fundamentos de la notación. Estos modelos de diagrama incluyen las interacciones en el caso de uso "ver información de paciente", donde un recepcionista médico puede conocer la información de algún paciente.

Los objetos y actores que intervienen se mencionan a lo largo de la parte superior del diagrama, con una línea punteada que se dibuja verticalmente a partir de éstos. Las interacciones entre los objetos se indican con flechas dirigidas. El rectángulo sobre las líneas punteadas indica la línea de vida del objeto tratado (es decir, el tiempo que la instancia del objeto está involucrada en la computación). La secuencia de interacciones se lee de arriba abajo. Las anotaciones sobre las flechas señalan las llamadas a los objetos, sus parámetros y los valores que regresan. En este ejemplo, también se muestra la notación empleada para exponer alternativas. Un recuadro marcado con "alt" se usa con las condiciones indicadas entre corchetes.

La figura 5.6 se lee del siguiente modo:

1. El recepcionista médico activa el método ViewInfo (ver información) en una instancia P de la clase de objeto PatientInfo, y suministra el identificador del paciente, PID. P es un objeto de interfaz de usuario, que se despliega como un formato que muestra la información del paciente.
2. La instancia P llama a la base de datos para regresar la información requerida, y suministra el identificador del recepcionista para permitir la verificación de seguridad (en esta etapa no se preocupe de dónde proviene este UID).

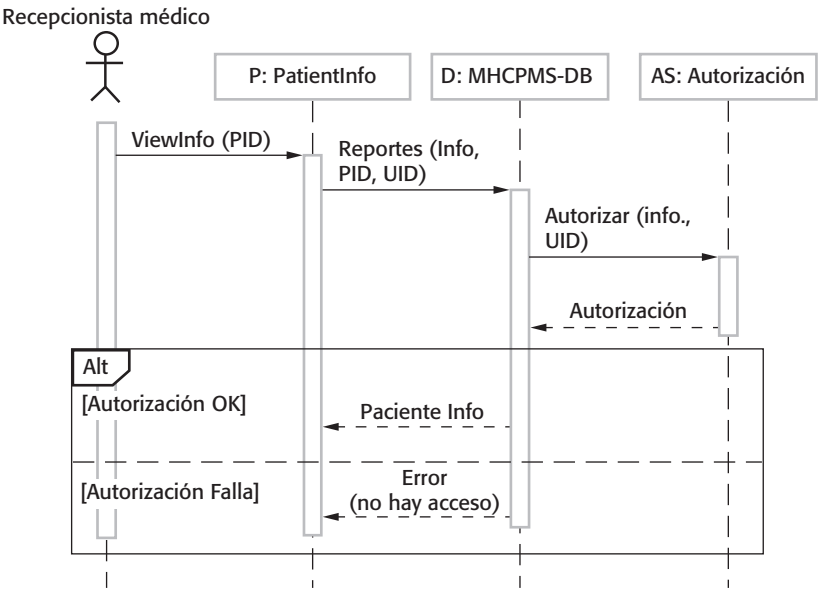


Figura 5.6 Diagrama de secuencia para "ver información del paciente"

3. La base de datos comprueba, mediante un sistema de autorización, que el usuario esté autorizado para tal acción.
4. Si está autorizado, se regresa la información del paciente y se llena un formato en la pantalla del usuario. Si la autorización falla, entonces se regresa un mensaje de error.

La figura 5.7 es un segundo ejemplo de un diagrama de secuencia del mismo sistema que ilustra dos características adicionales. Se trata de la comunicación directa entre los actores en el sistema y la creación de objetos como parte de una secuencia de operaciones. En este ejemplo, un objeto del tipo Summary (resumen) se crea para contener los datos del resumen que deben subirse al PRS (*patient record system*, es decir, el sistema de registro de paciente). Este diagrama se lee de la siguiente manera:

1. El recepcionista inicia sesión (log) en el PRS.
2. Hay dos opciones disponibles. Las opciones permiten la transferencia directa de información actualizada del paciente al PRS, y la transferencia de datos del resumen de salud del MHC-PMS al PRS.
3. En cada caso, se verifican los permisos del recepcionista usando el sistema de autorización.
4. La información personal se transfiere directamente del objeto de interfaz del usuario al PRS. De manera alternativa, es posible crear un registro del resumen de la base de datos y, luego, transferir dicho registro.
5. Al completar la transferencia, el PRS emite un mensaje de estatus y el usuario termina la sesión (log off).

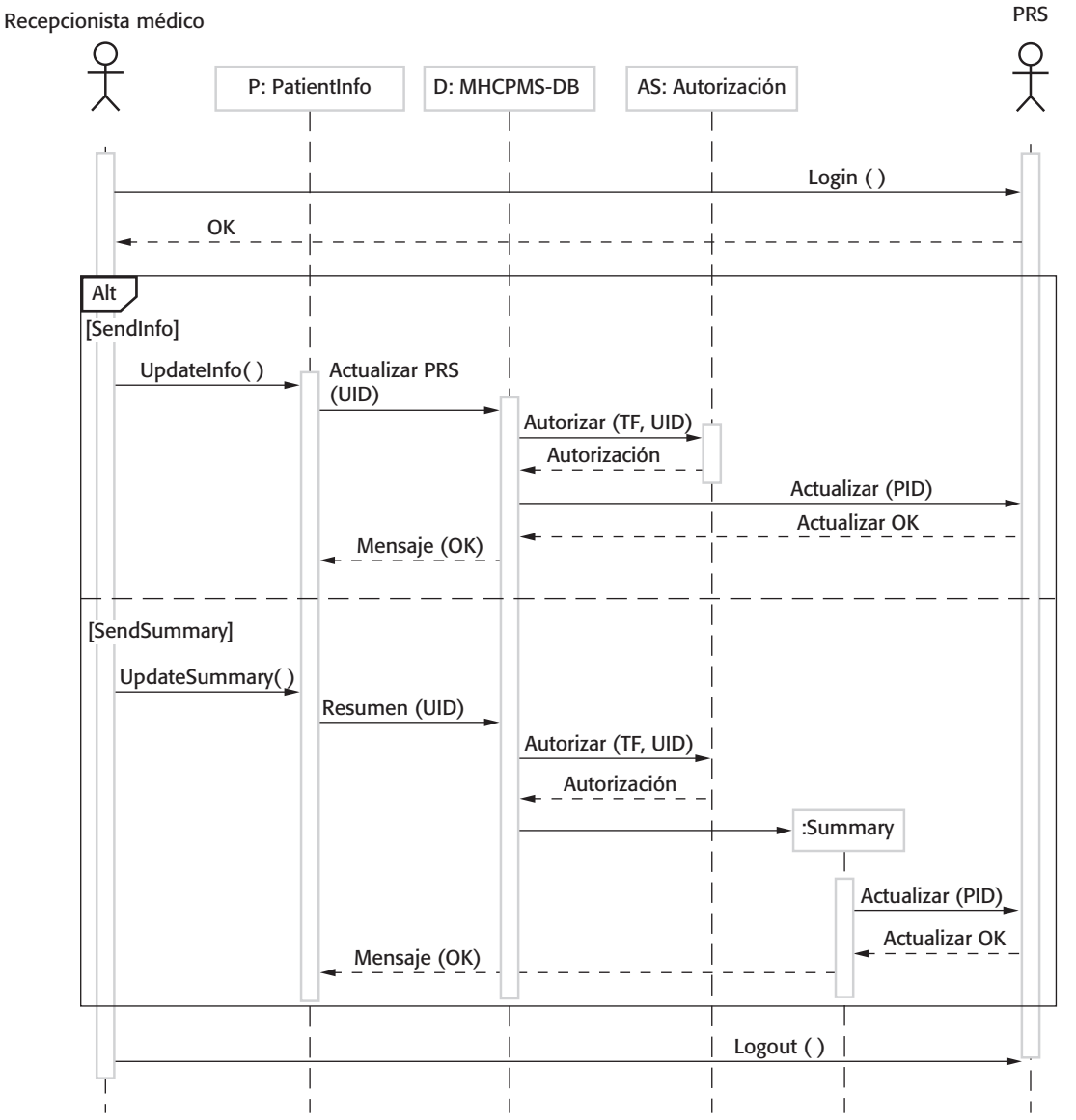


Figura 5.7 Diagrama de secuencia para transferir datos

A menos que use diagramas de secuencia para generación de código o documentación detallada, en dichos diagramas no tiene que incluir todas las interacciones. Si desarrolla modelos iniciales de sistema en el proceso de desarrollo para apoyar la ingeniería de requerimientos y el diseño de alto nivel, habrá muchas interacciones que dependan de decisiones de implementación. Por ejemplo, en la figura 5.7, la decisión sobre cómo conseguir el identificador del usuario para comprobar la autorización podría demorarse. En una implementación, esto implicaría la interacción con un objeto User (usuario), pero esto no es importante en esta etapa y, por lo tanto, no necesita incluirse en el diagrama de secuencia.

Análisis de requerimientos orientado a objetos

En el análisis de requerimientos orientado a objetos, se modelan entidades del mundo real usando clases de objetos. Usted puede crear diferentes tipos de modelos de objetos, que muestren cómo se relacionan mutuamente las clases de objetos, cómo se agregan objetos para formar otros objetos, cómo interactúan los objetos entre sí, etcétera. Cada uno de éstos presenta información única acerca del sistema que se especifica.

http://www.SoftwareEngineering-9.com/Web/OORA/

5.3 Modelos estructurales

Los modelos estructurales de software muestran la organización de un sistema, en términos de los componentes que constituyen dicho sistema y sus relaciones. Los modelos estructurales son modelos estáticos, que muestran la estructura del diseño del sistema, o modelos dinámicos, que revelan la organización del sistema cuando se ejecuta. No son lo mismo: la organización dinámica de un sistema como un conjunto de hilos en interacción tiende a ser muy diferente de un modelo estático de componentes del sistema.

Los modelos estructurales de un sistema se crean cuando se discute y diseña la arquitectura del sistema. El diseño arquitectónico es un tema particularmente importante en la ingeniería de software, y los diagramas UML de componente, de paquete y de implementación se utilizan cuando se presentan modelos arquitectónicos. En los capítulos 6, 18 y 19 se cubren diferentes aspectos de la arquitectura de software y del modelado arquitectónico. Esta sección se enfoca en el uso de diagramas de clase para modelar la estructura estática de las clases de objetos, en un sistema de software.

5.3.1 Diagramas de clase

Los diagramas de clase pueden usarse cuando se desarrolla un modelo de sistema orientado a objetos para mostrar las clases en un sistema y las asociaciones entre dichas clases. De manera holgada, una clase de objeto se considera como una definición general de un tipo de objeto del sistema. Una asociación es un vínculo entre clases, que indica que hay una relación entre dicha clases. En consecuencia, cada clase puede tener algún conocimiento de esta clase asociada.

Cuando se desarrollan modelos durante las primeras etapas del proceso de ingeniería de software, los objetos representan algo en el mundo real, como un paciente, una receta, un médico, etcétera. Conforme se desarrolla una implementación, por lo general se necesitan definir los objetos de implementación adicionales que se usan para dar la funcionalidad requerida del sistema. Aquí, el enfoque está sobre el modelado de objetos del mundo real, como parte de los requerimientos o los primeros procesos de diseño del software.

Los diagramas de clase en el UML pueden expresarse con diferentes niveles de detalle. Cuando se desarrolla un modelo, la primera etapa con frecuencia implica buscar en el mundo, identificar los objetos esenciales y representarlos como clases. La forma más sencilla de hacer esto es escribir el nombre de la clase en un recuadro. También puede anotar la existencia de una asociación dibujando simplemente una línea entre las clases.

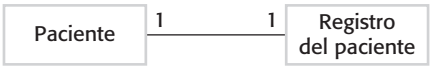


Figura 5.8 Clases y asociación UML

Por ejemplo, la figura 5.8 es un diagrama de clase simple que muestra dos clases: Patient (paciente) y Patient Record (registro del paciente), con una asociación entre ellos.

En la figura 5.8 se ilustra una característica más de los diagramas de clase: la habilidad para mostrar cuántos objetos intervienen en la asociación. En este ejemplo, cada extremo de la asociación se registra con un 1, lo cual significa que hay una relación 1:1 entre objetos de dichas clases. Esto es, cada paciente tiene exactamente un registro, y cada registro conserva información precisa del paciente. En los últimos ejemplos se observa que son posibles otras multiplicidades. Se define un número exacto de objetos que están implicados, o bien, con el uso de un asterisco (*), como se muestra en la figura 5.9, que hay un número indefinido de objetos en la asociación.

La figura 5.9 desarrolla este tipo de diagrama de clase para mostrar que los objetos de la clase "paciente" también intervienen en relaciones con varias otras clases. En este ejemplo, se observa que es posible nombrar las asociaciones para dar al lector un indicio del tipo de relación que existe. Asimismo, el UML permite especificar el papel de los objetos que participan en la asociación.

En este nivel de detalle, los diagramas de clase parecen modelos semánticos de datos. Los modelos semánticos de datos se usan en el diseño de bases de datos. Muestran las entidades de datos, sus atributos asociados y las relaciones entre dichas entidades. Este enfoque para modelar fue propuesto por primera vez por Chen (1976), a mediados de la década de 1970; desde entonces, se han desarrollado diversas variantes (Codd, 1979; Hammer y McLeod, 1981; Hull y King, 1987), todas con la misma forma básica.

El UML no incluye una notación específica para este modelado de bases de datos, ya que supone un proceso de desarrollo orientado a objetos, así como modelos de datos que usan objetos y sus relaciones. Sin embargo, es posible usar el UML para representar un modelo semántico de datos. En un modelo semántico de datos, piense en entidades como

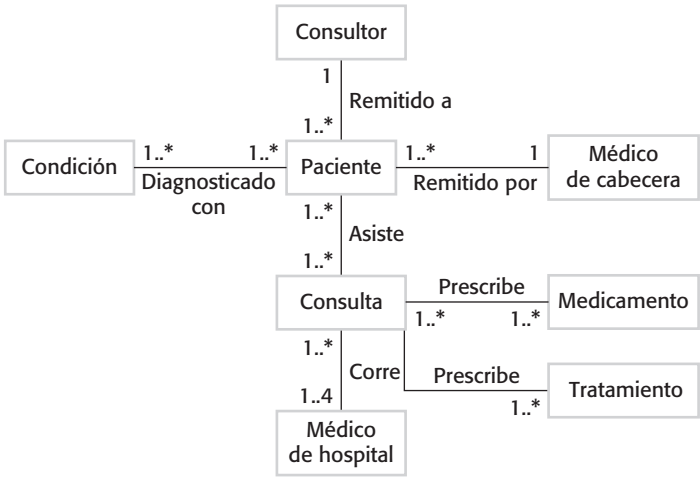


Figura 5.9 Clases y asociaciones en el MHC-PMS

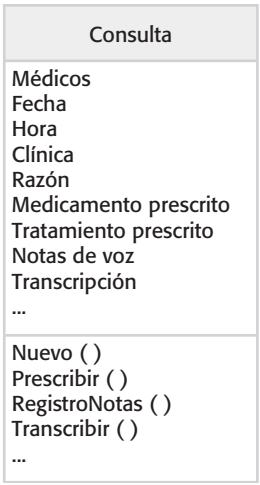


Figura 5.10 La clase de consulta

clases de objeto simplificadas (no tienen operaciones), atributos como atributos de clase de objeto y relaciones como nombres de asociaciones entre clases de objeto.

Cuando se muestran las asociaciones entre clases, es conveniente representar dichas clases en la forma más sencilla posible. Para definirlas con más detalle, agregue información sobre sus atributos (las características de un objeto) y operaciones (aquello que se puede solicitar de un objeto). Por ejemplo, un objeto Patient tendrá el atributo Address (dirección) y puede incluir una operación llamada ChangeAddress (cambiar dirección), que se llama cuando un paciente manifiesta que se mudó de una dirección a otra. En el UML, los atributos y las operaciones se muestran al extender el rectángulo simple que representa una clase. Esto se ilustra en la figura 5.10, donde:

1. El nombre de la clase de objeto está en la sección superior.
2. Los atributos de clase están en la sección media. Esto debe incluir los nombres del atributo y, opcionalmente, sus tipos.
3. Las operaciones (llamadas métodos en Java y en otros lenguajes de programación OO) asociadas con la clase de objeto están en la sección inferior del rectángulo.

La figura 5.10 expone posibles atributos y operaciones sobre la clase Consulta (Consultation). En este ejemplo, se supone que los médicos registran notas de voz que se transcriben más tarde para registrar detalles de la consulta. Al prescribir fármacos, el médico debe usar el método Prescribir (Prescribe) para generar una receta electrónica.

5.3.2 Generalización

La generalización es una técnica cotidiana que se usa para gestionar la complejidad. En vez de aprender las características detalladas de cada entidad que se experimenta, dichas entidades se colocan en clases más generales (animales, automóviles, casas,

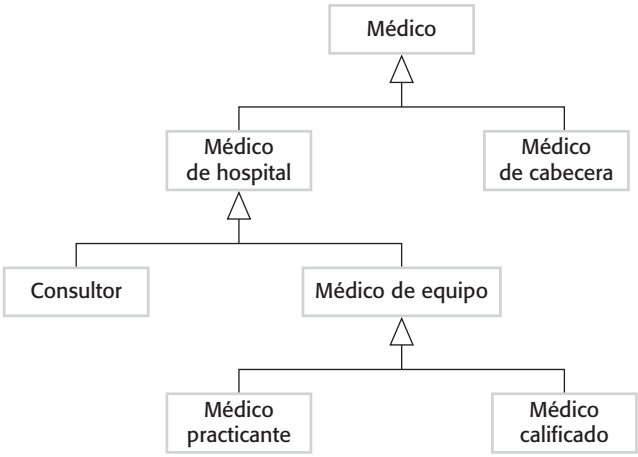


Figura 5.11 Jerarquía de generalización

etcétera) y se aprenden las características de dichas clases. Esto permite deducir que diferentes miembros de estas clases tienen algunas características comunes (por ejemplo, las ardillas y ratas son roedores). Es posible hacer enunciados generales que se apliquen a todos los miembros de la clase (por ejemplo, todos los roedores tienen dientes para roer).

En el modelado de sistemas, con frecuencia es útil examinar las clases en un sistema, con la finalidad de ver si hay ámbito para la generalización. Esto significa que la información común se mantendrá solamente en un lugar. Ésta es una buena práctica de diseño, pues significa que, si se proponen cambios, entonces no se tiene que buscar en todas las clases en el sistema, para observar si se ven afectadas por el cambio. En los lenguajes orientados a objetos, como Java, la generalización se implementa usando los mecanismos de herencia de clase construidos en el lenguaje.

El UML tiene un tipo específico de asociación para denotar la generalización, como se ilustra en la figura 5.11. La generalización se muestra como una flecha que apunta hacia la clase más general. Esto indica que los médicos de cabecera y los médicos de hospital pueden generalizarse como médicos, y que hay tres tipos de médicos de hospital: quienes se graduaron recientemente de la escuela de medicina y tienen que ser supervisados (médicos practicantes); quienes trabajan sin supervisión como parte de un equipo de consultores (médicos registrados); y los consultores, que son médicos experimentados con plenas responsabilidades en la toma de decisiones.

En una generalización, los atributos y las operaciones asociados con las clases de nivel superior también se asocian con las clases de nivel inferior. En esencia, las clases de nivel inferior son subclases que heredan los atributos y las operaciones de sus superclases. Entonces dichas clases de nivel inferior agregan atributos y operaciones más específicos. Por ejemplo, todos los médicos tienen un nombre y número telefónico; todos los médicos de hospital tienen un número de personal y un departamento, pero los médicos de cabecera no tienen tales atributos, pues trabajan de manera independiente. Sin embargo, sí tienen un nombre de consultorio y dirección. Esto se ilustra en la figura 5.12, que muestra parte de la jerarquía de generalización que se extendió con atributos de clase. Las operaciones asociadas con la clase “médico” buscan registrar y dar de baja al médico con el MHC-PMS.

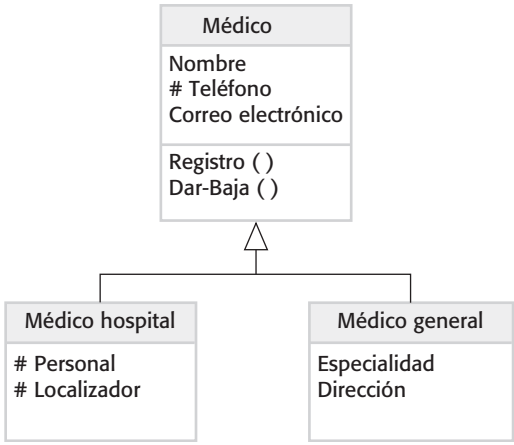


Figura 5.12 Jerarquía de generalización con detalles agregados

5.3.3 Agregación

Los objetos en el mundo real con frecuencia están compuestos por diferentes partes. Un paquete de estudio para un curso, por ejemplo, estaría compuesto por libro, diapositivas de PowerPoint, exámenes y recomendaciones para lecturas posteriores. En ocasiones, en un modelo de sistema, usted necesita ilustrar esto. El UML proporciona un tipo especial de asociación entre clases llamado agregación, que significa que un objeto (el todo) se compone de otros objetos (las partes). Para mostrarlo, se usa un trazo en forma de diamante, junto con la clase que representa el todo. Esto se ilustra en la figura 5.13, que indica que un registro de paciente es una composición de Paciente (Patient) y un número indefinido de Consulta (Consultations).

5.4 Modelos de comportamiento

Los modelos de comportamiento son modelos dinámicos del sistema conforme se ejecutan. En ellos se muestra lo que sucede o lo que se supone que pasa cuando un sistema responde ante un estímulo de su entorno. Tales estímulos son de dos tipos:

1. *Datos* Algunos datos que llegan se procesan por el sistema.
2. *Eventos* Algunos eventos activan el procesamiento del sistema. Los eventos pueden tener datos asociados, pero esto no es siempre el caso.

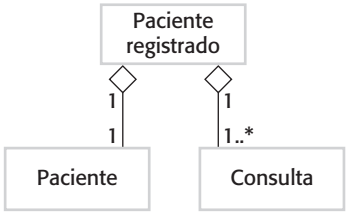


Figura 5.13 La asociación agregación

Diagramas de flujo de datos

Los diagramas de flujo de datos (DFD) son modelos de sistema que presentan una perspectiva funcional, donde cada transformación constituye una sola función o un solo proceso. Los DFD se usan para mostrar cómo fluyen los datos a través de una secuencia de pasos del procesamiento. Por ejemplo, un paso del procesamiento sería el filtrado de registros duplicados en una base de datos de clientes. Los datos se transforman en cada paso antes de moverse hacia la siguiente etapa. Dichos pasos o transformaciones del procesamiento representan procesos o funciones de software, en los cuales los diagramas de flujo de datos se usan para documentar un diseño de software.

http://www.SoftwareEngineering-9.com/Web/DFDs

Muchos sistemas empresariales son sistemas de procesamiento de datos que se activan principalmente por datos. Son controlados por la entrada de datos al sistema con relativamente poco procesamiento externo de eventos. Su procesamiento incluye una secuencia de acciones sobre dichos datos y la generación de una salida. Por ejemplo, un sistema de facturación telefónica aceptará información de las llamadas hechas por un cliente, calculará los costos de dichas llamadas y generará una factura para enviarla a dicho cliente. En contraste, los sistemas de tiempo real muchas veces están dirigidos por un evento con procesamiento de datos mínimo. Por ejemplo, un sistema de conmutación telefónico terrestre responde a eventos como "receptor ocupado" al generar un tono de dial, o al presionar las teclas de un teléfono para la captura del número telefónico, etcétera.

5.4.1 Modelado dirigido por datos

Los modelos dirigidos por datos muestran la secuencia de acciones involucradas en el procesamiento de datos de entrada, así como la generación de una salida asociada. Son particularmente útiles durante el análisis de requerimientos, pues sirven para mostrar procesamiento "extremo a extremo" en un sistema. Esto es, exhiben toda la secuencia de acciones que ocurren desde una entrada a procesar hasta la salida correspondiente, que es la respuesta del sistema.

Los modelos dirigidos por datos están entre los primeros modelos gráficos de software. En la década de 1970, los métodos estructurados como el análisis estructurado de DeMarco (DeMarco, 1978) introdujeron los diagramas de flujo de datos (DFD), como una forma de ilustrar los pasos del procesamiento en un sistema. Los modelos de flujo de datos son útiles porque el hecho de rastrear y documentar cómo los datos asociados con un proceso particular se mueven a través del sistema ayuda a los analistas y diseñadores a entender lo que sucede. Los diagramas de flujo de datos son simples e intuitivos y, por lo general, es posible explicarlos a los usuarios potenciales del sistema, quienes después pueden participar en la validación del modelo.

El UML no soporta diagramas de flujo de datos, puesto que originalmente se propusieron y usaron para modelar el procesamiento de datos. La razón para esto es que los DFD se enfocan en funciones del sistema y no reconocen objetos del sistema. Sin embargo, como los sistemas dirigidos por datos son tan comunes en los negocios, UML 2.0 introdujo diagramas de actividad, que son similares a los diagramas de flujo de datos. Por ejemplo, la figura 5.14 indica la cadena de procesamiento involucrada en el software de la bomba de insulina. En este diagrama, se observan los pasos de procesamiento (representados como actividades) y los datos que fluyen entre dichos pasos (representados como objetos).

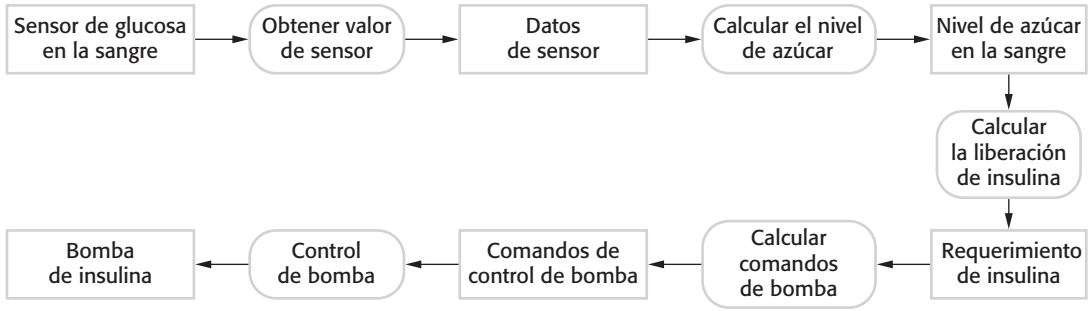


Figura 5.14 Modelo de actividad de la operación de la bomba de insulina

Una forma alternativa de mostrar la secuencia de procesamiento en un sistema es usar diagramas de secuencia UML. Ya se vio cómo se utilizan para modelar interacción pero, si los utiliza para que dichos mensajes sólo se envíen de izquierda a derecha, luego muestran el procesamiento secuencial de datos en el sistema. La figura 5.15 ilustra esto, con un modelo de secuencia del procesamiento de un pedido y envío a un proveedor. Los modelos de secuencia destacan los objetos en un sistema, mientras que los diagramas de flujo de datos resaltan las funciones. El diagrama de flujo de datos equivalente para la orden de procesamiento se incluye en las páginas Web del libro.

5.4.2 Modelado dirigido por un evento

El modelado dirigido por un evento muestra cómo responde un sistema a eventos externos e internos. Se basa en la suposición de que un sistema tiene un número finito de estados y que los eventos (estímulos) pueden causar una transición de un estado a otro. Por ejemplo, un sistema que controla una válvula puede moverse de un estado de "válvula abierta" a un estado de "válvula cerrada", cuando recibe un comando operador (el estímulo). Esta visión de un sistema es adecuado particularmente para sistema en tiempo real. El modelado basado en eventos se introdujo en los métodos de diseño en tiempo real, como los propuestos por Ward y Mellor (1985) y Harel (1987, 1988).

El UML soporta modelado basado en eventos usando diagramas de estado, que se fundamentaron en gráficos de estado (Harel, 1987, 1988). Los diagramas de estado muestran estados y eventos del sistema que causan transiciones de un estado a otro. No exponen el

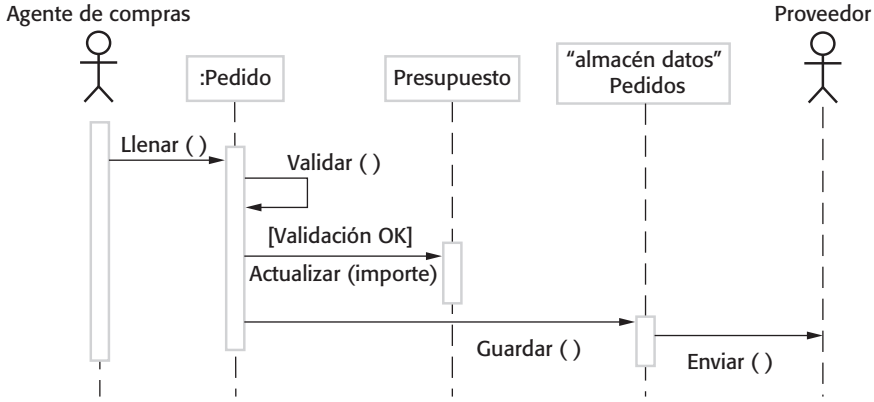


Figura 5.15 Orden de procesamiento

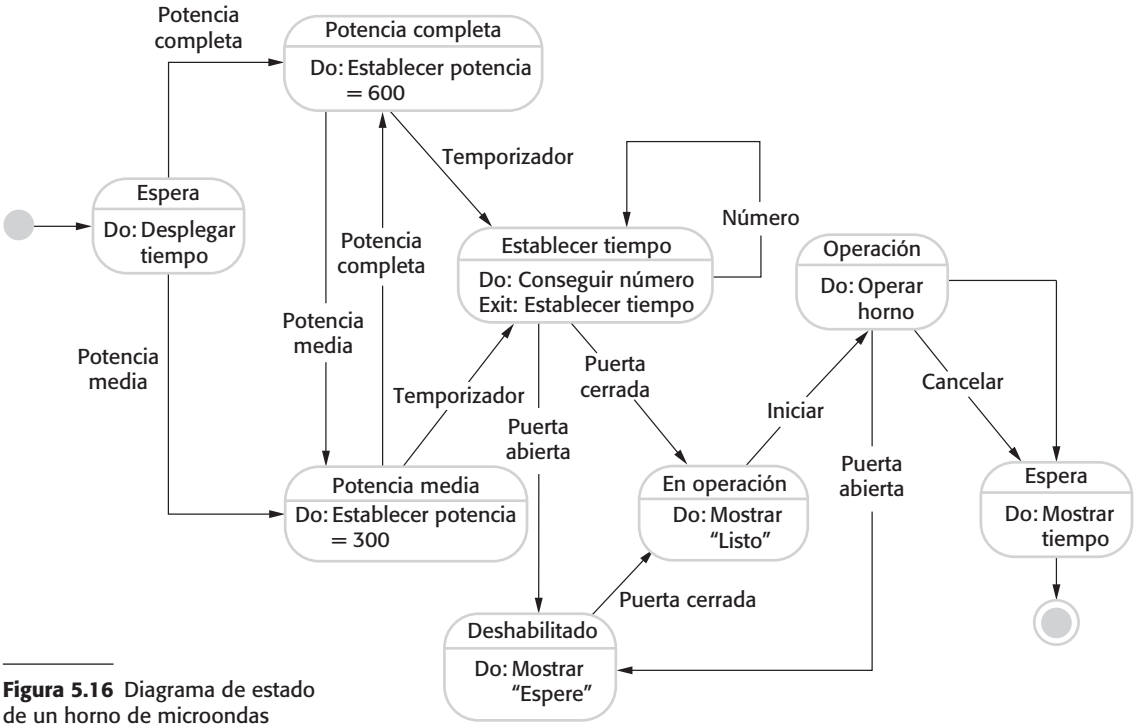


Figura 5.16 Diagrama de estado de un horno de microondas

flujo de datos dentro del sistema, pero suelen incluir información adicional acerca de los cálculos realizados en cada estado.

Se usa un ejemplo de software de control para un horno de microondas muy sencillo, que ilustra el modelado dirigido por un evento. Los hornos de microondas reales son mucho más complejos que este sistema, pero el sistema simplificado es más fácil de entender. Este microondas sencillo tiene un interruptor para seleccionar potencia completa o media, un teclado numérico para ingresar el tiempo de cocción, un botón de iniciar/detener y una pantalla alfanumérica.

Se supone que la secuencia de acciones al usar el horno de microondas es:

1. Seleccionar el nivel de potencia (ya sea media o completa)
2. Ingresar el tiempo de cocción con el teclado numérico.
3. Presionar "Iniciar", y la comida se cocina durante el tiempo dado.

Por razones de seguridad, el horno no opera cuando la puerta esté abierta y, al completar la cocción, se escuchará un timbre. El horno tiene una pantalla alfanumérica muy sencilla que se usa para mostrar varios avisos de alerta y mensajes de advertencia.

En los diagramas de estado UML, los rectángulos redondeados representan estados del sistema. Pueden incluir una breve descripción (después de "do") de las acciones que se tomarán en dicho estado. Las flechas etiquetadas representan estímulos que fuerzan una transición de un estado a otro. Puede indicar los estados inicial y final usando círculos rellenos, como en los diagramas de actividad.

A partir de la figura 5.16 se observa que el sistema empieza en un estado de espera e, inicialmente, responde al botón de potencia completa o al botón de potencia media. Los

Estado	Descripción
Esperar	El horno espera la entrada. La pantalla indica el tiempo actual.
Potencia media	La potencia del horno se establece en 300 watts. La pantalla muestra “Potencia media”.
Potencia completa	La potencia del horno se establece en 600 watts. La pantalla muestra “Potencia completa”.
Establecer tiempo	El tiempo de cocción se establece al valor de entrada del usuario. La pantalla indica el tiempo de cocción seleccionado y se actualiza conforme se establece el tiempo.
Deshabilitado	La operación del horno se deshabilita por cuestiones de seguridad. La luz interior del horno está encendida. La pantalla indica “No está listo”.
Habilitado	Se habilita la operación del horno. La luz interior del horno está apagada. La pantalla muestra “Listo para cocinar”.
Operación	Horno en operación. La luz interior del horno está encendida. La pantalla muestra la cuenta descendente del temporizador. Al completar la cocción, suena el timbre durante cinco segundos. La luz del horno está encendida. La pantalla muestra “Cocción completa” mientras suena el timbre.

Estímulo	Descripción
Potencia media	El usuario oprime el botón de potencia media.
Potencia completa	El usuario oprime el botón de potencia completa.
Temporizador	El usuario oprime uno de los botones del temporizador.
Número	El usuario oprime una tecla numérica.
Puerta abierta	El interruptor de la puerta del horno no está cerrado.
Puerta cerrada	El interruptor de la puerta del horno está cerrado.
Iniciar	El usuario oprime el botón Iniciar.
Cancelar	El usuario oprime el botón Cancelar.

Figura 5.17 Estados y estímulos para el horno de microondas

usuarios pueden cambiar su opinión después de seleccionar uno de ellos y oprimir el otro botón. Se establece el tiempo y, si la puerta está cerrada, se habilita el botón Iniciar. Al presionar este botón comienza la operación del horno y tiene lugar la cocción durante el tiempo especificado. Éste es el final del ciclo de cocción y el sistema regresa al estado de espera.

La notación UML permite indicar la actividad que ocurre en un estado. En una especificación detallada del sistema, hay que proporcionar más detalle tanto de los estímulos como de los estados del sistema. Esto se ilustra en la figura 5.17, la cual señala una descripción tabular de cada estado y cómo se generan los estímulos que fuerzan transiciones de estado.

El problema con el modelado basado en el estado es que el número de posibles estados se incrementa rápidamente. Por lo tanto, para modelos de sistemas grandes, necesita ocultar

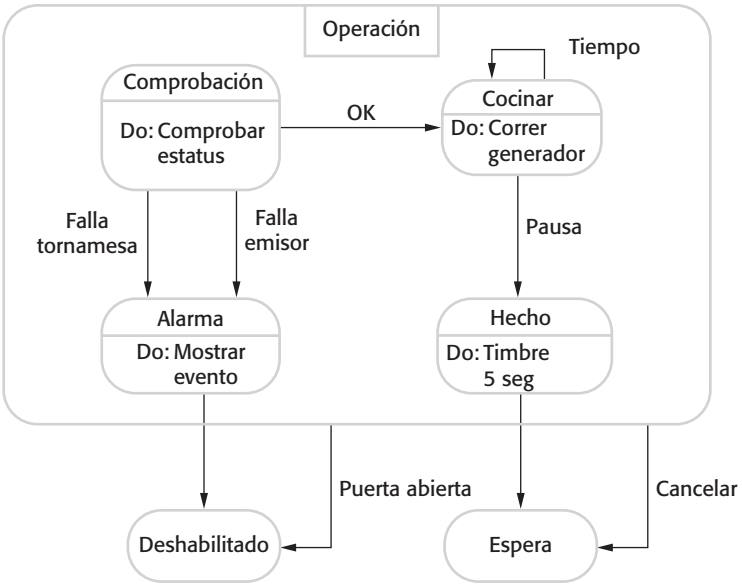


Figura 5.18 Operación del horno de microondas

detalles en los modelos. Una forma de hacer esto es mediante la noción de un superestado que encapsule algunos estados separados. Este superestado se parece a un solo estado en un modelo de nivel superior, pero entonces se expande para mostrar más detalles en un diagrama separado. Para ilustrar este concepto, considere el estado Operación en la figura 5.15. Éste es un superestado que puede expandirse, como se ilustra en la figura 5.18.

El estado Operación incluye algunos subestados. Muestra que la operación comienza con una comprobación de estatus y que, si se descubren problemas, se indica una alarma y la operación se deshabilita. La cocción implica operar el generador de microondas durante el tiempo especificado; al terminar, suena un timbre. Si la puerta está abierta durante la operación, el sistema se mueve hacia el estado deshabilitado, como se muestra en la figura 5.15.

5.5 Ingeniería dirigida por modelo

La ingeniería dirigida por modelo (MDE, por las siglas de *Model-Driven Engineering*) es un enfoque al desarrollo de software donde los modelos, y no los programas, son las salidas principales del proceso de desarrollo (Kent, 2002; Schmidt, 2006). Los programas que se ejecutan en una plataforma hardware/software se generan en tal caso automáticamente a partir de los modelos. Los partidarios de la MDE argumentan que ésta eleva el nivel de abstracción en la ingeniería de software, pues los ingenieros ya no tienen que preocuparse por detalles del lenguaje de programación o las especificidades de las plataformas de ejecución.

La ingeniería dirigida por modelo tiene sus raíces en la arquitectura dirigida por modelo (MDA, por las siglas de *Model-Driven Architecture*), que fue propuesta por el Object Management Group (OMG) en 2001 como un nuevo paradigma de desarrollo de software. La ingeniería dirigida por modelo y la arquitectura dirigida por modelo se

ven normalmente iguales. Sin embargo, se considera que la MDE tiene un ámbito más amplio que la MDA. Como se estudia más adelante en esta sección, la MDA se enfoca en las etapas de diseño e implementación del desarrollo de software, mientras que la MDE se interesa por todos los aspectos del proceso de ingeniería de software. Por lo tanto, los temas como ingeniería de requerimientos basada en un modelo, procesos de software para desarrollo basado en un modelo, y pruebas basadas en un modelo son parte de MDE, pero no, en este momento, de la MDA.

Aunque la MDA se usa desde 2001, la ingeniería basada en modelo aún está en una etapa temprana de desarrollo, y no es claro si tendrá o no un efecto significativo sobre la práctica de ingeniería de software. Los principales argumentos a favor y en contra de MDE son:

1. *En favor de la MDE* La ingeniería basada en modelo permite a los ingenieros pensar sobre sistemas en un nivel de abstracción elevado, sin ocuparse por los detalles de su implementación. Esto reduce la probabilidad de errores, acelera el diseño y el proceso de implementación, y permite la creación de modelos de aplicación reutilizables, independientes de la plataforma de aplicación. Al usar herramientas poderosas, las implementaciones de sistema pueden generarse para diferentes plataformas a partir del mismo modelo. En consecuencia, para adaptar el sistema a alguna nueva plataforma tecnológica, sólo es necesario escribir un traductor para dicha plataforma. Cuando está disponible, todos los modelos independientes de plataforma pueden reubicarse rápidamente en la nueva plataforma.

2. *Contra la MDE* Como se analizó anteriormente en este capítulo, los modelos son una buena forma de facilitar las discusiones sobre un diseño de software. Sin embargo, no siempre se sigue que las abstracciones que soporta el modelo son las abstracciones correctas para la implementación. De este modo, es posible crear modelos de diseño informal, pero siendo así, el sistema se implementa usando un paquete configurable comercial (*off-the-shelf*). Más aún, los argumentos para independencia de plataforma sólo son válidos para sistemas grandes de larga duración, donde las plataformas se vuelven obsoletas durante el tiempo de vida de un sistema. Sin embargo, para esta clase de sistemas, se sabe que la implementación no es el principal problema: ingeniería de requerimientos, seguridad y confiabilidad, integración con sistemas heredados, y las pruebas son más significativos.

El OMG reporta en sus páginas Web (www.omg.org/mda/products_success.htm) historias de éxito reveladoras de la MDE y el enfoque utilizado dentro de grandes compañías como IBM y Siemens. Las técnicas se usaron con éxito en el desarrollo de grandes sistemas de software de larga duración, como sistemas de manejo de tráfico aéreo. No obstante, en el momento de escribir este texto, los enfoques dirigidos por modelo no son ampliamente usados por la ingeniería de software. Como los métodos formales de la ingeniería de software, que se estudian en el capítulo 12, se considera que MDE es un importante desarrollo. Pero, como también es el caso con los métodos formales, no es claro si los costos y riesgos de los enfoques dirigidos por modelo superan los posibles beneficios.

5.5.1 Arquitectura dirigida por modelo

La arquitectura dirigida por modelo (Kleppe *et al*., 2003; Mellor *et al*., 2004; Stahl y Voelter, 2006) es un enfoque orientado a un modelos para el diseño y la implementación de software, que usa un subconjunto de modelos UML para describir un sistema. Aquí, se

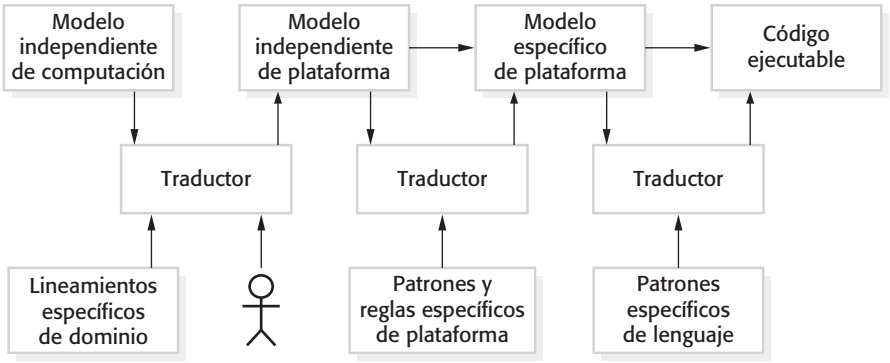


Figura 5.19 Transformaciones MDA

crean modelos a diferentes niveles de abstracción. A partir de un modelo independiente de plataforma de alto nivel, es posible, en principio, generar un programa funcional sin intervención manual.

El método de MDA recomienda la producción de tres tipos de modelo de sistema abstracto:

1. Un modelo independiente de computación (CIM) que modela las importantes abstracciones de dominio usadas en el sistema. En ocasiones, los CIM se llaman modelos de dominio. Es posible desarrollar varios CIM diferentes, que reflejen distintas percepciones del sistema. Por ejemplo, puede haber un CIM de seguridad, en el cual se identifiquen abstracciones de seguridad importantes, como un CIM de activo, un rol y un registro del paciente, que describan abstracciones como pacientes, consultas, etcétera.
2. Un modelo independiente de plataforma (PIM) que modele la operación del sistema sin referencia a su implementación. El PIM se describe usualmente mediante modelos UML que muestran la estructura estática del sistema y cómo responde a eventos externos e internos.
3. Modelos específicos de plataforma (PSM) que son transformaciones del modelo independiente de plataforma con un PSM separado para cada plataforma de aplicación. En principio, puede haber capas de PSM, y cada una agrega cierto detalle específico de la plataforma. De este modo, el PSM de primer nivel podría ser específico de "middleware", pero independiente de la base de datos. Cuando se elige una base de datos específica, podría generarse entonces un PSM específico de base de datos.

Como vimos, las transformaciones entre dichos modelos pueden definirse y aplicarse automáticamente con herramientas de software. Esto se ilustra en la figura 5.19, que también muestra un nivel final de transformación automática. Una transformación se aplica al PSM para generar un código ejecutable que opere en la plataforma de software designada.

En el momento de escribir este texto, la traducción automática CIM a PIM todavía está en etapa de investigación de prototipo. Es improbable que en el futuro cercano estén disponibles herramientas de traducción completamente automatizadas. Para el futuro

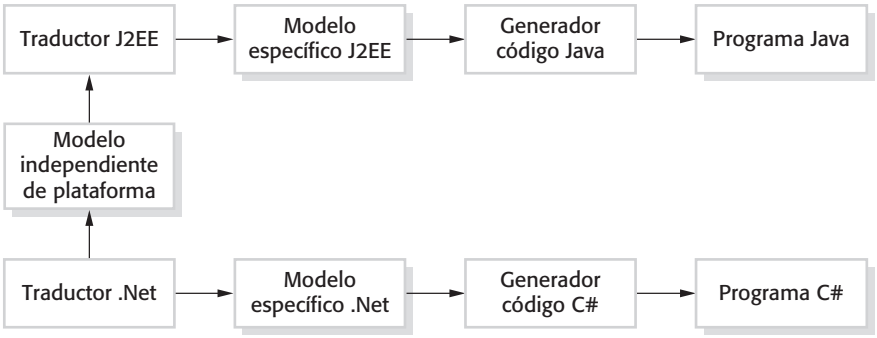


Figura 5.20 Múltiples modelos específicos de plataforma

previsible se necesitará la intervención humana, lo que se indica mediante una figura ilustrativa en la figura 5.19. Los CIM se relacionan, y parte del proceso de traducción puede involucrar conceptos de vinculación en diferentes CIM. Por ejemplo, el concepto de un papel en un CIM de seguridad que puede trazarse dentro del concepto de un miembro de personal en un CIM de hospital. Mellor y Balcer (2002) dan el nombre de "puentes" a la información que soporta el mapeo de un CIM a otro.

La traducción de PIM a PSM es más madura y se dispone de varias herramientas comerciales que proporcionan traductores de PIM a plataformas comunes como Java y J2EE. Éstas se apoyan en una extensa librería de reglas y patrones específicos de plataforma para convertir el PIM al PSM. Puede haber muchos PSM para cada PIM en el sistema. Si se tiene la intención de que un sistema de software funcione en diferentes plataformas (por ejemplo J2EE y .NET), entonces sólo es necesario mantener el PIM. Los PSM para cada plataforma se generan automáticamente. Esto se ilustra en la figura 5.20.

Aunque las herramientas de soporte MDA incluyen traductores específicos de plataforma, es frecuente el caso de que sólo ofrezcan soporte parcial para la traducción de PIM a PSM. En la gran mayoría de los casos, el entorno de ejecución para un sistema es más que la plataforma de ejecución estándar (por ejemplo, J2EE, .NET, etcétera). También incluye otros sistemas de aplicación, librerías de aplicación que son específicas a una compañía y librerías de interfaz de usuario. Como éstas varían significativamente de una compañía a otra, no está disponible un soporte estándar para herramientas. Por lo tanto, cuando se introduce MDA, quizá deban crearse traductores de propósito especial que tomen en cuenta las características del entorno local. En algunos casos (por ejemplo, para generación de interfaz de usuario), la traducción de PIM a PSM completamente automatizada es imposible.

Existe una relación difícil entre métodos ágiles y arquitectura dirigida por modelo. La noción de modelado frontal extenso contradice las ideas fundamentales del manifiesto ágil y se conjetura que pocos desarrolladores ágiles se sienten cómodos con la ingeniería dirigida por modelo. Los desarrolladores de MAD afirman que se tiene la intención de apoyar un enfoque iterativo para el desarrollo y, por lo tanto, puede usarse dentro de los métodos ágiles (Mellor *et al*., 2004). Si las transformaciones pueden automatizarse completamente y a partir de un PIM se genera un programa completo, entonces, en principio, MDA podría usarse en un proceso de desarrollo ágil, ya que no se requeriría codificación separada. Sin embargo, hasta donde se sabe, no hay herramientas de MDA que soporten prácticas como las pruebas de regresión y el desarrollo dirigido por pruebas.

5.5.2 UML ejecutable

La noción fundamental detrás de la ingeniería dirigida por modelo es que debe ser posible la transformación completamente automatizada de modelos a código. Para lograr esto, usted tiene que ser capaz de construir modelos gráficos, cuya semántica esté bien definida. También necesita una forma de agregar a los modelos gráficos, información sobre la forma en que se implementan las operaciones definidas en el modelo. Esto es posible usando un subconjunto de UML 2 llamado UML ejecutable o xUML (Mellor y Balcer, 2002). Aquí no hay espacio para describir los detalles del xUML, así que simplemente se presentará un breve panorama de sus principales características.

El UML se desarrolló como un lenguaje para soportar y documentar diseño de software, no como un lenguaje de programación. Los diseñadores del UML no estaban preocupados por los detalles semánticos del lenguaje, sino con su expresividad. Introdujeron nociones útiles como los diagramas de caso de uso, que ayudan con el diseño, pero que son demasiado informales para soportar la ejecución. Por ende, para crear un subconjunto ejecutable de UML, el número de tipos de modelo se redujo drásticamente a tres tipos de modelo clave:

1. Modelos de dominio que identifican las principales preocupaciones en el sistema. Se definen usando diagramas de clase UML que incluyen objetos, atributos y asociaciones.
2. Modelos de clase, en los que se definen clases, junto con sus atributos y operaciones.
3. Modelos de estado, en los que un diagrama de estado se asocia con cada clase y se usa para describir el ciclo de vida de la clase.

El comportamiento dinámico del sistema puede especificarse de manera declarativa usando el lenguaje de restricción de objeto (OCL) o puede expresarse mediante el lenguaje de acción de UML. El lenguaje de acción es como un lenguaje de programación de muy alto nivel, donde es posible referirse a los objetos y sus atributos, así como especificar acciones a realizar.

PUNTOS CLAVE

- Un modelo es una visión abstracta de un sistema que ignora algunos detalles del sistema. Pueden desarrollarse modelos complementarios del sistema para mostrar el contexto, las interacciones, la estructura y el comportamiento del sistema.
- Los modelos de contexto muestran cómo un sistema a modelar se coloca en un entorno con otros sistemas y procesos. Ayudan a definir las fronteras del sistema a desarrollar.
- Los diagramas de caso de uso y los diagramas de secuencia se emplean para describir las interacciones entre usuario/sistema a diseñar y usuarios/otros sistemas. Los casos de uso describen interacciones entre un sistema y actores externos; los diagramas de secuencia agregan más información a éstos al mostrar las interacciones entre objetos del sistema.

- Los modelos estructurales indican la organización y arquitectura de un sistema. Los diagramas de clase se usan para definir la estructura estática de clases en un sistema y sus asociaciones.
- Los modelos del comportamiento se usan para describir la conducta dinámica de un sistema en ejecución. Pueden modelarse desde la perspectiva de los datos procesados por el sistema, o mediante los eventos que estimulan respuestas de un sistema.
- Los diagramas de actividad se utilizan para modelar el procesamiento de datos, en que cada actividad representa un paso del proceso.
- Los diagramas de estado se utilizan para modelar el comportamiento de un sistema en respuesta a eventos internos o externos.
- La ingeniería dirigida por modelo es un enfoque al desarrollo del software donde un sistema se representa como un conjunto de modelos que pueden transformarse automáticamente a código ejecutable.

LECTURAS SUGERIDAS

Requirements Analysis and System Design. Este libro se enfoca en el análisis de sistemas de información y examina cómo pueden usarse diferentes modelos UML en el proceso de análisis. (L. Maciaszek, Addison-Wesley, 2001.)

MDA Distilled: Principles of Model-driven Architecture. Se trata de una introducción concisa y accesible al método MDA. Está escrito por autores entusiastas, de modo que con este enfoque el libro dice muy poco acerca de posibles problemas. (S. J. Mellor, K. Scott y D. Weise, Addison-Wesley, 2004.)

Using UML: Software Engineering with Objects and Components, 2nd ed. Una breve introducción legible al uso del UML en la especificación y el diseño de sistemas. Éste es un excelente libro para comprender y el UML, aunque no tiene una descripción completa de la notación. (P. Stevens con R. Pooley, Addison-Wesley, 2006.)

EJERCICIOS

5.1. Explique por qué es importante modelar el contexto de un sistema que se desarrollará. Mencione dos ejemplos de posibles errores que surgirían si los ingenieros de software no entienden el contexto del sistema.

5.2. ¿Cómo podría usar un modelo de un sistema que ya existe? Explique por qué no siempre es necesario que un modelo de sistema esté completo y sea correcto. ¿Lo mismo sería cierto si estuviera desarrollando un modelo de un sistema nuevo?

5.3. Se le pide desarrollar un sistema que ayudará con la planeación de eventos y fiestas a gran escala, como bodas, fiestas de graduación, cumpleaños, etcétera. Con un diagrama de actividad, modele el contexto del proceso para tal sistema, que muestre las actividades que hay en la planeación de una fiesta (reservación de local, organización de invitaciones, y más) y los elementos del sistema que se tengan que usar en cada etapa.

5.4. Para el MHC-PMS, proponga un conjunto de casos de uso que ilustren las interacciones entre un médico, que atiende pacientes y prescribe medicamentos y tratamientos, y el MHC-PMS.

5.5. Desarrolle un diagrama de secuencia que muestre las interacciones que hay cuando un estudiante se registra para un curso en una universidad. Los cursos pueden tener matrícula limitada, de modo que el proceso de registro debe incluir la comprobación de qué lugares están disponibles. Suponga que el estudiante accede a un catálogo electrónico de cursos para encontrar los cursos disponibles.

5.6. Busque cuidadosamente cómo se representan los mensajes y buzones en el sistema de correo electrónico que utilice. Modele las clases de objetos que puedan usarse en la implementación del sistema para representar un buzón y un mensaje de correo electrónico.

5.7. Con base en su experiencia con un cajero automático (ATM), dibuje un diagrama de actividad que modele el procesamiento de datos cuando un cliente retira dinero de la máquina.

5.8. Dibuje un diagrama de secuencia para el mismo sistema. Explique por qué debe desarrollar tanto diagramas de actividad como de secuencia, cuando modela el comportamiento de un sistema.

5.9. Dibuje diagramas de estado del software de control para:

- Una lavadora automática con diferentes programas para distintos tipos de ropa.
- El software para un reproductor de DVD.
- Una contestadora telefónica que registre los mensajes entrantes y muestre el número de mensajes aceptados en un display de LEDs. El sistema debe permitir al usuario del teléfono marcar desde cualquier ubicación, escribir una secuencia de números (identificados como tonos) y reproducir cualquier mensaje grabado.

5.10. Usted es administrador de ingeniería de software y su equipo propone que debe usarse la ingeniería dirigida por modelo para desarrollar el sistema nuevo. ¿Qué factores debe tomar en cuenta cuando decide introducir o no este nuevo enfoque al desarrollo de software?

REFERENCIAS

Ambler, S. W. y Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process.* New York: John Wiley & Sons.

Booch, G., Rumbaugh, J. y Jacobson, I. (2005). *The Unified Modeling Language User Guide, 2nd ed.* Boston: Addison-Wesley.

Chen, P. (1976). "The entity relationship model—Towards a unified view of data". *ACM Trans. on Database Systems,* **1** (1), 9–36.

Codd, E. F. (1979). "Extending the database relational model to capture more meaning". *ACM Trans. on Database Systems,* **4** (4), 397–434.

DeMarco, T. (1978). *Structured Analysis and System Specification.* New York: Yourdon Press.

Erickson, J. y Siau, K. (2007). "Theoretical and practical complexity of modeling methods". *Comm. ACM,* **50** (8), 46–51.

Hammer, M. y McLeod, D. (1981). "Database descriptions with SDM: A semantic database model". *ACM Trans. on Database Sys.,* **6** (3), 351–86.

Harel, D. (1987). "Statecharts: A visual formalism for complex systems". *Sci. Comput. Programming,* **8** (3), 231–74.

Harel, D. (1988). "On visual formalisms". *Comm. ACM,* **31** (5), 514–30.

Hull, R. y King, R. (1987). "Semantic database modeling: Survey, applications and research issues". *ACM Computing Surveys,* **19** (3), 201–60.

Jacobson, I., Christerson, M., Jonsson, P. y Overgaard, G. (1993). *Object-Oriented Software Engineering.* Wokingham.: Addison-Wesley.

Kent, S. (2002). "Model-driven engineering". Proc. 3rd Int. Conf. on Integrated Formal Methods, 286–98.

Kleppe, A., Warmer, J. y Bast, W. (2003). *MDA Explained: The Model Driven Architecture—Practice and Promise.* Boston: Addison-Wesley.

Kruchten, P. (1995). "The 4 + 1 view model of architecture". *IEEE Software,* **11** (6), 42–50.

Mellor, S. J. y Balcer, M. J. (2002). *Executable UML.* Boston: Addison-Wesley.

Mellor, S. J., Scott, K. y Weise, D. (2004). *MDA Distilled: Principles of Model-driven Architecture.* Boston: Addison-Wesley.

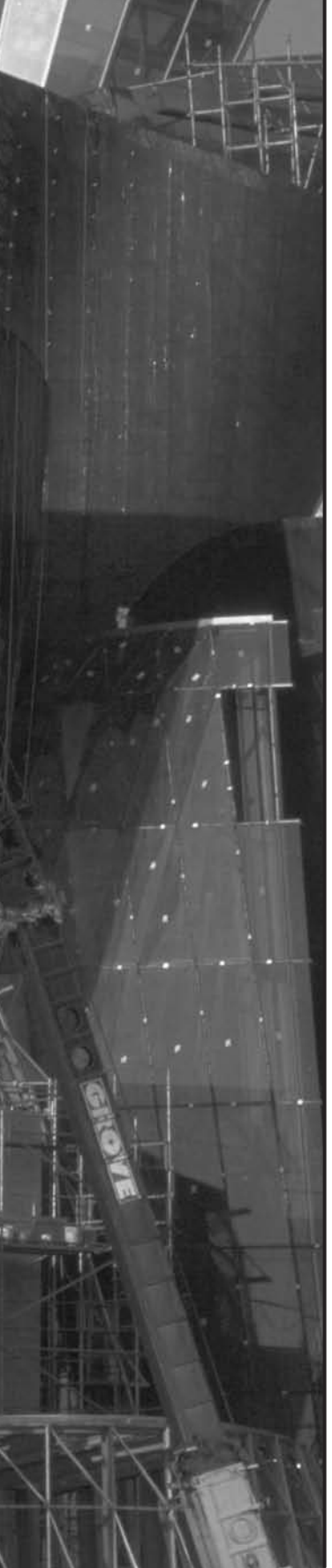
Rumbaugh, J., Jacobson, I. y Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Reading, Mass.: Addison-Wesley.

Rumbaugh, J., Jacobson, I. y Booch, G. (2004). *The Unified Modeling Language Reference Manual, 2nd ed*. Boston: Addison-Wesley.

Schmidt, D. C. (2006). "Model-Driven Engineering". *IEEE Computer,* **39** (2), 25–31.

Stahl, T. y Voelter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.

Ward, P. y Mellor, S. (1985). *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Prentice Hall.



6

Diseño arquitectónico

Objetivos

El objetivo de este capítulo es introducir los conceptos de arquitectura de software y diseño arquitectónico. Al estudiar este capítulo:

- comprenderá por qué es importante el diseño arquitectónico del software;
- conocerá las decisiones que deben tomarse sobre la arquitectura de software durante el proceso de diseño arquitectónico;
- asimilará la idea de los patrones arquitectónicos, formas bien reconocidas de organización de las arquitecturas del sistema, que pueden reutilizarse en los diseños del sistema;
- identificará los patrones arquitectónicos usados frecuentemente en diferentes tipos de sistemas de aplicación, incluidos los sistemas de procesamiento de transacción y los sistemas de procesamiento de lenguaje.

Contenido

6.1 Decisiones en el diseño arquitectónico
6.2 Vistas arquitectónicas
6.3 Patrones arquitectónicos
6.4 Arquitecturas de aplicación

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. En el modelo del proceso de desarrollo de software, como se mostró en el capítulo 2, el diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación.

En los procesos ágiles, por lo general se acepta que una de las primeras etapas en el proceso de desarrollo debe preocuparse por establecer una arquitectura global del sistema. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas. Mientras que la refactorización de componentes en respuesta a los cambios suele ser relativamente fácil, tal vez resulte costoso refactorizar una arquitectura de sistema.

Para ayudar a comprender lo que se entiende por arquitectura del sistema, tome en cuenta la figura 6.1. En ella se presenta un modelo abstracto de la arquitectura para un sistema de robot de empaquetado, que indica los componentes que tienen que desarrollarse. Este sistema robótico empaca diferentes clases de objetos. Usa un componente de visión para recoger los objetos de una banda transportadora, identifica la clase de objeto y selecciona el tipo correcto de empaque. Luego, el sistema mueve los objetos que va a empacar de la banda transportadora de entrega y coloca los objetos empacados en otro transportador. El modelo arquitectónico presenta dichos componentes y los vínculos entre ellos.

En la práctica, hay un significativo traslape entre los procesos de ingeniería de requerimientos y el diseño arquitectónico. De manera ideal, una especificación de sistema no debe incluir cierta información de diseño. Esto no es realista, excepto para sistemas muy pequeños. La descomposición arquitectónica es por lo general necesaria para estructurar y organizar la especificación. Por lo tanto, como parte del proceso de ingeniería de requerimientos, usted podría proponer una arquitectura de sistema abstracta donde se asocien grupos de funciones de sistemas o características con componentes o subsistemas a gran escala. Luego, puede usar esta descomposición para discutir con los participantes sobre los requerimientos y las características del sistema.

Las arquitecturas de software se diseñan en dos niveles de abstracción, que en este texto se llaman *arquitectura en pequeño* y *arquitectura en grande*:

1. La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes. Este capítulo se centra principalmente en arquitecturas de programa.
2. La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen. En los capítulos 18 y 19 se cubren las arquitecturas grandes; en ellos se estudiarán las arquitecturas de los sistemas distribuidos.

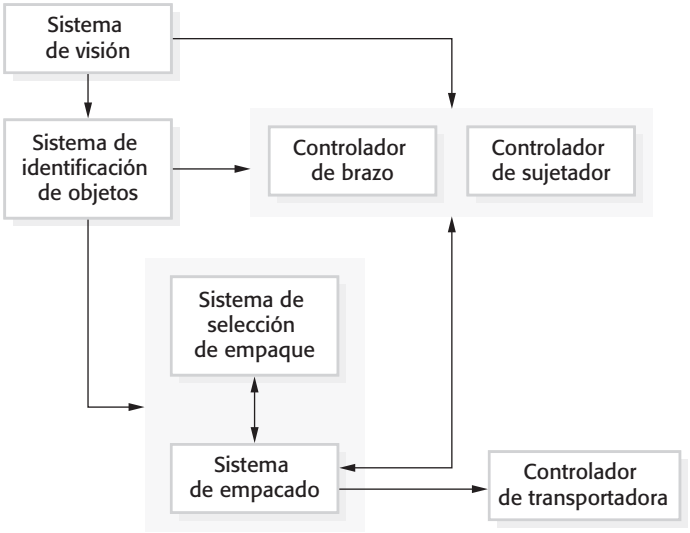


Figura 6.1 Arquitectura de un sistema de control para un robot empacador

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante.

Bass y sus colaboradores (2003) analizan tres ventajas de diseñar y documentar de manera explícita la arquitectura de software:

1. *Comunicación con los participantes* La arquitectura es una presentación de alto nivel del sistema, que puede usarse como un enfoque para la discusión de un amplio número de participantes.
2. *Análisis del sistema* En una etapa temprana en el desarrollo del sistema, aclarar la arquitectura del sistema requiere cierto análisis. Las decisiones de diseño arquitectónico tienen un efecto profundo sobre si el sistema puede o no cubrir requerimientos críticos como rendimiento, fiabilidad y mantenibilidad.
3. *Reutilización a gran escala* Un modelo de una arquitectura de sistema es una descripción corta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. Por lo general, la arquitectura del sistema es la misma para sistemas con requerimientos similares y, por lo tanto, puede soportar reutilización de software a gran escala. Como se explica en el capítulo 16, es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Hofmeister y sus colaboradores (2000) proponen que una arquitectura de software sirve en primer lugar como un plan de diseño para la negociación de requerimientos de sistema y, en segundo lugar, como un medio para establecer discusiones con clientes, desarrolladores y administradores. También sugieren que es una herramienta esencial para la administración de la complejidad. Oculta los detalles y permite a los diseñadores enfocarse en las abstracciones clave del sistema.

Las arquitecturas de sistemas se modelan con frecuencia usando diagramas de bloques simples, como en la figura 6.1. Cada recuadro en el diagrama representa un componente. Los recuadros dentro de recuadros indican que el componente se dividió en subcomponentes. Las flechas significan que los datos y/o señales de control pasan de un componente a otro en la dirección de las flechas. Hay varios ejemplos de este tipo de modelo arquitectónico en el catálogo de arquitectura de software de Booch (Booch, 2009).

Los diagramas de bloque presentan una imagen de alto nivel de la estructura del sistema e incluyen fácilmente a individuos de diferentes disciplinas que intervienen en el proceso de desarrollo del sistema. No obstante su amplio uso, Bass y sus colaboradores (2003) no están de acuerdo con los diagramas de bloque informales para describir una arquitectura. Afirman que tales diagramas informales son representaciones arquitectónicas deficientes, pues no muestran ni el tipo de relaciones entre los componentes del sistema ni las propiedades externamente visibles de los componentes.

Las aparentes contradicciones entre práctica y teoría arquitectónica surgen porque hay dos formas en que se utiliza un modelo arquitectónico de un programa:

1. *Como una forma de facilitar la discusión acerca del diseño del sistema* Una visión arquitectónica de alto nivel de un sistema es útil para la comunicación con los participantes de un sistema y la planeación del proyecto, ya que no se satura con detalles. Los participantes pueden relacionarse con él y entender una visión abstracta del sistema. En tal caso, analizan el sistema como un todo sin confundirse por los detalles. El modelo arquitectónico identifica los componentes clave que se desarrollarán, de modo que los administradores pueden asignar a individuos para planear el desarrollo de dichos sistemas.
2. *Como una forma de documentar una arquitectura que se haya diseñado* La meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes en un sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema.

Los diagramas de bloque son una forma adecuada para describir la arquitectura del sistema durante el proceso de diseño, pues son una buena manera de soportar las comunicaciones entre las personas involucradas en el proceso. En muchos proyectos, suele ser la única documentación arquitectónica que existe. Sin embargo, si la arquitectura de un sistema debe documentarse ampliamente, entonces es mejor usar una notación con semántica bien definida para la descripción arquitectónica. No obstante, tal como se estudia en la sección 6.2, algunas personas consideran que la documentación detallada ni es útil ni vale realmente la pena el costo de su desarrollo.

6.1 Decisiones en el diseño arquitectónico

El diseño arquitectónico es un proceso creativo en el cual se diseña una organización del sistema que cubrirá los requerimientos funcionales y no funcionales de éste. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades.

Durante el proceso de diseño arquitectónico, los arquitectos del sistema deben tomar algunas decisiones estructurales que afectarán profundamente el sistema y su proceso de desarrollo. Con base en su conocimiento y experiencia, deben considerar las siguientes preguntas fundamentales sobre el sistema:

1. ¿Existe alguna arquitectura de aplicación genérica que actúe como plantilla para el sistema que se está diseñando?
2. ¿Cómo se distribuirá el sistema a través de algunos núcleos o procesadores?
3. ¿Qué patrones o estilos arquitectónicos pueden usarse?
4. ¿Cuál será el enfoque fundamental usado para estructurar el sistema?
5. ¿Cómo los componentes estructurales en el sistema se separarán en subcomponentes?
6. ¿Qué estrategia se usará para controlar la operación de los componentes en el sistema?
7. ¿Cuál organización arquitectónica es mejor para entregar los requerimientos no funcionales del sistema?
8. ¿Cómo se evaluará el diseño arquitectónico?
9. ¿Cómo se documentará la arquitectura del sistema?

Aunque cada sistema de software es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Por ejemplo, las líneas de producto de aplicación son aplicaciones que se construyen en torno a una arquitectura central con variantes que cubren requerimientos específicos del cliente. Cuando se diseña una arquitectura de sistema, debe decidirse qué tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se puede reutilizar de dichas arquitecturas de aplicación. En la sección 6.4 se estudian las arquitecturas de aplicación genéricas y en el capítulo 16 las líneas de producto de aplicación.

Para sistemas embebidos y sistemas diseñados para computadoras personales, por lo general, hay un solo procesador y no tendrá que diseñar una arquitectura distribuida para el sistema. Sin embargo, los sistemas más grandes ahora son sistemas distribuidos donde el software de sistema se distribuye a través de muchas y diferentes computadoras. La elección de arquitectura de distribución es una decisión clave que afecta el rendimiento y

la fiabilidad del sistema. Éste es un tema importante por derecho propio y se trata por separado en el capítulo 18.

La arquitectura de un sistema de software puede basarse en un patrón o un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema (Garlan y Shaw, 1993), tal como una organización cliente-servidor o una arquitectura por capas. Los patrones arquitectónicos captan la esencia de una arquitectura que se usó en diferentes sistemas de software. Usted tiene que conocer tanto los patrones comunes, en que éstos se usen, como sus fortalezas y debilidades cuando se tomen decisiones sobre la arquitectura de un sistema. En la sección 6.3 se analizan algunos patrones de uso frecuente.

La noción de un estilo arquitectónico de Garlan y Shaw (estilo y patrón llegaron a significar lo mismo) cubre las preguntas 4 a 6 de la lista anterior. Es necesario elegir la estructura más adecuada, como cliente-servidor o estructura en capas, que le permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural, usted opta por la estrategia de separar los componentes en subcomponentes. Los enfoques que pueden usarse permiten la implementación de diferentes tipos de arquitectura. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de componentes. Se desarrolla un modelo general de las relaciones de control entre las diferentes partes del sistema.

Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónicos particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales:

1. *Rendimiento* Si el rendimiento es un requerimiento crítico, la arquitectura debe diseñarse para localizar operaciones críticas dentro de un pequeño número de componentes, con todos estos componentes desplegados en la misma computadora en vez de distribuirlos por la red. Esto significaría usar algunos componentes relativamente grandes, en vez de pequeños componentes de grano fino, lo cual reduce el número de comunicaciones entre componentes. También puede considerar organizaciones del sistema en tiempo de operación que permitan a éste ser replicable y ejecutable en diferentes procesadores.

2. *Seguridad* Si la seguridad es un requerimiento crítico, será necesario usar una estructura en capas para la arquitectura, con los activos más críticos protegidos en las capas más internas, y con un alto nivel de validación de seguridad aplicado a dichas capas.

3. *Protección* Si la protección es un requerimiento crítico, la arquitectura debe diseñarse de modo que las operaciones relacionadas con la protección se ubiquen en algún componente individual o en un pequeño número de componentes. Esto reduce los costos y problemas de validación de la protección, y hace posible ofrecer sistemas de protección relacionados que, en caso de falla, desactiven con seguridad el sistema.

4. *Disponibilidad* Si la disponibilidad es un requerimiento crítico, la arquitectura tiene que diseñarse para incluir componentes redundantes de manera que sea posible sustituir y actualizar componentes sin detener el sistema. En el capítulo 13 se describen dos arquitecturas de sistema tolerantes a fallas en sistemas de alta disponibilidad.

5. *Mantenibilidad* Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debe diseñarse usando componentes autocontenidos de grano fino que

puedan cambiarse con facilidad. Los productores de datos tienen que separarse de los consumidores y hay que evitar compartir las estructuras de datos.

Evidentemente, hay un conflicto potencial entre algunas de estas arquitecturas. Por ejemplo, usar componentes grandes mejora el rendimiento, y utilizar componentes pequeños de grano fino aumenta la mantenibilidad. Si tanto el rendimiento como la mantenibilidad son requerimientos importantes del sistema, entonces debe encontrarse algún compromiso. Esto en ocasiones se logra usando diferentes patrones o estilos arquitectónicos para distintas partes del sistema.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba de una arquitectura es qué tan bien el sistema cubre sus requerimientos funcionales y no funcionales cuando está en uso. Sin embargo, es posible hacer cierta evaluación al comparar el diseño contra arquitecturas de referencia o patrones arquitectónicos genéricos. Para ayudar con la evaluación arquitectónica, también puede usarse la descripción de Bosch (2000) de las características no funcionales de los patrones arquitectónicos.

6.2 Vistas arquitectónicas

En la introducción a este capítulo se explicó que los modelos arquitectónicos de un sistema de software sirven para enfocar la discusión sobre los requerimientos o el diseño del software. De manera alternativa, pueden emplearse para documentar un diseño, de modo que se usen como base en el diseño y la implementación más detallados, así como en la evolución futura del sistema. En esta sección se estudian dos temas que son relevantes para ambos:

1. ¿Qué vistas o perspectivas son útiles al diseñar y documentar una arquitectura del sistema?
2. ¿Qué notaciones deben usarse para describir modelos arquitectónicos?

Es imposible representar toda la información relevante sobre la arquitectura de un sistema en un solo modelo arquitectónico, ya que cada uno presenta únicamente una vista o perspectiva del sistema. Ésta puede mostrar cómo un sistema se descompone en módulos, cómo interactúan los procesos de tiempo de operación o las diferentes formas en que los componentes del sistema se distribuyen a través de una red. Todo ello es útil en diferentes momentos de manera que, para el diseño y la documentación, por lo general se necesita presentar múltiples vistas de la arquitectura de software.

Existen diferentes opiniones relativas a qué vistas se requieren. Krutchen (1995), en su bien conocido modelo de vista 4+1 de la arquitectura de software, sugiere que deben existir cuatro vistas arquitectónicas fundamentales, que se relacionan usando casos de uso o escenarios. Las vistas que él sugiere son:

1. Una vista lógica, que indique las abstracciones clave en el sistema como objetos o clases de objeto. En este tipo de vista se tienen que relacionar los requerimientos del sistema con entidades.

2. Una vista de proceso, que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad.
3. Una vista de desarrollo, que muestre cómo el software está descompuesto para su desarrollo, esto es, indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o equipo de desarrollo. Esta vista es útil para administradores y programadores de software.
4. Una vista física, que exponga el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema. Esta vista es útil para los ingenieros de sistemas que planean una implementación de sistema.

Hofmeister y sus colaboradores (2000) sugieren el uso de vistas similares, pero a éstas agregan la noción de vista conceptual. Esta última es una vista abstracta del sistema que puede ser la base para descomponer los requerimientos de alto nivel en especificaciones más detalladas, ayudar a los ingenieros a tomar decisiones sobre componentes que puedan reutilizarse, y representar una línea de producto (que se estudia en el capítulo 16) en vez de un solo sistema. La figura 6.1, que describe la arquitectura de un robot de empacado, es un ejemplo de una vista conceptual del sistema.

En la práctica, las vistas conceptuales casi siempre se desarrollan durante el proceso de diseño y se usan para apoyar la toma de decisiones arquitectónicas. Son una forma de comunicar a diferentes participantes la esencia de un sistema. Durante el proceso de diseño, también pueden desarrollarse algunas de las otras vistas, al tiempo que se discuten diferentes aspectos del sistema, aunque no haya necesidad de una descripción completa desde todas las perspectivas. Además se podrían asociar patrones arquitectónicos, estudiados en la siguiente sección, con las diferentes vistas de un sistema.

Hay diferentes opiniones respecto de si los arquitectos de software deben o no usar el UML para una descripción arquitectónica (Clements *et al*., 2002). Un estudio en 2006 (Lange *et al*., 2006) demostró que, cuando se usa el UML, se aplica principalmente en una forma holgada e informal. Los autores de dicho ensayo argumentan que esto era incorrecto. El autor no está de acuerdo con esta visión. El UML se diseñó para describir sistemas orientados a objetos y, en la etapa de diseño arquitectónico, uno quiere describir con frecuencia sistemas en un nivel superior de abstracción. Las clases de objetos están muy cerca de la implementación, como para ser útiles en la descripción arquitectónica.

Para el autor, el UML no es útil durante el proceso de diseño en sí y prefiere notaciones informales que sean más rápidas de escribir y puedan dibujarse fácilmente en un pizarrón. El UML es de más valor cuando se documenta una arquitectura a detalle o se usa un desarrollo dirigido por modelo, como se estudió en el capítulo 5.

Algunos investigadores proponen el uso de lenguajes de descripción arquitectónica (ADL, por las siglas de *Architectural Description Languages*) más especializados (Bass *et al*., 2003) para describir arquitecturas del sistema. Los elementos básicos de los ADL son componentes y conectores, e incluyen reglas y lineamientos para arquitecturas bien formadas. Sin embargo, debido a su naturaleza especializada, los expertos de dominio y aplicación tienen dificultad para entender y usar los ADL. Esto dificulta la valoración de su utilidad para la ingeniería práctica del software. Los ADL diseñados para un dominio particular (por ejemplo, sistemas automotores) pueden usarse como una base

Nombre	MVC (modelo de vista del controlador)
Descripción	Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El componente Modelo maneja los datos del sistema y las operaciones asociadas a esos datos. El componente Vista define y gestiona cómo se presentan los datos al usuario. El componente Controlador dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo. Véase la figura 6.3.
Ejemplo	La figura 6.4 muestra la arquitectura de un sistema de aplicación basado en la Web, que se organiza con el uso del patrón MVC.
Cuándo se usa	Se usa cuando existen múltiples formas de ver e interactuar con los datos. También se utiliza al desconocerse los requerimientos futuros para la interacción y presentación.
Ventajas	Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos.
Desventajas	Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

Figura 6.2 Patrón modelo de vista del controlador (MVC)

para el desarrollo dirigido por modelo. Sin embargo, se considera que los modelos y las notaciones informales, como el UML, seguirán siendo las formas de uso más común para documentar las arquitecturas del sistema.

Los usuarios de métodos ágiles afirman que, por lo general, no se utiliza la documentación detallada del diseño. Por lo tanto, desarrollarla es un desperdicio de tiempo y dinero. El autor está en gran medida de acuerdo con esta visión y considera que, para la mayoría de los sistemas, no vale la pena desarrollar una descripción arquitectónica detallada desde estas cuatro perspectivas. Uno debe desarrollar las vistas que sean útiles para la comunicación sin preocuparse si la documentación arquitectónica está completa o no. Sin embargo, una excepción es al desarrollar sistemas críticos, cuando es necesario realizar un análisis de confiabilidad detallado del sistema. Tal vez se deba convencer a reguladores externos de que el sistema se hizo conforme a sus regulaciones y, en consecuencia, puede requerirse una documentación arquitectónica completa.

6.3 Patrones arquitectónicos

La idea de los patrones como una forma de presentar, compartir y reutilizar el conocimiento sobre los sistemas de software se usa ahora ampliamente. El origen de esto fue la publicación de un libro acerca de patrones de diseño orientados a objetos (Gamma *et al*., 1995), que incitó el desarrollo de otros tipos de patrón, como los patrones para el diseño organizacional (Coplien y Harrison, 2004), patrones de usabilidad (Usability Group, 1998), interacción (Martin y Sommerville, 2004), administración de la configuración

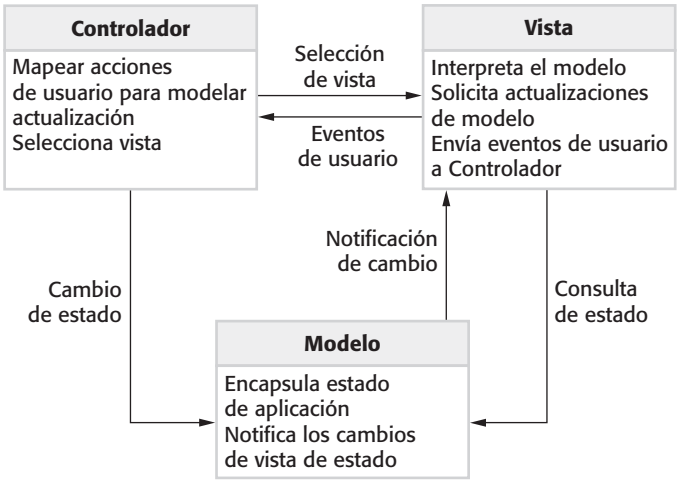


Figura 6.3 La organización del MVC

(Berczuk y Appleton, 2002), etcétera. Los patrones arquitectónicos se propusieron en la década de 1990, con el nombre de "estilos arquitectónicos" (Shaw y Garlan, 1996), en una serie de cinco volúmenes de manuales sobre arquitectura de software orientada a patrones, publicados entre 1996 y 2007 (Buschmann *et al*., 1996; Buschmann *et al*., 2007a; Buschmann *et al*., 2007b; Kircher y Jain, 2004; Schmidt *et al*., 2000).

En esta sección se introducen los patrones arquitectónicos y se describe brevemente una selección de patrones arquitectónicos de uso común en diferentes tipos de sistemas. Para más información de patrones y su uso, debe remitirse a los manuales de patrones publicados.

Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. De este modo, un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como sobre las fortalezas y debilidades del patrón.

Por ejemplo, la figura 6.2 describe el muy conocido patrón Modelo-Vista-Controlador. Este patrón es el soporte del manejo de la interacción en muchos sistemas basados en la Web. La descripción del patrón estilizado incluye el nombre del patrón, una breve descripción (con un modelo gráfico asociado) y un ejemplo del tipo de sistema donde se usa el patrón (de nuevo, quizá con un modelo gráfico). También debe incluir información sobre cuándo hay que usar el patrón, así como sobre sus ventajas y desventajas. En las figuras 6.3 y 6.4 se presentan los modelos gráficos de la arquitectura asociada con el patrón MVC. En ellas se muestra la arquitectura desde diferentes vistas: la figura 6.3 es una vista conceptual; en tanto que la figura 6.4 ilustra una posible arquitectura en tiempo de operación, cuando este patrón se usa para el manejo de la interacción en un sistema basado en la Web.

En una breve sección de un capítulo general es imposible describir todos los patrones genéricos que se usan en el desarrollo de software. En cambio, se presentan algunos ejemplos seleccionados de patrones que se utilizan ampliamente y captan buenos principios de diseño arquitectónico. En las páginas Web del libro se incluyen más ejemplos acerca de patrones arquitectónicos genéricos.

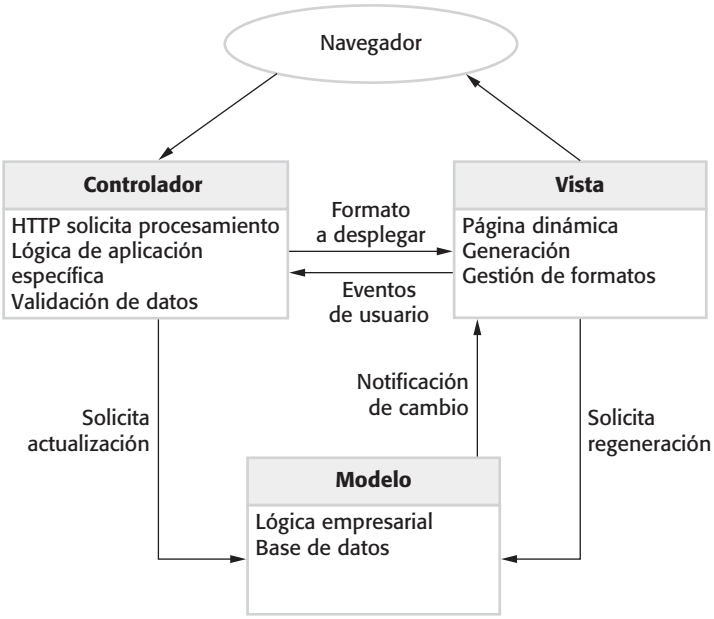


Figura 6.4 Arquitectura de aplicación Web con el patrón MVC

6.3.1 Arquitectura en capas

Las nociones de separación e independencia son fundamentales para el diseño arquitectónico porque permiten localizar cambios. El patrón MVC, que se muestra en la figura 6.2, separa elementos de un sistema, permitiéndoles cambiar de forma independiente. Por ejemplo, agregar una nueva vista o cambiar una vista existente puede hacerse sin modificación alguna a los datos subyacentes en el modelo. El patrón de arquitectura en capas es otra forma de lograr separación e independencia. Este patrón se ilustra en la figura 6.5. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada una se apoya sólo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella.

Este enfoque en capas soporta el desarrollo incremental de sistemas. Conforme se desarrolla una capa, algunos de los servicios proporcionados por esta capa deben quedar a disposición de los usuarios. La arquitectura también es cambiable y portátil. En tanto su interfaz no varíe, una capa puede sustituirse por otra equivalente. Más aún, cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente. A medida que los sistemas en capas localizan dependencias de máquina en capas más internas, se facilita el ofrecimiento de implementaciones multiplataforma de un sistema de aplicación. Sólo las capas más internas dependientes de la máquina deben reimplantarse para considerar las facilidades de un sistema operativo o base de datos diferentes.

La figura 6.6 es un ejemplo de una arquitectura en capas con cuatro capas. La capa inferior incluye software de soporte al sistema, por lo general soporte de base de datos y sistema operativo. La siguiente capa es la de aplicación, que comprende los componentes relacionados con la funcionalidad de la aplicación, así como los componentes de utilidad que usan otros componentes de aplicación. La tercera capa se relaciona con la gestión de interfaz del usuario y con brindar autenticación y autorización al usuario, mientras que la

Nombre	Arquitectura en capas
Descripción	Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima, de modo que las capas de nivel inferior representan servicios núcleo que es probable se utilicen a lo largo de todo el sistema. Véase la figura 6.6.
Ejemplo	Un modelo en capas de un sistema para compartir documentos con derechos de autor se tiene en diferentes bibliotecas, como se ilustra en la figura 6.7.
Cuándo se usa	Se usa al construirse nuevas facilidades encima de los sistemas existentes; cuando el desarrollo se dispersa a través de varios equipos de trabajo, y cada uno es responsable de una capa de funcionalidad; cuando exista un requerimiento para seguridad multinivel.
Ventajas	Permite la sustitución de capas completas en tanto se conserve la interfaz. Para aumentar la confiabilidad del sistema, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación).
Desventajas	En la práctica, suele ser difícil ofrecer una separación limpia entre capas, y es posible que una capa de nivel superior deba interactuar directamente con capas de nivel inferior, en vez de que sea a través de la capa inmediatamente abajo de ella. El rendimiento suele ser un problema, debido a múltiples niveles de interpretación de una solicitud de servicio mientras se procesa en cada capa.

Figura 6.5 Patrón de arquitectura en capas

capa superior proporciona facilidades de interfaz de usuario. Desde luego, es arbitrario el número de capas. Cualquiera de las capas en la figura 6.6 podría dividirse en dos o más capas.

La figura 6.7 es un ejemplo de cómo puede aplicarse este patrón de arquitectura en capas a un sistema de biblioteca llamado LIBSYS, que permite el acceso electrónico controlado a material con derechos de autor de un conjunto de bibliotecas universitarias. Tiene una arquitectura de cinco capas y, en la capa inferior, están las bases de datos individuales en cada biblioteca.

En la figura 6.17 (que se encuentra en la sección 6.4) se observa otro ejemplo de patrón de arquitectura en capas. Muestra la organización del sistema para atención a la salud mental (MHC-PMS) que se estudió en capítulos anteriores.

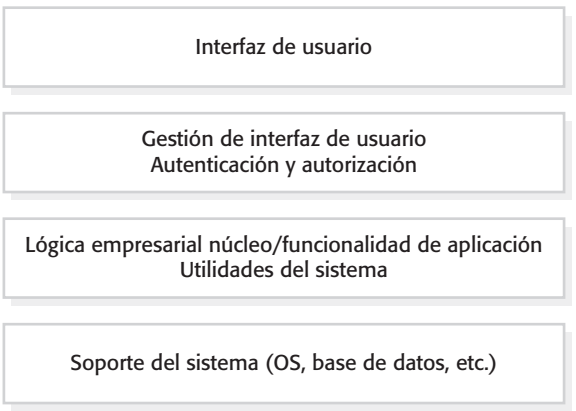


Figura 6.6 Arquitectura genérica en capas

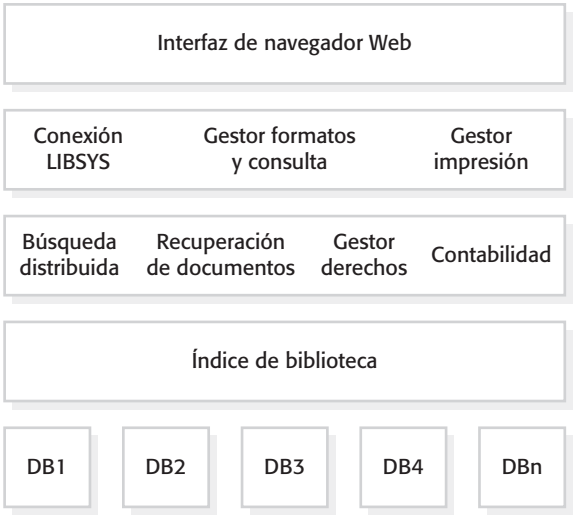


Figura 6.7 Arquitectura del sistema LIBSYS

6.3.2 Arquitectura de repositorio

Los patrones de arquitectura en capas y MVC son ejemplos de patrones en que la vista presentada es la organización conceptual de un sistema. El siguiente ejemplo, el patrón de repositorio (figura 6.8), describe cómo comparte datos un conjunto de componentes en interacción.

La mayoría de los sistemas que usan grandes cantidades de datos se organizan sobre una base de datos o un repositorio compartido. Por lo tanto, este modelo es adecuado

Figura 6.8 El patrón de repositorio

Nombre	Repositorio
Descripción	Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan sólo a través del repositorio.
Ejemplo	La figura 6.9 es un ejemplo de un IDE donde los componentes usan un repositorio de información de diseño de sistema. Cada herramienta de software genera información que, en ese momento, está disponible para uso de otras herramientas.
Cuándo se usa	Este patrón se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. También puede usarse en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active una acción o herramienta.
Ventajas	Los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de manera consistente (por ejemplo, respaldos realizados al mismo tiempo), pues todos están en un lugar.
Desventajas	El repositorio es un punto de falla único, de modo que los problemas en el repositorio afectan a todo el sistema. Es posible que haya ineficiencias al organizar toda la comunicación a través del repositorio. Quizá sea difícil distribuir el repositorio por medio de varias computadoras.

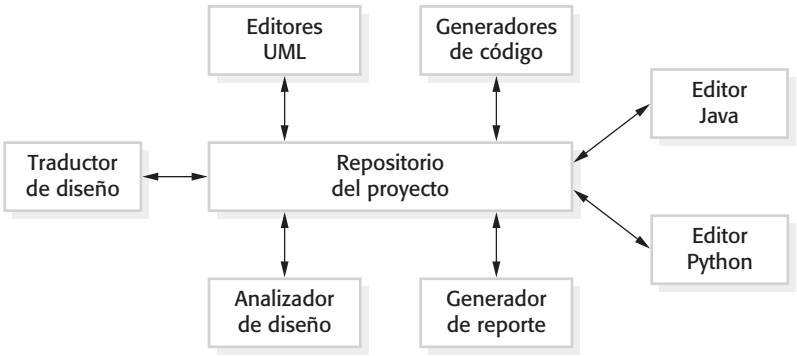


Figura 6.9 Arquitectura de repositorio para un IDE

para aplicaciones en las que un componente genere datos y otro los use. Los ejemplos de este tipo de sistema incluyen sistemas de comando y control, sistemas de información administrativa, sistemas CAD y entornos de desarrollo interactivo para software.

La figura 6.9 ilustra una situación en la que puede usarse un repositorio. Este diagrama muestra un IDE que incluye diferentes herramientas para soportar desarrollo dirigido por modelo. En este caso, el repositorio puede ser un entorno controlado por versión (como se estudia en el capítulo 25) que hace un seguimiento de los cambios al software y permite regresar (*rollback*) a versiones anteriores.

Organizar herramientas alrededor de un repositorio es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir explícitamente datos de un componente a otro. Sin embargo, los componentes deben operar en torno a un modelo de repositorio de datos acordado. Inevitablemente, éste es un compromiso entre las necesidades específicas de cada herramienta y sería difícil o imposible integrar nuevos componentes, si sus modelos de datos no se ajustan al esquema acordado. En la práctica, llega a ser complicado distribuir el repositorio sobre un número de máquinas. Aunque es posible distribuir un repositorio lógicamente centralizado, puede haber problemas con la redundancia e inconsistencia de los datos.

En el ejemplo que se muestra en la figura 6.9, el repositorio es pasivo, y el control es responsabilidad de los componentes que usan el repositorio. Un enfoque alternativo, que se derivó para sistemas IA, utiliza un modelo “blackboard” (pizarrón) que activa componentes cuando los datos particulares se tornan disponibles. Esto es adecuado cuando la forma de los datos del repositorio está menos estructurada. Las decisiones sobre cuál herramienta activar puede hacerse sólo cuando se hayan analizado los datos. Este modelo lo introdujo Nii (1986). Bosch (2000) incluye un buen análisis de cómo este estilo se relaciona con los atributos de calidad del sistema.

6.3.3 Arquitectura cliente-servidor

El patrón de repositorio se interesa por la estructura estática de un sistema sin mostrar su organización en tiempo de operación. El siguiente ejemplo ilustra una organización en tiempo de operación, de uso muy común para sistemas distribuidos. En la figura 6.10 se describe el patrón cliente-servidor.

Nombre	Cliente-servidor
Descripción	En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
Ejemplo	La figura 6.11 es un ejemplo de una filmoteca y videoteca (videos/DVD) organizada como un sistema cliente-servidor.
Cuándo se usa	Se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. Como los servidores se pueden replicar, también se usan cuando la carga de un sistema es variable.
Ventajas	La principal ventaja de este modelo es que los servidores se pueden distribuir a través de una red. La funcionalidad general (por ejemplo, un servicio de impresión) estaría disponible a todos los clientes, así que no necesita implementarse en todos los servicios.
Desventajas	Cada servicio es un solo punto de falla, de modo que es susceptible a ataques de rechazo de servicio o a fallas del servidor. El rendimiento resultará impredecible porque depende de la red, así como del sistema. Quizás haya problemas administrativos cuando los servidores sean propiedad de diferentes organizaciones.

Figura 6.10 Patrón cliente-servidor

Un sistema que sigue el patrón cliente-servidor se organiza como un conjunto de servicios y servidores asociados, y de clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

1. Un conjunto de servidores que ofrecen servicios a otros componentes. Ejemplos de éstos incluyen servidores de impresión; servidores de archivo que brindan servicios de administración de archivos, y un servidor compilador, que proporciona servicios de compilación de lenguaje de programación.
2. Un conjunto de clientes que solicitan los servicios que ofrecen los servidores. Habrá usualmente varias instancias de un programa cliente que se ejecuten de manera concurrente en diferentes computadoras.
3. Una red que permite a los clientes acceder a dichos servicios. La mayoría de los sistemas cliente-servidor se implementan como sistemas distribuidos, conectados mediante protocolos de Internet.

Las arquitecturas cliente-servidor se consideran a menudo como arquitecturas de sistemas distribuidos; sin embargo, el modelo lógico de servicios independientes que opera en servidores separados puede implementarse en una sola computadora. De nuevo, un beneficio importante es la separación e independencia. Los servicios y servidores pueden cambiar sin afectar otras partes del sistema.

Es posible que los clientes deban conocer los nombres de los servidores disponibles, así como los servicios que proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes acceden a sus servicios. Los clientes acceden a los servicios que proporciona un servidor, a través de llamadas a procedimiento remoto usando un protocolo solicitud-respuesta, como el protocolo http utilizado

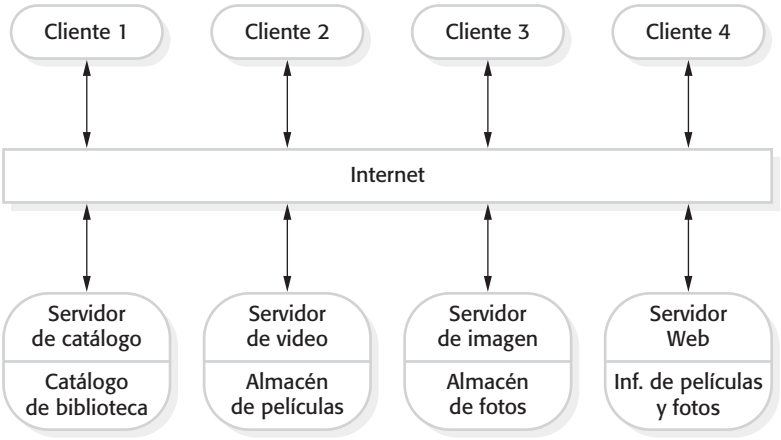


Figura 6.11 Arquitectura cliente-servidor para una filmoteca

en la WWW. En esencia, un cliente realiza una petición a un servidor y espera hasta que recibe una respuesta.

La figura 6.11 es un ejemplo de un sistema que se basa en el modelo cliente-servidor. Se trata de un sistema multiusuario basado en la Web, para ofrecer un repertorio de películas y fotografías. En este sistema, varios servidores manejan y despliegan los diferentes tipos de medios. Los cuadros de video necesitan transmitirse rápidamente y en sincronía, aunque a una resolución relativamente baja. Tal vez estén comprimidos en un almacén, de manera que el servidor de video puede manipular en diferentes formatos la compresión y descompresión del video. Sin embargo, las imágenes fijas deben conservarse en una resolución alta, por lo que es adecuado mantenerlas en un servidor independiente.

El catálogo debe manejar una variedad de consultas y ofrecer vínculos hacia el sistema de información Web, que incluye datos acerca de las películas y los videos, así

Figura 6.12 Patrón de tubería y filtro (*pipe and filter*)

Nombre	Tubería y filtro (*pipe and filter*)
Descripción	El procesamiento de datos en un sistema se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos. Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento.
Ejemplo	La figura 6.13 es un ejemplo de un sistema de tubería y filtro usado para el procesamiento de facturas.
Cuándo se usa	Se suele utilizar en aplicaciones de procesamiento de datos (tanto basadas en lotes [*batch*] como en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas.
Ventajas	Fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como un sistema secuencial o como uno concurrente.
Desventajas	El formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

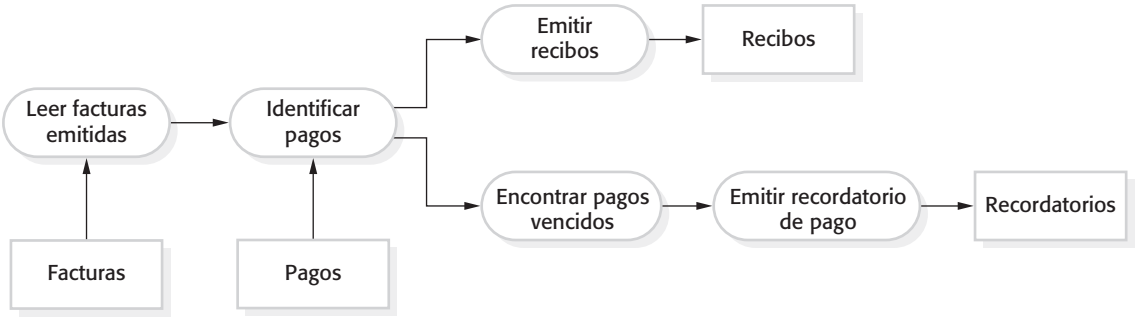


Figura 6.13 Ejemplo de arquitectura de tubería y filtro

como un sistema de comercio electrónico que soporte la venta de fotografías, películas y videos. El programa cliente es simplemente una interfaz integrada de usuario, construida mediante un navegador Web, para acceder a dichos servicios.

La ventaja más importante del modelo cliente-servidor consiste en que es una arquitectura distribuida. Éste puede usarse de manera efectiva en sistemas en red con distintos procesadores distribuidos. Es fácil agregar un nuevo servidor e integrarlo al resto del sistema, o bien, actualizar de manera clara servidores sin afectar otras partes del sistema. En el capítulo 18 se estudian las arquitecturas distribuidas, incluidas las arquitecturas cliente-servidor y las arquitecturas de objeto distribuidas.

6.3.4 Arquitectura de tubería y filtro

El ejemplo final de un patrón arquitectónico es el patrón tubería y filtro (*pipe and filter*). Éste es un modelo de la organización en tiempo de operación de un sistema, donde las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de uno a otro y se transforman conforme se desplazan a través de la secuencia. Cada paso de procesamiento se implementa como un transformador. Los datos de entrada fluyen por medio de dichos transformadores hasta que se convierten en salida. Las transformaciones pueden ejecutarse secuencialmente o en forma paralela. Es posible que los datos se procesen por cada transformador ítem por ítem o en un solo lote.

El nombre "tubería y filtro" proviene del sistema Unix original, donde era posible vincular procesos empleando "tuberías". Por ellas pasaba una secuencia de texto de un proceso a otro. Los sistemas conformados con este modelo pueden implementarse al combinar comandos Unix, usando las tuberías y las instalaciones de control del intérprete de comandos Unix. Se usa el término "filtro" porque una transformación "filtra" los datos que puede procesar de su secuencia de datos de entrada.

Se han utilizado variantes de este patrón desde que se usaron por primera vez computadoras para el procesamiento automático de datos. Cuando las transformaciones son secuenciales, con datos procesados en lotes, este modelo arquitectónico de tubería y filtro se convierte en un modelo secuencial en lote, una arquitectura común para sistemas de procesamiento de datos (por ejemplo, un sistema de facturación). La arquitectura de un sistema embebido puede organizarse también como un proceso por entubamiento, donde cada proceso se ejecuta de manera concurrente. En el capítulo 20 se estudia el uso de este patrón en sistemas embebidos.

En la figura 6.13 se muestra un ejemplo de este tipo de arquitectura de sistema, que se usa en una aplicación de procesamiento en lote. Una organización emite facturas a los clientes. Una vez a la semana, los pagos efectuados se incorporan a las facturas. Para

Patrones arquitectónicos para control

Existen patrones arquitectónicos específicos que reflejan formas usadas comúnmente para organizar el control en un sistema. En ellos se incluyen el control centralizado, basado en un componente que llama otros componentes, y el control con base en un evento, donde el sistema reacciona a eventos externos.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

las facturas pagadas se emite un recibo. Para las facturas no saldadas dentro del plazo de pago se emite un recordatorio.

Los sistemas interactivos son difíciles de escribir con el modelo tubería y filtro, debido a la necesidad de procesar una secuencia de datos. Aunque las entradas y salidas textuales simples pueden modelarse de esta forma, las interfaces gráficas de usuario tienen formatos I/O más complejos, así como una estrategia de control que se basa en eventos como clics del mouse o selecciones del menú. Es difícil traducir esto en una forma compatible con el modelo *pipelining* (entubamiento).

6.4 Arquitecturas de aplicación

Los sistemas de aplicación tienen la intención de cubrir las necesidades de una empresa u organización. Todas las empresas tienen mucho en común: necesitan contratar personal, emitir facturas, llevar la contabilidad, etcétera. Las empresas que operan en el mismo sector usan aplicaciones comunes específicas para el sector. De esta forma, además de las funciones empresariales generales, todas las compañías telefónicas necesitan sistemas para conectar llamadas, administrar sus redes y emitir facturas a los clientes, entre otros. En consecuencia, también los sistemas de aplicación que utilizan dichas empresas tienen mucho en común.

Estos factores en común condujeron al desarrollo de arquitecturas de software que describen la estructura y la organización de tipos particulares de sistemas de software. Las arquitecturas de aplicación encapsulan las principales características de una clase de sistemas. Por ejemplo, en los sistemas de tiempo real puede haber modelos arquitectónicos genéricos de diferentes tipos de sistema, tales como sistemas de recolección de datos o sistemas de monitorización. Aunque las instancias de dichos sistemas difieren en detalle, la estructura arquitectónica común puede reutilizarse cuando se desarrollen nuevos sistemas del mismo tipo.

La arquitectura de aplicación puede reimplantarse cuando se desarrollen nuevos sistemas, pero, para diversos sistemas empresariales, la reutilización de aplicaciones es posible sin reimplementación. Esto se observa en el crecimiento de los sistemas de planeación de recursos empresariales (ERP, por las siglas de *Enterprise Resource Planning*) de compañías como SAP y Oracle, y paquetes de software vertical (COTS) para aplicaciones especializadas en diferentes áreas de negocios. En dichos sistemas, un sistema genérico se configura y adapta para crear una aplicación empresarial específica.

Arquitecturas de aplicación

En el sitio Web del libro existen muchos ejemplos de arquitecturas de aplicación. Se incluyen descripciones de sistemas de procesamiento de datos en lote, sistemas de asignación de recursos y sistemas de edición basados en eventos.

http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/

Por ejemplo, un sistema para suministrar la administración en cadena se adapta a diferentes tipos de proveedores, bienes y arreglos contractuales.

Como diseñador de software, usted puede usar modelos de arquitecturas de aplicación en varias formas:

1. *Como punto de partida para el proceso de diseño arquitectónico* Si no está familiarizado con el tipo de aplicación que desarrolla, podría basar su diseño inicial en una arquitectura de aplicación genérica. Desde luego, ésta tendrá que ser especializada para el sistema específico que se va a desarrollar, pero es un buen comienzo para el diseño.
2. *Como lista de verificación del diseño* Si usted desarrolló un diseño arquitectónico para un sistema de aplicación, puede comparar éste con la arquitectura de aplicación genérica y luego, verificar que su diseño sea consistente con la arquitectura genérica.
3. *Como una forma de organizar el trabajo del equipo de desarrollo* Las arquitecturas de aplicación identifican características estructurales estables de las arquitecturas del sistema y, en muchos casos, es posible desarrollar éstas en paralelo. Puede asignar trabajo a los miembros del grupo para implementar diferentes componentes dentro de la arquitectura.
4. *Como un medio para valorar los componentes a reutilizar* Si tiene componentes por reutilizar, compare éstos con las estructuras genéricas para saber si existen componentes similares en la arquitectura de aplicación.
5. *Como un vocabulario para hablar acerca de los tipos de aplicaciones* Si discute acerca de una aplicación específica o trata de comparar aplicaciones del mismo tipo, entonces puede usar los conceptos identificados en la arquitectura genérica para hablar sobre las aplicaciones.

Hay muchos tipos de sistema de aplicación y, en algunos casos, parecerían muy diferentes. Sin embargo, muchas de estas aplicaciones distintas superficialmente en realidad tienen mucho en común y, por ende, suelen representarse mediante una sola arquitectura de aplicación abstracta. Esto se ilustra aquí al describir las siguientes arquitecturas de dos tipos de aplicación:

1. *Aplicaciones de procesamiento de transacción* Este tipo de aplicaciones son aplicaciones centradas en bases de datos, que procesan los requerimientos del usuario mediante la información y actualizan ésta en una base de datos. Se trata del tipo más común de sistemas empresariales interactivos. Se organizan de tal forma que las acciones del usuario no pueden interferir unas con otras y se mantiene la integridad de la base

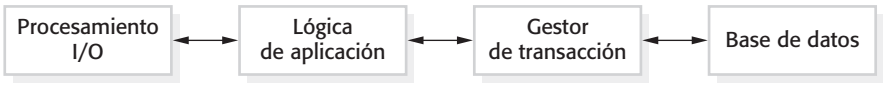


Figura 6.14 Estructura de las aplicaciones de procesamiento de transacción

de datos. Esta clase de sistema incluye los sistemas bancarios interactivos, sistemas de comercio electrónico, sistemas de información y sistemas de reservaciones.

2. *Sistemas de procesamiento de lenguaje* Son sistemas en los que las intenciones del usuario se expresan en un lenguaje formal (como Java). El sistema de procesamiento de lenguaje elabora este lenguaje en un formato interno y después interpreta dicha representación interna. Los sistemas de procesamiento de lenguaje mejor conocidos son los compiladores, que traducen los programas en lenguaje de alto nivel dentro de un código de máquina. Sin embargo, los sistemas de procesamiento de lenguaje se usan también en la interpretación de lenguajes de comandos para bases de datos y sistemas de información, así como de lenguajes de marcado como XML (Harold y Means, 2002; Hunter *et al*., 2007).

Se eligieron estos tipos particulares de sistemas porque una gran cantidad de sistemas empresariales basados en la Web son sistemas de procesamiento de transacciones, y todo el desarrollo del software se apoya en los sistemas de procesamiento de lenguaje.

6.4.1 Sistemas de procesamiento de transacciones

Los sistemas de procesamiento de transacciones (TP, por las siglas de *Transaction Processing*) están diseñados para procesar peticiones del usuario mediante la información de una base de datos, o los requerimientos para actualizar una base de datos (Lewis *et al*., 2003). Técnicamente, una transacción de base de datos es una secuencia de operaciones que se trata como una sola unidad (una unidad atómica). Todas las operaciones en una transacción tienen que completarse antes de que sean permanentes los cambios en la base de datos. Esto garantiza que la falla en las operaciones dentro de la transacción no conduzca a inconsistencias en la base de datos.

Desde una perspectiva de usuario, una transacción es cualquier secuencia coherente de operaciones que satisfacen un objetivo, como "encontrar los horarios de vuelos de Londres a París". Si la transacción del usuario no requiere el cambio en la base de datos, entonces sería innecesario empaquetar esto como una transacción técnica de base de datos.

Un ejemplo de una transacción es una petición de cliente para retirar dinero de una cuenta bancaria mediante un cajero automático. Esto incluye obtener detalles de la cuenta del cliente, verificar y modificar el saldo por la cantidad retirada y enviar comandos al cajero automático para entregar el dinero. Hasta que todos estos pasos se completan, la transacción permanece inconclusa y no cambia la base de datos de la cuenta del cliente.

Por lo general, los sistemas de procesamiento de transacción son sistemas interactivos donde los usuarios hacen peticiones asíncronas de servicios. La figura 6.14 ilustra la estructura arquitectónica conceptual de las aplicaciones de TP. Primero, un usuario hace una petición al sistema a través de un componente de procesamiento I/O. La petición se procesa mediante alguna lógica específica de la aplicación. Se crea una transacción y pasa hacia un gestor de transacciones que, por lo general, está embebido en el sistema de manejo de la

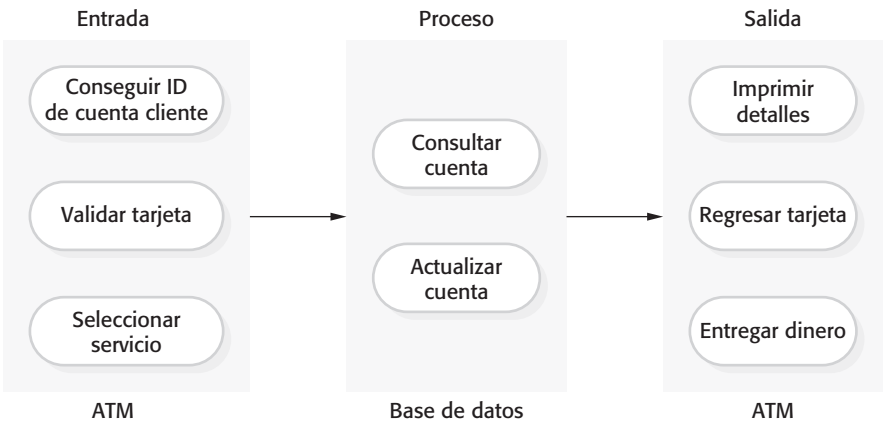


Figura 6.15 Arquitectura de software de un sistema ATM

base de datos. Después de que el gestor de transacciones asegura que la transacción se ha completado adecuadamente, señala a la aplicación que terminó el procesamiento.

Los sistemas de procesamiento de transacción pueden organizarse como una arquitectura "tubería y filtro" con componentes de sistema responsables de entradas, procesamiento y salida. Por ejemplo, considere un sistema bancario que permite a los clientes consultar sus cuentas y retirar dinero de un cajero automático. El sistema está constituido por dos componentes de software cooperadores: el software del cajero automático y el software de procesamiento de cuentas en el servidor de la base de datos del banco. Los componentes de entrada y salida se implementan como software en el cajero automático y el componente de procesamiento es parte del servidor de la base de datos del banco. La figura 6.15 muestra la arquitectura de este sistema e ilustra las funciones de los componentes de entrada, proceso y salida.

6.4.2 Sistemas de información

Todos los sistemas que incluyen interacción con una base de datos compartida se consideran sistemas de información basados en transacciones. Un sistema de información permite acceso controlado a una gran base de información, tales como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. Cada vez más, los sistemas de información son sistemas basados en la Web, cuyo acceso es mediante un navegador Web.

La figura 6.16 presenta un modelo muy general de un sistema de información. El sistema se modela con un enfoque por capas (estudiado en la sección 6.3), donde la capa superior soporta la interfaz de usuario, y la capa inferior es la base de datos del sistema. La capa de comunicaciones con el usuario maneja todas las entradas y salidas de la interfaz de usuario, y la capa de recuperación de información incluye la lógica específica de aplicación para acceder y actualizar la base de datos. Como se verá más adelante, las capas en este modelo pueden trazarse directamente hacia servidores dentro de un sistema basado en Internet.

Como ejemplo de una instancia de este modelo en capas, la figura 6.17 muestra la arquitectura del MHC-PMS. Recuerde que este sistema mantiene y administra detalles de los pacientes que consultan médicos especialistas en problemas de salud mental. En el

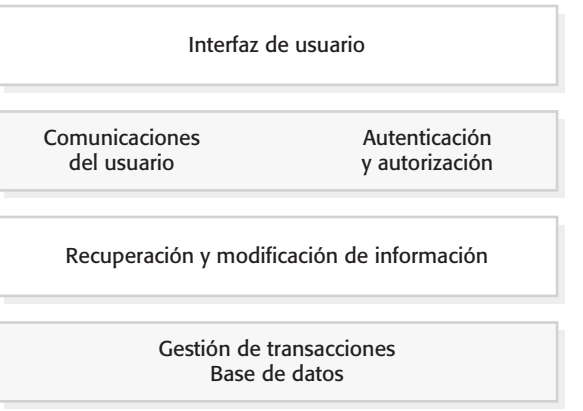


Figura 6.16 Arquitectura de sistema de información en capas

modelo se agregaron detalles a cada capa al identificar los componentes que soportan las comunicaciones del usuario, así como la recuperación y el acceso a la información:

1. La capa superior es responsable de implementar la interfaz de usuario. En este caso, la UI se implementó con el uso de un navegador Web.
2. La segunda capa proporciona la funcionalidad de interfaz de usuario que se entrega a través del navegador Web. Incluye componentes que permiten a los usuarios ingresar al sistema, y componentes de verificación para garantizar que las operaciones utilizadas estén permitidas de acuerdo con su rol. Esta capa incluye componentes de gestión de formato y menú que presentan información a los usuarios, así como componentes de validación de datos que comprueban la consistencia de la información.
3. La tercera capa implementa la funcionalidad del sistema y ofrece componentes que ponen en operación la seguridad del sistema, la creación y actualización de la información del paciente, la importación y exportación de datos del paciente desde otras bases de datos, y los generadores de reporte que elaboran informes administrativos.

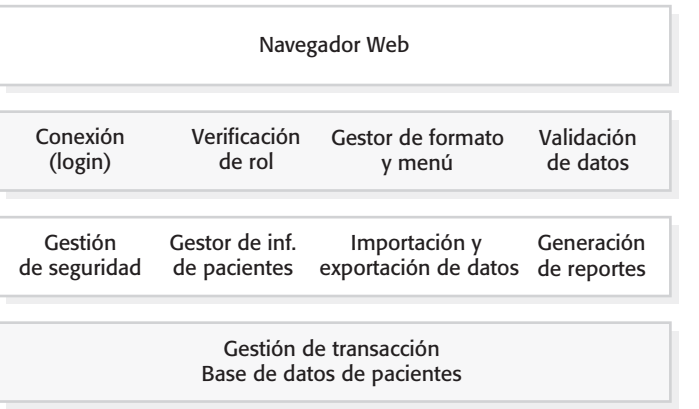


Figura 6.17 Arquitectura del MHC-PMS

4. Finalmente, la capa más baja, que se construye al usar un sistema comercial de gestión de base de datos, ofrece administración de transacciones y almacenamiento constante de datos.

Los sistemas de gestión de información y recursos, por lo general, son ahora sistemas basados en la Web donde las interfaces de usuario se implementan con el uso de un navegador Web. Por ejemplo, los sistemas de comercio electrónico son sistemas de gestión de recursos basados en Internet, que aceptan pedidos electrónicos por bienes o servicios y, luego, ordenan la entrega de dichos bienes o servicios al cliente. En un sistema de comercio electrónico, la capa específica de aplicación incluye funcionalidad adicional que soporta un "carrito de compras", donde los usuarios pueden colocar algunos objetos en transacciones separadas y, luego, pagarlos en una sola transacción.

La organización de servidores en dichos sistemas refleja usualmente el modelo genérico en cuatro capas presentadas en la figura 6.16. Dichos sistemas se suelen implementar como arquitecturas cliente-servidor de multinivel, como se estudia en el capítulo 18:

1. El servidor Web es responsable de todas las comunicaciones del usuario, y la interfaz de usuario se pone en función mediante un navegador Web;
2. El servidor de aplicación es responsable de implementar la lógica específica de la aplicación, así como del almacenamiento de la información y las peticiones de recuperación;
3. El servidor de la base de datos mueve la información hacia y desde la base de datos y, además, manipula la gestión de transacciones.

El uso de múltiples servidores permite un rendimiento elevado, al igual que posibilita la manipulación de cientos de transacciones por minuto. Conforme aumenta la demanda, pueden agregarse servidores en cada nivel, para lidiar con el procesamiento adicional implicado.

6.4.3 Sistemas de procesamiento de lenguaje

Los sistemas de procesamiento de lenguaje convierten un lenguaje natural o artificial en otra representación del lenguaje y, para lenguajes de programación, también pueden ejecutar el código resultante. En ingeniería de software, los compiladores traducen un lenguaje de programación artificial en código de máquina. Otros sistemas de procesamiento de lenguaje traducen una descripción de datos XML en comandos para consultar una base de datos o una representación XML alternativa. Los sistemas de procesamiento de lenguaje natural pueden transformar un lenguaje natural a otro, por ejemplo, francés a noruego.

En la figura 6.18 se ilustra una posible arquitectura para un sistema de procesamiento de lenguaje hacia un lenguaje de programación. Las instrucciones en el lenguaje fuente definen el programa a ejecutar, en tanto que un traductor las convierte en instrucciones para una máquina abstracta. Luego, dichas instrucciones se interpretan mediante otro componente que lee (*fetch*) las instrucciones para su ejecución y las ejecuta al usar (si es necesario) datos del entorno. La salida del proceso es el resultado de la interpretación de instrucciones en los datos de entrada.

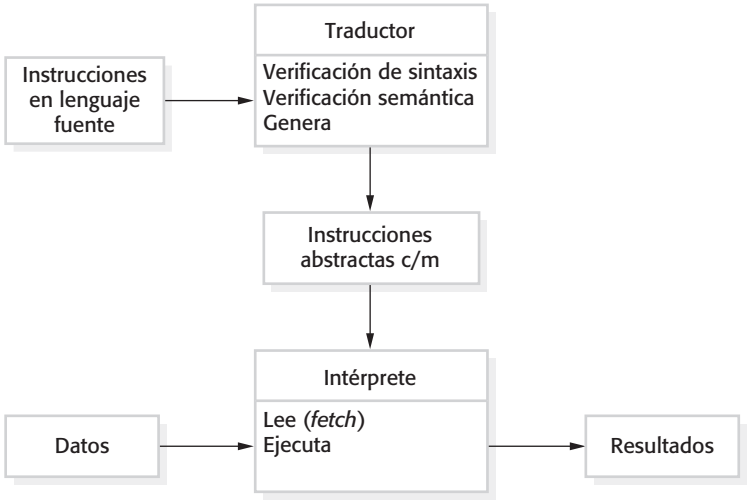


Figura 6.18 Arquitectura de un sistema de procesamiento de lenguaje

Desde luego, para muchos compiladores, el intérprete es una unidad de hardware que procesa instrucciones de la máquina, en tanto que la máquina abstracta es el procesador real. Sin embargo, para lenguajes escritos de manera dinámica, como Python, el intérprete puede ser un componente de software.

Los compiladores de lenguaje de programación que forman parte de un entorno de programación más general tienen una arquitectura genérica (figura 6.19) que incluye los siguientes componentes:

1. Un analizador léxico, que toma valores simbólicos (*tokens*) y los convierte en una forma interna.
2. Una tabla de símbolos, que contiene información de los nombres de las entidades (variables, de clase, de objeto, etcétera) usados en el texto que se traduce.
3. Un analizador de sintaxis, el cual verifica la sintaxis del lenguaje que se va a traducir. Emplea una gramática definida del lenguaje y construye un árbol de sintaxis.
4. Un árbol de sintaxis es una estructura interna que representa el programa a compilar.

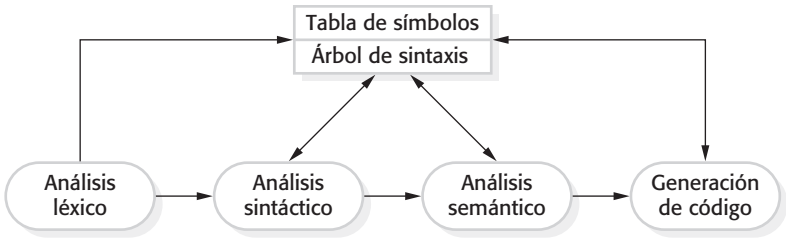


Figura 6.19 Una arquitectura de compilador de tubería y filtro

Arquitecturas de referencia

Las arquitecturas de referencia captan en un dominio características importantes de las arquitecturas del sistema. En esencia, incluyen todo lo que pueda estar en una arquitectura de aplicación aunque, en realidad, es muy improbable que alguna aplicación individual contenga todas las características mostradas en una arquitectura de referencia. El objetivo principal de las arquitecturas de referencia consiste en evaluar y comparar las propuestas de diseño y, en dicho dominio, educar a las personas sobre las características arquitectónicas.

http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html

5. Un analizador semántico que usa información del árbol de sintaxis y la tabla de símbolos, para verificar la exactitud semántica del texto en lenguaje de entrada.
6. Un generador de código que "recorre" el árbol de sintaxis y genera un código de máquina abstracto.

También pueden incluirse otros componentes que analizan y transforman el árbol de sintaxis para mejorar la eficiencia y remover redundancia del código de máquina generado. En otros tipos de sistema de procesamiento de lenguaje, como un traductor de lenguaje natural, habrá componentes adicionales, por ejemplo, un diccionario, y el código generado en realidad es el texto de entrada traducido en otro lenguaje.

Existen patrones arquitectónicos alternativos que pueden usarse en un sistema de procesamiento de lenguaje (Garlan y Shaw, 1993). Pueden implementarse compiladores con una composición de un repositorio y un modelo de tubería y filtro. En una arquitectura de compilador, la tabla de símbolos es un repositorio para datos compartidos. Las fases de análisis léxico, sintáctico y semántico se organizan de manera secuencial, como se muestra en la figura 6.19, y se comunican a través de la tabla de símbolos compartida.

Este modelo de tubería y filtro de compilación de lenguaje es efectivo en entornos *batch*, donde los programas se compilan y ejecutan sin interacción del usuario; por ejemplo, en la traducción de un documento XML a otro. Es menos efectivo cuando un compilador se integra con otras herramientas de procesamiento de lenguaje, como un sistema de edición estructurado, un depurador interactivo o un programa de impresión estética (*prettyprinter*). En esta situación, los cambios de un componente deben reflejarse de inmediato en otros componentes. Por lo tanto, es mejor organizar el sistema en torno a un repositorio, como se muestra en la figura 6.20.

Esta figura ilustra cómo un sistema de procesamiento de lenguaje puede formar parte de un conjunto integrado de herramientas de soporte de programación. En este ejemplo, la tabla de símbolos y el árbol de sintaxis actúan como un almacén de información central. Las herramientas y los fragmentos de herramienta se comunican a través de él. Otra información que en ocasiones se incrusta en las herramientas, como la definición gramática y la definición del formato de salida para el programa, se toma de las herramientas y se coloca en el repositorio. En consecuencia, un editor enfocado en la sintaxis podría verificar que ésta sea correcta mientras se escribe un programa, y un *prettyprinter* puede crear listados del programa en un formato que sea fácil de leer.

Figura 6.20 Arquitectura de repositorio para un sistema de procesamiento de lenguaje

PUNTOS CLAVE

- Una arquitectura de software es una descripción de cómo se organiza un sistema de software. Las propiedades de un sistema, como rendimiento, seguridad y disponibilidad, están influidas por la arquitectura utilizada.
- Las decisiones de diseño arquitectónico incluyen decisiones sobre el tipo de aplicación, la distribución del sistema, los estilos arquitectónicos a usar y las formas en que la arquitectura debe documentarse y evaluarse.
- Las arquitecturas pueden documentarse desde varias perspectivas o diferentes vistas. Las posibles vistas incluyen la conceptual, la lógica, la de proceso, la de desarrollo y la física.
- Los patrones arquitectónicos son medios para reutilizar el conocimiento sobre las arquitecturas de sistemas genéricos. Describen la arquitectura, explican cuándo debe usarse, y exponen sus ventajas y desventajas.
- Los patrones arquitectónicos usados comúnmente incluyen el modelo de vista del controlador, arquitectura en capas, repositorio, cliente-servidor, y tubería y filtro.
- Los modelos genéricos de las arquitecturas de sistemas de aplicación ayudan a entender la operación de las aplicaciones, comparar aplicaciones del mismo tipo, validar diseños del sistema de aplicación y valorar componentes para reutilización a gran escala.
- Los sistemas de procesamiento de transacción son sistemas interactivos que permiten el acceso y la modificación remota de la información, en una base de datos por parte de varios usuarios. Los sistemas de información y los sistemas de gestión de recursos son ejemplos de sistemas de procesamiento de transacciones.
- Los sistemas de procesamiento de lenguaje se usan para traducir textos de un lenguaje a otro y para realizar las instrucciones especificadas en el lenguaje de entrada. Incluyen un traductor y una máquina abstracta que ejecuta el lenguaje generado.

LECTURAS SUGERIDAS

Software Architecture: Perspectives on an Emerging Discipline. Éste fue el primer libro sobre arquitectura de software e incluye una amplia discusión acerca de los diferentes estilos arquitectónicos. (M. Shaw y D. Garlan, Prentice-Hall, 1996.)

Software Architecture in Practice, 2nd ed. Se trata de una cuestión práctica de arquitecturas de software que no exagera los beneficios del diseño arquitectónico. Ofrece una clara razón empresarial sobre por qué son importantes las arquitecturas. (L. Bass, P. Clements y R. Kazman, Addison-Wesley, 2003.)

"The Golden Age of Software Architecture". Este ensayo estudia el desarrollo de la arquitectura de software, desde sus inicios en la década de 1980 hasta su uso actual. Aun cuando tiene poco contenido técnico, presenta un panorama histórico amplio e interesante. (M. Shaw y P. Clements, *IEEE Software,* **21** (2), marzo-abril 2006.) http://dx.doi.org/10.1109/MS.2006.58.

Handbook of Software Architecture. Éste es un trabajo en progreso de Grady Booch, uno de los primeros difusores de la arquitectura de software. Ha documentado las arquitecturas de varios sistemas de software, de manera que el lector puede ver la realidad en vez de abstracción académica. Disponible en la Web y con la intención de aparecer como libro. http://www.handbookofsoftwarearchitecture.com/.

EJERCICIOS

6.1. Cuando se describe un sistema, explique por qué es posible que deba diseñar la arquitectura del sistema antes de completar la especificación de requerimientos.

6.2. Se le pide preparar y entregar una presentación a un administrador no técnico para justificar la contratación de un arquitecto de sistemas para un nuevo proyecto. Escriba una lista que establezca los puntos clave de su presentación. Por supuesto, debe explicar qué se entiende por arquitecto de sistemas.

6.3. Exponga por qué pueden surgir conflictos de diseño cuando se desarrolla una arquitectura para la que tanto los requerimientos de disponibilidad como los de seguridad son los requerimientos no funcionales más importantes.

6.4. Dibuje diagramas que muestren una vista conceptual y una vista de proceso de las arquitecturas de los siguientes sistemas:

Un sistema automatizado de emisión de boletos que utilizan los pasajeros en una estación de ferrocarril.

Un sistema de videoconferencia controlado por computadora, que permita que los datos de video, audio y computadora sean al mismo tiempo visibles a muchos participantes.

Un robot limpiador de pisos cuya función sea asear espacios relativamente despejados, como corredores. El limpiador debe detectar las paredes y otros obstáculos.

6.5. Explique por qué usted usa normalmente muchos patrones arquitectónicos cuando diseña la arquitectura de un sistema grande. Además de la información sobre los patrones estudiados en este capítulo, ¿qué información adicional puede serle útil al diseñar sistemas grandes?

6.6. Sugiera una arquitectura para un sistema (como iTunes) que se use para vender y distribuir música por Internet. ¿Qué patrones arquitectónicos son la base para esta arquitectura?

6.7. Especifique cómo usaría el modelo de referencia de entornos CASE (disponibles en las páginas Web del libro), para comparar los IDE ofrecidos por diferentes proveedores de un lenguaje de programación como Java.

6.8. Con el modelo genérico de un sistema de procesamiento de lenguaje presentado aquí, diseñe la arquitectura de un sistema que acepte comandos en lenguaje natural y los traduzca en consultas de base de datos en un lenguaje como SQL.

6.9. Con el modelo básico de un sistema de información, como se presentó en la figura 6.16, sugiera los componentes que puedan ser parte de un sistema de información que permita a los usuarios consultar información de los vuelos que llegan y salen de un aeropuerto específico.

6.10. ¿Debe existir una profesión separada de "arquitecto de software", cuyo papel sea trabajar de manera independiente con un cliente para diseñar la arquitectura de un sistema de software? Entonces, una compañía de software aparte implementaría el sistema. ¿Cuáles serían las dificultades de establecer tal profesión?

REFERENCIAS

Bass, L., Clements, P. y Kazman, R. (2003). Software Architecture in Practice, 2a ed. Boston: Addison-Wesley.

Berczuk, S. P. y Appleton, B. (2002). Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston: Addison-Wesley.

Booch, G. (2009). "Handbook of software architecture". Publicación Web. http://www.handbookofsoftwarearchitecture.com/.

Bosch, J. (2000). Design and Use of Software Architectures. Harlow, UK: Addison-Wesley.

Buschmann, F., Henney, K. y Schmidt, D. C. (2007a). Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. New York: John Wiley & Sons.

Buschmann, F., Henney, K. y Schmidt, D. C. (2007b). Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages. New York: John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H. y Sommerlad, P. (1996). Pattern-oriented Software Architecture Volume 1: A System of Patterns. New York: John Wiley & Sons.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. y Stafford, J. (2002). Documenting Software Architectures: Views and Beyond. Boston: Addison-Wesley.

Coplien, J. H. y Harrison, N. B. (2004). Organizational Patterns of Agile Software Development. Englewood Cliffs, NJ: Prentice Hall.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley.

Garlan, D. y Shaw, M. (1993). "An introduction to software architecture". Advances in Software Engineering and Knowledge Engineering, 1 1–39.

Harold, E. R. y Means, W. S. (2002). XML in a Nutshell. Sebastopol. Calif.: O'Reilly.

Hofmeister, C., Nord, R. y Soni, D. (2000). Applied Software Architecture. Boston: Addison-Wesley.

Hunter, D., Rafter, J., Fawcett, J. y Van Der Vlist, E. (2007). Beginning XML, 4a ed. Indianapolis, Ind.: Wrox Press.

Kircher, M. y Jain, P. (2004). Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. New York: John Wiley & Sons.

Krutchen, P. (1995). "The 4+1 view model of software architecture". IEEE Software, **12** (6), 42–50.

Lange, C. F. J., Chaudron, M. R. V. y Muskens, J. (2006). "UML software description and architecture description". IEEE Software, **23** (2), 40–6.

Lewis, P. M., Bernstein, A. J. y Kifer, M. (2003). Databases and Transaction Processing: An Application-oriented Approach. Boston: Addison-Wesley.

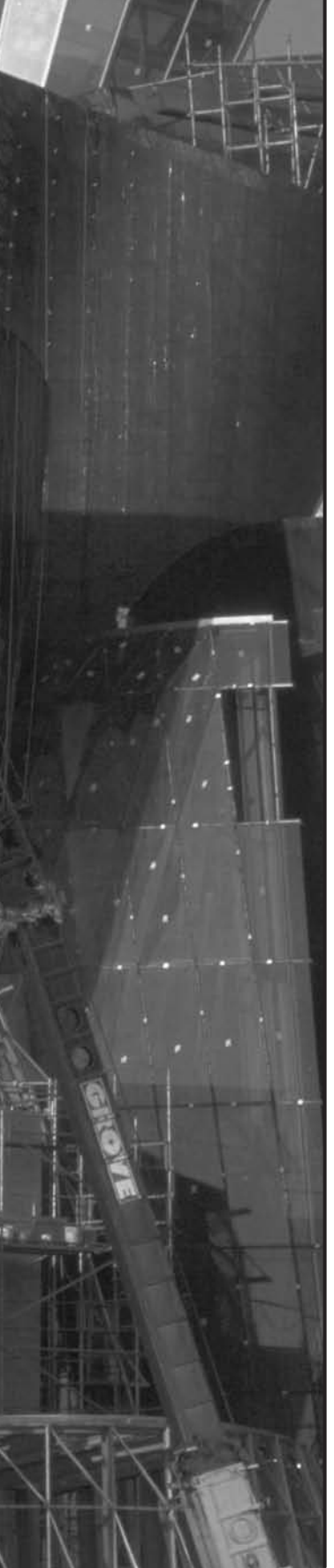
Martin, D. y Sommerville, I. (2004). "Patterns of interaction: Linking ethnomethodology and design". ACM Trans. on Computer-Human Interaction, **11** (1), 59–89.

Nii, H. P. (1986). "Blackboard systems, parts 1 and 2". AI Magazine, **7** (3 y 4), 38–53 y 62–9.

Schmidt, D., Stal, M., Rohnert, H. y Buschmann, F. (2000). Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. New York: John Wiley & Sons.

Shaw, M. y Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, NJ: Prentice Hall.

Usability group. (1998). "Usability patterns". Publicación Web. http://www.it.bton.ac.uk/cil/usability/patterns/.



7

Diseño e implementación

Objetivos

Los objetivos de este capítulo son introducirlo al diseño de software orientado a objetos con el uso del UML, así como resaltar las preocupaciones relevantes en la implementación. Al estudiar este capítulo:

- comprenderá las actividades más importantes dentro de un proceso de diseño general orientado a objetos;
- identificará algunos de los diferentes modelos que pueden usarse para documentar un diseño orientado a objetos;
- conocerá la idea de patrones de diseño y cómo éstos son una forma de reutilizar el conocimiento y la experiencia de diseño;
- se introducirá en los conflictos clave que debe considerar al implementar el software, incluida la reutilización de software y el desarrollo de código abierto.

Contenido

7.1 Diseño orientado a objetos con el uso del UML
7.2 Patrones de diseño
7.3 Conflictos de implementación
7.4 Desarrollo de código abierto

El diseño y la implementación del software es la etapa del proceso de ingeniería de software en que se desarrolla un sistema de software ejecutable. Para algunos sistemas simples, el diseño y la implementación del software es ingeniería de software, y todas las demás actividades se fusionan con este proceso. Sin embargo, para sistemas grandes, el diseño y la implementación del software son sólo uno de una serie de procesos (ingeniería de requerimientos, verificación y validación, etcétera) implicados en la ingeniería de software.

Las actividades de diseño e implementación de software se encuentran invariablemente entrelazadas. El diseño de software es una actividad creativa donde se identifican los componentes del software y sus relaciones, con base en los requerimientos de un cliente. La implementación es el proceso de realizar el diseño como un programa. Algunas veces, hay una etapa de diseño separada y este último se modela y documenta. En otras ocasiones, un diseño se halla en la mente del programador o se bosqueja burdamente en un pizarrón o en hojas de papel. El diseño trata sobre cómo resolver un problema, de modo que siempre existe un proceso de diseño. Sin embargo, en ocasiones no es necesario o adecuado describir con detalle el diseño usando el UML u otro lenguaje de descripción de diseño.

Diseño e implementación están estrechamente vinculados y usted, por lo general, debe tomar en cuenta los conflictos de implementación cuando desarrolle un diseño. Por ejemplo, usar el UML para documentar un diseño puede ser lo correcto si está programando en un lenguaje orientado a objetos, como Java o C#. Es menos útil cuando se desarrolla en un lenguaje de escritura dinámica, como Python, y no tiene sentido en absoluto si implementa su sistema al configurar un paquete comercial. Como se estudió en el capítulo 3, los métodos ágiles suelen funcionar a partir de bosquejos informales del diseño y dejan a los programadores muchas de las decisiones de diseño.

Una de las decisiones de implementación más importantes, que se toman en una etapa inicial de un proyecto de software, consiste en determinar si debe comprar o diseñar el software de aplicación. En un gran rango de dominios, ahora es posible comprar sistemas comerciales (COTS) que se adapten y personalicen según los requerimientos de los usuarios. Por ejemplo, si desea implementar un sistema de registros médicos, puede comprar un paquete que ya se use en hospitales. Es posible que sea más barato y rápido aplicar este enfoque en vez de desarrollar un sistema en un lenguaje de programación convencional.

Al desarrollarse de esta forma una aplicación, el proceso de diseño se preocupa sobre cómo usar las características de configuración de dicho sistema, para entregar los requerimientos del mismo. Por lo general, no se desarrollan modelos de diseño del sistema, como los modelos de los objetos del sistema y sus interacciones. En el capítulo 16 se estudia el enfoque basado en COTS.

Se supone que la mayoría de los lectores de este libro cuentan con algo de experiencia en diseño e implementación de programas. Esto se adquiere conforme se aprende a programar y se dominan los elementos de un lenguaje de programación como Java o Python. Probablemente usted se instruyó en las buenas prácticas de programación cuando estudió lenguajes de programación, así como al depurar los programas que desarrolla. Por lo tanto, aquí no se cubren temas de programación. En cambio, este capítulo tiene dos metas:

1. Mostrar cómo el modelado de sistemas y el diseño arquitectónico (que se estudian en los capítulos 5 y 6) se ponen en práctica en el desarrollo de un diseño de software orientado a objetos.

Métodos de diseño estructurado

Los métodos de diseño estructurado refieren que el diseño del software debe realizarse en una forma metódica. Diseñar un sistema incluye seguir los pasos del método, así como corregir el diseño de un sistema en niveles cada vez más detallados. En la década de 1990 había algunos métodos en competencia para diseño orientado a objetos. Sin embargo, los creadores de los métodos de uso más común se reunieron e inventaron el UML, que unificó las anotaciones usadas con los diferentes métodos.

En lugar de enfocarse en los métodos, la mayoría de las discusiones son ahora sobre procesos donde el diseño se ve como parte del proceso global de desarrollo del software. El proceso racional unificado (RUP, por las siglas de *Rational Unified Process*) es una buena muestra de proceso de desarrollo genérico.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

2. Introducir importantes temas de implementación que no se tratan normalmente en los libros de programación. En ellos se incluyen la reutilización de software, la administración de la configuración y el desarrollo de código abierto.

Como hay gran variedad de plataformas de desarrollo, el capítulo no se desvía hacia algún lenguaje de programación o tecnología de implementación específicos. Por lo tanto, todos los ejemplos se presentan usando el UML en vez de un lenguaje de programación como Java o Python.

7.1 Diseño orientado a objetos con el uso del UML

Un sistema orientado a objetos se constituye con objetos que interactúan y mantienen su propio estado local y ofrecen operaciones sobre dicho estado. La representación del estado es privada y no se puede acceder directamente desde afuera del objeto. Los procesos de diseño orientado a objetos implican el diseño de clases de objetos y las relaciones entre dichas clases; tales clases definen tanto los objetos en el sistema como sus interacciones. Cuando el diseño se realiza como un programa en ejecución, los objetos se crean dinámicamente a partir de estas definiciones de clase.

Los sistemas orientados a objetos son más fáciles de cambiar que aquellos sistemas desarrollados usando enfoques funcionales. Los objetos incluyen datos y operaciones para manipular dichos datos. En consecuencia, pueden entenderse y modificarse como entidades independientes. Cambiar la implementación de un objeto o agregar servicios no afectará a otros objetos del sistema. Puesto que los objetos se asocian con cosas, con frecuencia hay un mapeo claro entre entidades del mundo real (como componentes de hardware) y sus objetos controladores en el sistema. Esto mejora la comprensibilidad y, por ende, la mantenibilidad del diseño.

Para desarrollar un diseño de sistema desde el concepto hasta el diseño detallado orientado a objetos, hay muchas cuestiones por hacer:

1. Comprender y definir el contexto y las interacciones externas con el sistema.
2. Diseñar la arquitectura del sistema.

3. Identificar los objetos principales en el sistema.
4. Desarrollar modelos de diseño.
5. Especificar interfaces.

Como todas las actividades creativas, el diseño no es un proceso secuencial tajante. El diseño se desarrolla al obtener ideas, proponer soluciones y corregir dichas soluciones conforme la información se encuentra disponible. Cuando surjan problemas, se tendrá inevitablemente que regresar y volver a intentar. En ocasiones se exploran opciones a detalle para observar si funcionan; en otras, se ignoran hasta los detalles finales del proceso. En consecuencia, en el texto no se ilustra deliberadamente este proceso como un diagrama simple, debido a que ello implicaría que el diseño se pudiera considerar como una secuencia clara de actividades. De hecho, todas las actividades anteriores están entrelazadas y, por consiguiente, influyen entre sí.

Estas actividades de proceso se explican al diseñar parte del software para la estación meteorológica a campo abierto que se presentó en el capítulo 1. Las estaciones meteorológicas a campo abierto se despliegan a áreas remotas. Cada estación meteorológica registra información meteorológica local y la transmite periódicamente, a través de un vínculo satelital, a un sistema de información meteorológica.

7.1.1 Contexto e interacciones del sistema

La primera etapa en cualquier proceso de diseño de software es desarrollar la comprensión de las relaciones entre el software que se diseñará y su ambiente externo. Esto es esencial para decidir cómo proporcionar la funcionalidad requerida del sistema y cómo estructurar el sistema para que se comunique con su entorno. La comprensión del contexto permite también determinar las fronteras del sistema.

El establecimiento de las fronteras del sistema ayuda a decidir sobre las características que se implementarán en el sistema que se va a diseñar, así como sobre las de otros sistemas asociados. En este caso, es necesario decidir cómo se distribuirá la funcionalidad entre el sistema de control para todas las estaciones meteorológicas y el software embebido en la estación meteorológica en sí.

Los modelos de contexto del sistema y los modelos de interacción presentan vistas complementarias de las relaciones entre un sistema y su entorno:

1. Un modelo de contexto del sistema es un modelo estructural, que muestra los otros sistemas en el entorno del sistema a desarrollar.
2. Un modelo de interacción es un modelo dinámico que indica la forma en que el sistema interactúa con su entorno conforme se utiliza.

El modelo del contexto de un sistema puede representarse mediante asociaciones, las cuales muestran simplemente que existen algunas relaciones entre las entidades que intervienen en la asociación. La naturaleza de la relaciones es ahora específica. Por lo tanto, es posible documentar el entorno del sistema con un simple diagrama de bloques, que manifieste las entidades en el sistema y sus asociaciones. Esto se expone en la figura 7.1, la cual indica que los sistemas en el entorno de cada estación meteorológica son un sistema

Casos de uso de la estación meteorológica

Reporte del clima: envía datos meteorológicos al sistema de información meteorológica
Reporte de estatus: manda información de estatus al sistema de información meteorológica
Reinicio: si la estación meteorológica se apaga, reinicia el sistema
Apagar: desconecta la estación meteorológica
Reconfigurar: vuelve a configurar el software de la estación meteorológica
Ahorro de energía: pone la estación meteorológica en modo de ahorro de energía
Control remoto: envía comandos de control a cualquier subsistema de la estación meteorológica

http://www.SoftwareEngineering-9.com/Web/WS/Usecases.html

de información meteorológica, un sistema de satélite a bordo y un sistema de control. La información cardinal en el vínculo muestra que hay un sistema de control, pero existen muchas estaciones meteorológicas, un satélite y un sistema de información meteorológica general.

Al modelar las interacciones de un sistema con su entorno, se debe usar un enfoque abstracto que no contenga muchos detalles. Una forma de hacerlo es usar un modelo de caso de uso. Como se estudió en los capítulos 4 y 5, cada caso de uso representa una interacción con el sistema. Cada posible interacción se menciona en una elipse, y la entidad externa involucrada en la interacción se representa con una figurilla.

En la figura 7.2 se presenta el modelo de caso de uso para la estación meteorológica. Ahí se demuestra que la estación meteorológica interactúa con el sistema de información meteorológica para reportar datos meteorológicos y el estatus del hardware de la estación meteorológica. Otras interacciones son a través de un sistema de control que puede emitir comandos de control específicos a la estación meteorológica. Como se explicó en el capítulo 5, en el UML se usa una figura para representar otros sistemas, así como a usuarios.

Cada uno de estos casos de uso tiene que describirse en lenguaje natural estructurado. Eso ayudaría a los diseñadores a identificar objetos en el sistema y les daría claridad acerca de lo que se pretende que haga el sistema. Para esta descripción, se usa un formato estándar que identifica con sencillez qué información se intercambia, cómo

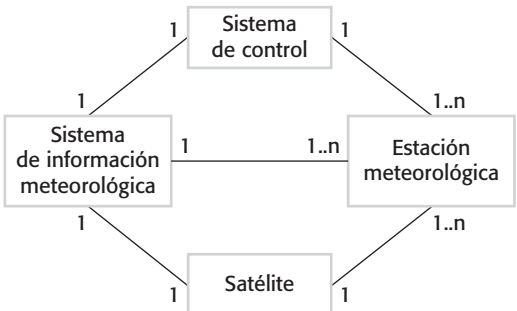


Figura 7.1 Contexto de sistema para la estación meteorológica

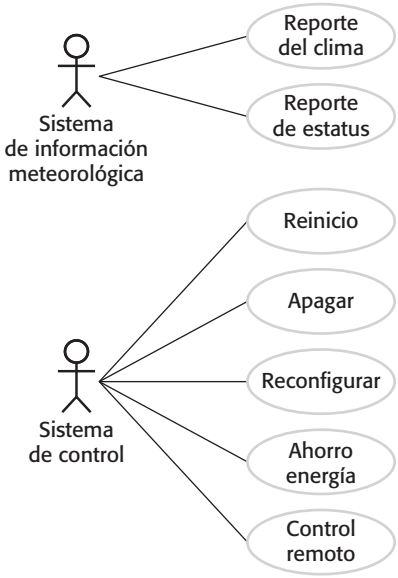


Figura 7.2 Casos de uso para estación meteorológica

se inicia la interacción, etcétera. Eso se muestra en la figura 7.3, que describe el caso de uso sobre el reporte del clima de la figura 7.2. En la Web hay ejemplos de algunos otros casos de uso.

7.1.2 Diseño arquitectónico

Una vez definidas las interacciones entre el sistema de software y el entorno del sistema, se aplica esta información como base para diseñar la arquitectura del sistema. Desde luego, es preciso combinar esto con el conocimiento general de los principios del diseño

Figura 7.3 Descripción de caso de uso: reporte del clima

Sistema	Estación meteorológica
Caso de uso	Reporte del clima
Actores	Sistema de información meteorológica, estación meteorológica
Datos	La estación meteorológica envía un resumen de datos meteorológicos, recopilados de los instrumentos en el periodo de recolección, al sistema de información meteorológica. Los datos enviados incluyen las temperaturas, máxima, mínima y promedio de la tierra y el aire; asimismo, las presiones de aire máxima, mínima y promedio; la rapidez del viento, máxima, mínima y promedio; la totalidad de la lluvia y la dirección del viento que se muestrea a intervalos de cinco minutos.
Estímulos	El sistema de información meteorológica establece un vínculo de comunicación satelital con la estación meteorológica y solicita la transmisión de los datos.
Respuesta	Los datos ya resumidos se envían al sistema de información meteorológica.
Comentarios	Por lo general, se pide a las estaciones meteorológicas reportarse una vez cada hora, pero esta periodicidad puede diferir de una estación a otra y modificarse en el futuro.

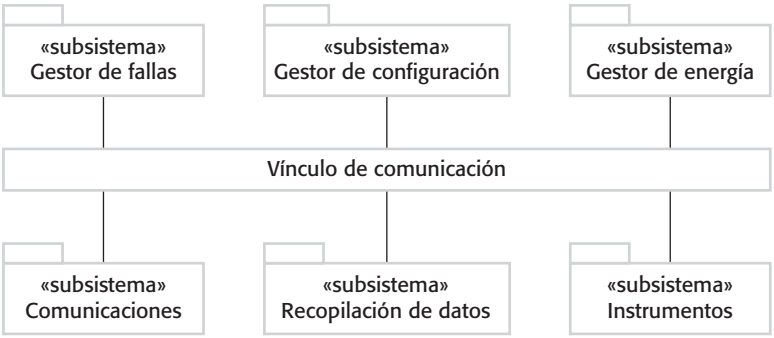


Figura 7.4 Arquitectura de alto nivel de la estación meteorológica

arquitectónico y con la comprensión más detallada del dominio. Identifique los principales componentes que constituyen el sistema y sus interacciones, y posteriormente organice los componentes utilizando un patrón arquitectónico como un modelo en capas o cliente-servidor, aunque esto no sea esencial en esta etapa.

La figura 7.4 muestra el diseño arquitectónico de alto nivel para el software de la estación meteorológica. Esta última se compone de subsistemas independientes que se comunican a través de mensajes de radiodifusión en una infraestructura común, que se presenta como el vínculo "comunicación" en la figura 7.4. Cada subsistema escucha los mensajes en esa infraestructura y capta los mensajes que se dirigen a él. Éste es otro estilo arquitectónico de uso común, además de los descritos en el capítulo 6.

Por ejemplo, cuando el subsistema de comunicaciones recibe un comando de control, como desconectarse, el comando es recibido por cada uno de los otros subsistemas, que entonces se apagan en la forma correcta. El beneficio clave de esta arquitectura consiste en que es fácil soportar diferentes configuraciones de subsistemas debido a que el emisor del mensaje no necesita dirigir el mensaje a un subsistema específico.

La figura 7.5 expone la arquitectura del subsistema de recolección de datos, incluida en la figura 7.4. Los objetos "transmisor" y "receptor" se ocupan de administrar las comunicaciones, y el objeto WeatherData (datos meteorológicos) encapsula la información que se recolecta de los instrumentos y se transmite al sistema de información meteorológica. Este arreglo sigue el patrón productor-consumidor, estudiado en el capítulo 20.

7.1.3 Identificación de clase de objeto

En esta etapa del proceso de diseño, es necesario tener algunas ideas sobre los objetos esenciales en el sistema que se diseña. Conforme aumente su comprensión del diseño, corregirá estas ideas de los objetos del sistema. La descripción del caso de uso ayuda a identificar objetos y operaciones del sistema. A partir de la descripción del caso de uso "reporte del clima", es evidente que se necesitarán objetos que representen los instrumentos que recopilan los datos meteorológicos, así como un objeto que simbolice el resumen de los datos meteorológicos. También se suele requerir de un(unos) objeto(s) de sistema de alto nivel que encapsule(n) las interacciones del sistema definidas en los

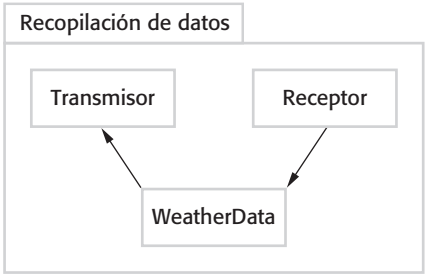


Figura 7.5 Arquitectura de sistema de recopilación de datos

casos de uso. Con dichos objetos en mente, usted puede comenzar a identificar las clases de objetos en el sistema.

Hay varias propuestas sobre cómo identificar las clases de objetos en los sistemas orientados a objetos:

1. Use un análisis gramatical de una descripción en lenguaje natural del sistema a construir. Objetos y atributos son sustantivos; operaciones o servicios son verbos (Abbott, 1983).
2. Utilice entidades tangibles (cosas) en el dominio de aplicación (como aeronave), roles (como administrador o médico), eventos (como peticiones), interacciones (como reuniones), ubicaciones (como oficinas), unidades organizacionales (como compañías), etcétera (Coad y Yourdon, 1990; Shlaer y Mellor, 1988; Wirfs-Brock *et al*., 1990).
3. Emplee un análisis basado en escenarios, donde a la vez se identifiquen y analicen varios escenarios de uso de sistema. A medida que se analiza cada escenario, el equipo responsable del análisis debe identificar los objetos, los atributos y las operaciones requeridos (Beck y Cunningham, 1989).

En la práctica, debe usar varias fuentes de conocimiento para descubrir clases de objetos. Dichas clases, así como los atributos y las operaciones que se identificaron al comienzo a partir de la descripción informal del sistema, pueden ser un punto de inicio para el diseño. Entonces es posible usar más información del conocimiento del dominio de aplicación o del análisis del escenario para corregir y ampliar los objetos iniciales. Esta información se recopila de los documentos de requerimientos, de discusiones con usuarios o de análisis de los sistemas existentes.

En la estación meteorológica a campo abierto, la identificación de objetos se basa en el hardware tangible del sistema. Aun cuando no se tiene suficiente espacio para incluir todos los objetos de sistema, en la figura 7.6 se muestran cinco clases de objetos. Los objetos termómetro de tierra, anemómetro y barómetro pertenecen al dominio de aplicación; en tanto que los objetos WeatherStation (estación meteorológica) y WeatherData (datos meteorológicos) se identificaron a partir de la descripción del sistema y de la descripción del escenario (caso de uso):

1. La clase de objeto WeatherStation proporciona la interfaz básica de la estación meteorológica con su entorno. Sus operaciones reflejan las interacciones que se

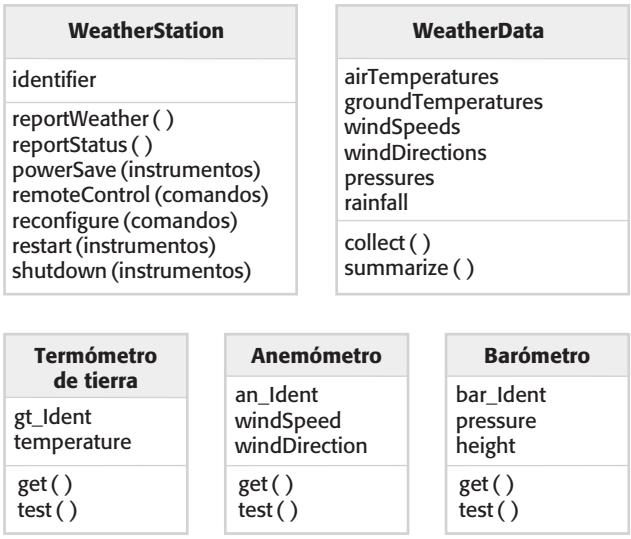


Figura 7.6 Objetos de estación meteorológica

muestran en la figura 7.3. En este caso, aunque se usa una sola clase de objeto para encapsular todas las interacciones, en otros diseños se pueden elaborar varias clases diferentes de la interfaz del sistema.

2. La clase de objeto WeatherData es responsable de procesar el comando del reportWeather (reporte del clima). Envía los datos resumidos desde los aparatos de la estación meteorológica hacia el sistema de información meteorológica.

3. Las clases de objetos "termómetro de tierra", "anemómetro" y "barómetro" se relacionan directamente con instrumentos en el sistema. Reflejan las entidades de hardware tangibles en el sistema, y las operaciones se relacionan con el control de dicho hardware. Tales objetos operan de manera autónoma para recopilar datos en la frecuencia especificada y almacenar localmente los datos recopilados. Estos datos se entregan por encargo al objeto WeatherData.

El conocimiento del dominio de aplicación se usa para identificar otros objetos, atributos y servicios. Se sabe que las estaciones meteorológicas se localizan generalmente en lugares remotos e incluyen varios instrumentos que algunas veces funcionan mal. Las fallas en los instrumentos deben reportarse automáticamente. Esto implica que se necesitan atributos y operaciones para comprobar el buen funcionamiento de los instrumentos. Existen muchas estaciones meteorológicas remotas, de modo que cada estación meteorológica debe tener su propio identificador (*identifier*).

En esta etapa del proceso de diseño hay que enfocarse en los objetos en sí, sin pensar en cómo podrían implementarse. Una vez identificados los objetos, corrija el diseño del objeto. Busque características comunes y luego elabore la jerarquía de herencia para el sistema. Por ejemplo, puede identificar una superclase "instrumento" que defina las características comunes de todos los instrumentos, como un identificador y las operaciones *get* (conseguir) y *test* (probar). También es posible agregar nuevos atributos y operaciones a la superclase, como un atributo que mantenga la frecuencia de recolección de datos.

7.1.4 Modelos de diseño

Como se estudió en el capítulo 5, los modelos de diseño o sistema muestran los objetos o clases de objetos en un sistema. También indican las asociaciones y relaciones entre tales entidades. Dichos modelos son el puente entre los requerimientos y la implementación de un sistema. Deben ser abstractos, de manera que el detalle innecesario no oculte las relaciones entre ellos y los requerimientos del sistema. Sin embargo, deben incluir suficiente detalle para que los programadores tomen decisiones de implementación.

Por lo general, este tipo de conflicto se supera cuando se desarrollan modelos con diferentes niveles de detalle. Donde existan vínculos cercanos entre ingenieros de requerimientos, diseñadores y programadores, los modelos abstractos pueden ser entonces todo lo que se requiera. Es posible tomar decisiones de diseño específico a medida que se implementa el sistema, y los problemas se resuelven mediante discusiones informales. Cuando los vínculos entre especificadores, diseñadores y programadores del sistema son indirectos (por ejemplo, donde un sistema se diseña en una parte de una organización, pero se implementa en otra), es probable que se necesiten modelos más detallados.

Por consiguiente, un paso importante en el proceso de diseño es determinar los modelos de diseño que se necesitarán y el nivel de detalle requerido en dichos modelos. Lo anterior depende del tipo de sistema a desarrollar. Usted diseña un sistema de procesamiento secuencial de datos de forma diferente que un sistema embebido de tiempo real, así que necesitará distintos modelos de diseño. El UML soporta 13 diversos tipos de modelos, pero, como se estudió en el capítulo 5, rara vez los usará todos. Minimizar el número de la producción de modelos reduce los costos del diseño y el tiempo requerido para completar el proceso de diseño.

Al usar el UML para la elaboración de un diseño, usted por lo regular desarrollará dos tipos de modelo de diseño:

1. Modelos estructurales, que describen la estructura estática del sistema usando las clases de objetos y sus relaciones. Las relaciones importantes que pueden documentarse en esta etapa son las relaciones de generalización (herencia), las relaciones usa/usado por y las relaciones de composición.
2. Modelos dinámicos, que explican la estructura dinámica del sistema y muestran las interacciones entre los objetos del sistema. Las interacciones que pueden documentarse incluyen la secuencia de peticiones de servicio realizadas por los objetos, así como los cambios de estado que activan las interacciones de dichos objetos.

En las primeras fases del proceso de diseño, se considera que existen tres modelos que son útiles particularmente para agregar detalle a los modelos de caso de uso y arquitectónico:

1. Modelos de subsistema, que exponen los agrupamientos lógicos de objetos en subsistemas coherentes. Se representan mediante una forma de diagrama de clase en el que cada subsistema se muestra como un paquete con objetos encerrados. Los modelos de subsistema son modelos estáticos (estructurales).

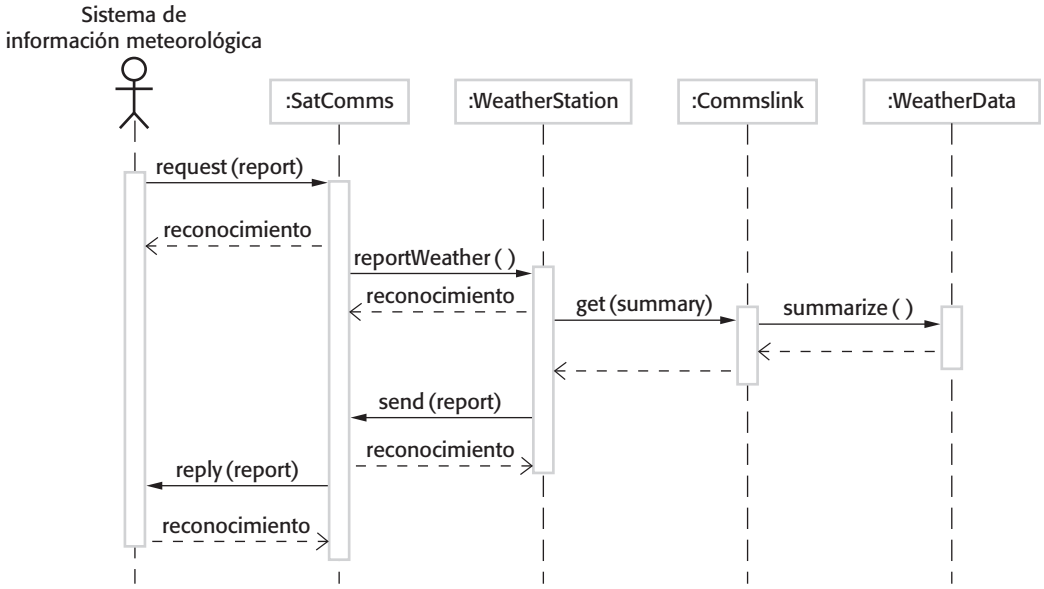


Figura 7.7 Diagrama de secuencia que describe recolección de datos

2. Modelos de secuencia, que ilustran la secuencia de interacciones de objetos. Se representan mediante una secuencia UML o un diagrama de colaboración. Los modelos de secuencia son modelos dinámicos.

3. Modelos de máquina de estado, que muestran cómo los objetos individuales cambian su estado en respuesta a eventos. Se representan en UML a través de diagramas de estado. Los modelos de máquina de estado son modelos dinámicos.

Un modelo de subsistema es un modelo estático útil, pues señala cómo está organizado un diseño en grupos de objetos relacionados lógicamente. En la figura 7.4 ya se expuso este tipo de modelo, para mostrar los subsistemas en el sistema del mapeo meteorológico. Al igual que los modelos de subsistemas, puede diseñar también modelos de objeto detallados, que presenten todos los objetos en los sistemas y sus asociaciones (herencia, generalización, agregación, etcétera). Sin embargo, hay un peligro al hacer demasiado modelado. No debe tomar decisiones detalladas acerca de la implementación que en realidad debería dejarse a los programadores del sistema.

Los modelos de secuencia son modelos dinámicos que describen, para cada modo de interacción, la secuencia de interacciones de objeto que tienen lugar. Cuando se documenta un diseño, debe producirse un modelo de secuencia por cada interacción significativa. Si usted desarrolló un modelo de caso de uso, entonces debe haber un modelo de secuencia para cada caso de uso que identifique.

La figura 7.7 es un ejemplo de modelo de secuencia, que se muestra como un diagrama de secuencia UML. Este diagrama indica la secuencia de interacciones que tienen lugar cuando un sistema externo solicita los datos resumidos de la estación meteorológica. Los diagramas de secuencia se leen de arriba abajo:

1. El objeto SatComms (comunicaciones de satélite) recibe una petición del sistema de información meteorológica para recopilar un reporte del clima de una estación

meteorológica. Reconoce la recepción de esta petición. La flecha continua en el mensaje enviado señala que el sistema externo no espera una respuesta, sino que puede realizar otro procesamiento.

2. SatComms envía un mensaje a WeatherStation, vía un vínculo satelital, para crear un resumen de los datos meteorológicos recolectados. De nuevo, la flecha continua indica que SatComms no se suspende en espera de una respuesta.

3. WeatherStation envía un mensaje a un objeto Commslink (vínculos comunicaciones) para resumir los datos meteorológicos. En este caso, la punta de flecha discontinua revela que la instancia del objeto WeatherStation espera una respuesta.

4. Commslink llama al método summarize (resumir) en el objeto WeatherData y espera una respuesta.

5. El resumen de datos meteorológicos se calcula y regresa a WeatherStation vía el objeto Commslink.

6. WeatherStation entonces llama al objeto SatComms para transmitir los datos resumidos al sistema de información meteorológica, a través del sistema de comunicaciones de satélite.

Los objetos SatComms y WeatherStation se implementan como procesos concurrentes, cuya ejecución puede suspenderse y resumirse. La instancia del objeto SatComms escucha los mensajes del sistema externo, decodifica dichos mensajes e inicia operaciones de estación meteorológica.

Los diagramas de secuencia se usan para modelar el comportamiento combinado de un grupo de objetos, pero quizá también se desee resumir el comportamiento de un objeto o un subsistema, en respuesta a mensajes y eventos. Para hacerlo, se usa un modelo de máquina de estado que muestre cómo la instancia objeto cambia de estado dependiendo de los mensajes que recibe. El UML incluye diagramas de estado, inventados inicialmente por Harel (1987) para describir modelos de máquina de estado.

La figura 7.8 es un diagrama de estado para el sistema de estación meteorológica que indica cómo responde a las peticiones de varios servicios.

Puede leer este diagrama del siguiente modo:

1. Si el estado del sistema es Shutdown (apagado), entonces puede responder a un mensaje restart(), reconfigure() o powerSave() (reiniciar, reconfigurar o ahorrar energía, respectivamente). La flecha sin etiqueta con la burbuja negra evidencia que el estado Shutdown es el estado inicial. Un mensaje restart() causa una transición a operación normal. Los mensajes powerSave() y reconfigure() producen una transición a un estado donde el sistema se reconfigura a sí mismo. El diagrama de estado muestra que la reconfiguración sólo se permite cuando el sistema ha estado apagado.

2. En el estado Running (operación), el sistema espera más mensajes. Si recibe un mensaje Shutdown(), el objeto regresa al estado apagado.

3. Si capta un mensaje reportWeather() (reporte del clima), el sistema avanza al estado Summarizing (resumir). Cuando el resumen está completo, el sistema avanza hacia un estado Transmitting (transmisión), donde la información se transfiere al sistema remoto. Luego regresa al estado Running.

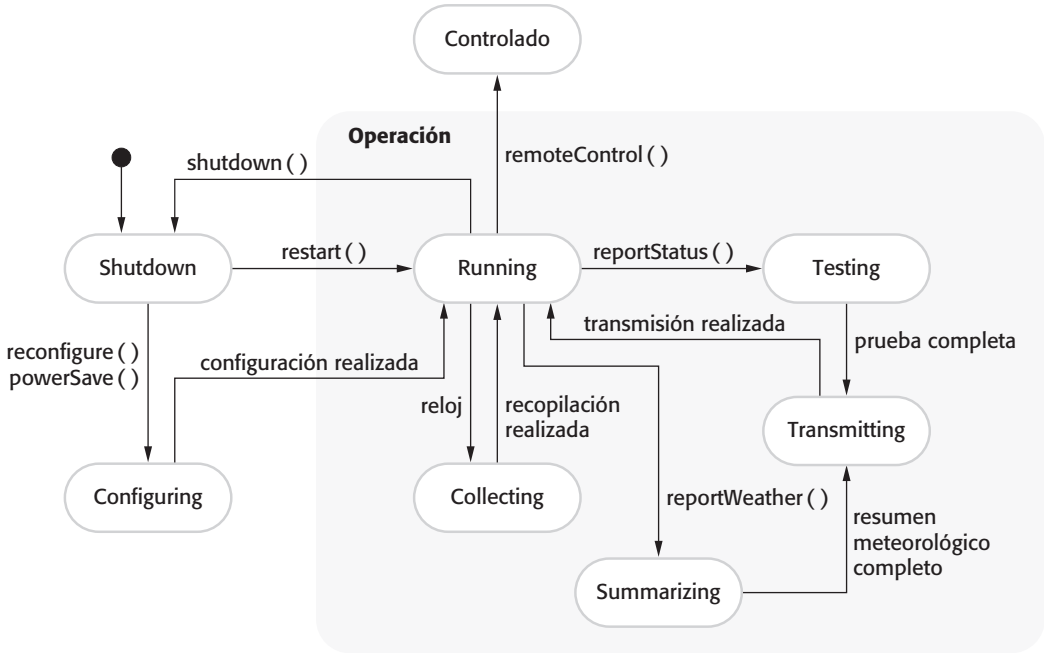


Figura 7.8 Diagrama de estado para la estación meteorológica

4. Si recibe un mensaje reportStatus() (reportar estatus), el sistema avanza hacia el estado Testing (probar), luego al estado Transmitting, antes de regresar al estado Running.
5. Si recibe una señal del reloj, el sistema se mueve hacia el estado Collecting (recolección), donde recaba los datos de los instrumentos. A cada instrumento se le instruye a su vez para recolectar sus datos de los sensores asociados.
6. Si recibe un mensaje remoteControl(), el sistema avanza hacia un estado controlado donde responde a diferentes conjuntos de mensajes desde la sala de control remoto. Éstos no se muestran en el diagrama.

Los diagramas de estado son modelos útiles de alto nivel de un sistema o la operación de un objeto. Por lo general, no se requiere un diagrama de estado para todos los objetos en el sistema. Muchos de los objetos en un sistema son relativamente simples y un modelo de estado añade detalle innecesario al diseño.

7.1.5 Especificación de interfaz

Una parte importante de cualquier proceso de diseño es la especificación de las interfaces entre los componentes en el diseño. Es necesario especificar las interfaces de modo que los objetos y subsistemas puedan diseñarse en paralelo. Una vez especificada la interfaz, los desarrolladores de otros objetos pueden suponer que se implementará la interfaz.

El diseño de interfaz se preocupa por la especificación del detalle de la interfaz hacia un objeto o un grupo de objetos. Esto significa definir las firmas y la semántica de los

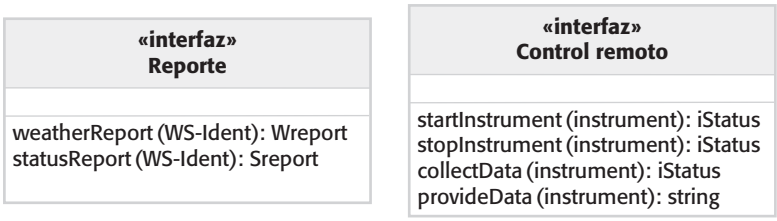


Figura 7.9 Interfaces de la estación meteorológica

servicios que ofrecerá el objeto o un grupo de objetos. Las interfaces pueden especificarse en el UML con la misma notación de un diagrama de clase. Sin embargo, no hay sección de atributos y debe incluirse el estereotipo UML «interface» en la parte del nombre. La semántica de la interfaz se define mediante el lenguaje de restricción de objeto (OCL). Esto se explica en el capítulo 17, donde se estudia la ingeniería de software basada en componentes. También muestra una forma alternativa para representar interfaces en el UML.

En un diseño de interfaz no se deben incluir detalles de la representación de datos, pues los atributos no se definen en una especificación de interfaz. Sin embargo, debe contener operaciones para acceder a los datos y actualizarlos. Puesto que la representación de datos está oculta, puede cambiar fácilmente sin afectar los objetos que usan dichos datos. Esto conduce a un diseño que inherentemente es más mantenible. Por ejemplo, una representación de arreglo en pila puede cambiarse a una representación en lista, sin afectar otros objetos que usen la pila. En contraste, con frecuencia tiene sentido exponer los atributos en un modelo de diseño estático, pues es la forma más compacta de ilustrar las características esenciales de los objetos.

No hay una relación simple 1:1 entre objetos e interfaces. El mismo objeto puede tener muchas interfaces, cada una de las cuales es un punto de vista de los métodos que ofrece. Esto se soporta directamente en Java, donde las interfaces se declaran por separado de los objetos, y los objetos "implementan" interfaces. De igual modo, puede accederse a un grupo de objetos a través de una sola interfaz.

La figura 7.9 indica dos interfaces que pueden definirse para la estación meteorológica. La interfaz de la izquierda es una interfaz de reporte que precisa los nombres de operación que se usan para generar reportes del clima y de estatus. Éstos se mapean directamente a operaciones en el objeto WeatherStation. La interfaz de control remoto proporciona cuatro operaciones, que se mapean en un solo método en el objeto WeatherStation. En este caso, las operaciones individuales se codifican en la cadena (*string*) de comando asociada con el método remoteControl, que se muestra en la figura 7.6.

7.2 Patrones de diseño

Los patrones de diseño se derivaron de ideas planteadas por Christopher Alexander (Alexander *et al*., 1977), quien sugirió que había ciertos patrones comunes de diseño de construcción que eran relativamente agradables y efectivos. El patrón es una descripción del problema y la esencia de su solución, de modo que la solución puede reutilizarse en diferentes configuraciones. El patrón no es una especificación detallada. Más bien, puede

Nombre del patrón: Observer
Descripción: Separa el despliegue del estado de un objeto del objeto en sí y permite el ofrecimiento de despliegues alternativos. Cuando cambia el estado del objeto, todos los despliegues se notifican automáticamente y se actualizan para reflejar el cambio.
Descripción del problema: En muchas situaciones hay que proporcionar múltiples despliegues de información del estado, tales como un despliegue gráfico y un despliegue tabular. Tal vez no se conozcan todos éstos cuando se especifica la información. Todas las presentaciones alternativas deben soportar la interacción y, cuando cambia el estado, los despliegues en su totalidad deben actualizarse.

Este patrón puede usarse en todas las situaciones en que se requiera más de un formato de despliegue para información del estado y donde no es necesario que el objeto mantenga la información del estado para conocer sobre los formatos específicos de despliegue utilizados.
Descripción de la solución: Esto implica dos objetos abstractos, Subject y Observer, y dos objetos concretos, ConcreteSubject (sujeto concreto) y ConcreteObject (objeto concreto), que heredan los atributos de los objetos abstractos relacionados. Los objetos abstractos contienen operaciones generales que son aplicables en todas las situaciones. El estado a desplegar se mantiene en ConcreteSubject, que hereda operaciones de Subject y le permite agregar y remover Observers (cada observador corresponde a un despliegue) y emite una notificación cuando cambia el estado.

ConcreteObserver (observador concreto) mantiene una copia del estado de ConcreteSubject e implementa la interfaz Update() de Observer que permite que dichas copias se conserven al paso. ConcreteObserver automáticamente despliega el estado y refleja los cambios siempre que se actualice el estado.
El modelo UML del patrón se ilustra en la figura 7.12.
Consecuencias: El sujeto sólo conoce al Observer abstracto y no los detalles de la clase concreta. Por lo tanto, existe un acoplamiento mínimo entre dichos objetos. Debido a esta falta de conocimiento, son imprácticas las optimizaciones que mejoran el rendimiento del despliegue. Los cambios al sujeto podrían generar un conjunto de actualizaciones vinculadas a observadores, de las cuales algunas quizá no sean necesarias.

Figura 7.10 El patrón Observer (observador)

considerarla como un descripción de sabiduría y experiencia acumuladas, una solución bien probada a un problema común.

Una cita del sitio Web de Hillside Group (http://hillside.net), que se dedica a mantener información acerca de patrones, resume su papel en la reutilización:

> *Los patrones y los lenguajes de patrón son formas de describir mejores prácticas, buenos diseños, y captan la experiencia de tal manera que es posible que otros reutilicen esta experiencia.*

Los patrones causaron un enorme impacto en el diseño de software orientado a objetos. Como son soluciones probadas a problemas comunes, se convirtieron en un vocabulario para hablar sobre un diseño. Por lo tanto, usted puede explicar su diseño al describir los patrones que utilizó. Esto es en particular verdadero para los patrones de diseño más conocidos que originalmente describió la "Banda de los cuatro" en su libro de patrones (Gamma *et al*., 1995). Otras descripciones de patrón en especial importantes son las publicadas en una serie de libros por autores de Siemens, una gran compañía tecnológica europea (Buschmann *et al*., 1996; Buschmann *et al*., 2007a; Buschmann *et al*., 2007b; Kircher y Jain, 2004; Schmidt *et al*., 2000).

Los patrones de diseño se asocian usualmente con el diseño orientado a objetos. Los patrones publicados se suelen apoyar en características de objetos como herencia y polimorfismo para dar generalidad. Sin embargo, el principio universal de encapsular

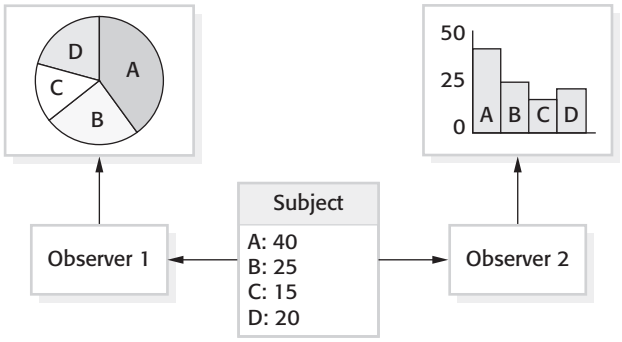


Figura 7.11 Despliegues múltiples

la experiencia en un patrón es igualmente aplicable a cualquier tipo de diseño de software. De este modo, usted podría tener patrones de configuración para sistemas COTS. Los patrones son una forma de reutilizar el conocimiento y la experiencia de otros diseñadores.

Los cuatro elementos esenciales de los patrones de diseño, definidos por la "Banda de los cuatro" en su libro de patrones, son:

1. Un nombre que sea una referencia significativa al patrón.
2. Una descripción del área problemática que enuncie cuándo puede aplicarse el patrón.
3. Una descripción de solución de las partes de la solución de diseño, sus relaciones y responsabilidades. No es una descripción concreta de diseño; es una plantilla para que una solución de diseño se instale en diferentes formas. Esto con frecuencia se expresa gráficamente y muestra las relaciones entre los objetos y las clases de objetos en la solución.
4. Un estado de las consecuencias, los resultados y las negociaciones, al aplicar el patrón. Lo anterior ayuda a los diseñadores a entender si es factible usar o no un patrón en una situación particular.

Gamma y sus coautores descomponen la descripción del problema en motivación (una descripción del porqué es útil el patrón) y aplicabilidad (una descripción de las situaciones en que puede usarse el patrón). Con la descripción de la solución, explican la estructura del patrón, los participantes, las colaboraciones y la implementación.

Para ilustrar la descripción del patrón, en el texto se usa el patrón Observer, tomado del libro de Gamma *et al*. (1995). Esto se muestra en la figura 7.10. En la descripción del texto, se emplean los cuatro elementos esenciales de descripción y también se incluye un breve enunciado sobre qué puede hacer el patrón. Este patrón se utiliza en situaciones donde se requieren diferentes presentaciones del estado de un objeto. Separa el objeto que debe desplegarse de las diferentes formas de presentación. Lo vemos en la figura 7.11, que muestra dos presentaciones gráficas del mismo conjunto de datos.

Las representaciones gráficas se usan por lo general para explicar las clases de objetos en patrones y sus relaciones. Ello complementa la descripción del patrón y agrega

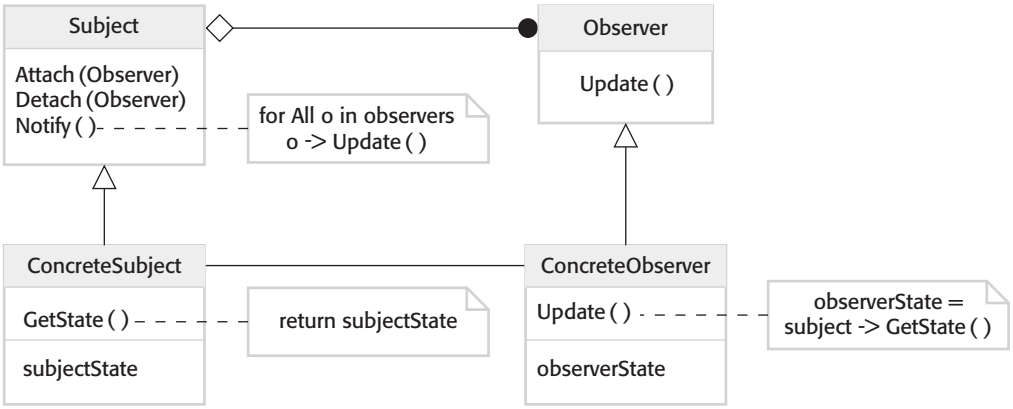


Figura 7.12 Modelo UML del patrón Observer

detalles a la descripción de la solución. La figura 7.12 es la representación en UML del patrón Observer.

Para usar patrones en su diseño, se debe reconocer que cualquier problema de diseño que enfrente es posible que tenga un patrón asociado para aplicarse. Los ejemplos de tales problemas, documentados en el libro de patrones original de la "Banda de los cuatro", incluyen:

1. Señalar a varios objetos que cambiaron el estado de algún otro objeto (patrón Observer).
2. Ordenar las interfaces en un número de objetos relacionados que a menudo se hayan desarrollado incrementalmente (patrón Façade, fachada).
3. Proporcionar una forma estándar para ingresar a los elementos en una colección, sin importar cómo se implementó dicha colección (patrón Iterator, iterador).
4. Permitir la posibilidad de extender la funcionalidad de una clase existente en tiempo de operación (patrón Decorator, decorador).

Los patrones soportan reutilización de concepto de alto nivel. Cuando intente reutilizar componentes ejecutables, estará restringido inevitablemente por decisiones de diseño detalladas que los implementadores tomaron de dichos componentes. Éstas varían desde los algoritmos particulares usados para implementar los componentes, hasta los objetos y tipos en las interfaces del componente. Cuando dichas decisiones de diseño entran en conflicto con los requerimientos particulares, la reutilización de componentes resulta imposible o introduce ineficiencias en su sistema. El uso de patrones significa que se reutilizan las ideas, pero podría adaptar la implementación para ajustarse al sistema que se desarrolla.

Cuando usted comienza el diseño de un sistema, quizá sea difícil saber, por adelantado, si necesitará un patrón particular. Por lo tanto, el uso de patrones en un proceso de diseño con frecuencia implica el desarrollo de un diseño, experimentar un problema y, luego, reconocer que puede usarse un patrón. En efecto, esto es posible si se enfoca en los 23 patrones de propósito general documentados en el libro de patrones original. Sin

embargo, si su problema es diferente, quizá descubra que es difícil encontrar un patrón adecuado entre los cientos de patrones propuestos.

Los patrones son una gran idea, pero para usarlos de manera efectiva se necesita experiencia en diseño de software. Hay que reconocer las situaciones donde se aplicaría un patrón. Los programadores inexpertos, incluso si leyeron los libros acerca de patrones, siempre descubrirán que es difícil decidir si deben reutilizar un patrón o necesitan desarrollar una solución de propósito especial.

7.3 Conflictos de implementación

La ingeniería de software incluye todas las actividades implicadas en el desarrollo de software, desde los requerimientos iniciales del sistema hasta el mantenimiento y la administración del sistema desplegado. Una etapa crítica de este proceso es, desde luego, la implementación del sistema, en la cual se crea una versión ejecutable del software. La implementación quizá requiera el desarrollo de programas en lenguajes de programación de alto o bajo niveles, o bien, la personalización y adaptación de sistemas comerciales genéricos para cubrir los requerimientos específicos de una organización.

Se supone que la mayoría de los lectores de este libro comprenderán los principios de programación y tendrán alguna experiencia al respecto. Como este capítulo tiene la intención de ofrecer un enfoque independiente de lenguaje, no se centró en conflictos de la buena práctica de programación, pues para esto se tendrían que usar ejemplos específicos de lenguaje. En su lugar, se introducen algunos aspectos de implementación que son muy importantes para la ingeniería de software que, por lo general, no se tocan en los textos de programación. Éstos son:

1. *Reutilización* La mayoría del software moderno se construye por la reutilización de los componentes o sistemas existentes. Cuando se desarrolla software, debe usarse el código existente tanto como sea posible.
2. *Administración de la configuración* Durante el proceso de desarrollo se crean muchas versiones diferentes de cada componente de software. Si usted no sigue la huella de dichas versiones en un sistema de gestión de configuración, estará proclive a incluir en su sistema las versiones equivocadas de dichos componentes.
3. *Desarrollo de huésped-objetivo* La producción de software no se ejecuta por lo general en la misma computadora que el entorno de desarrollo de software. En vez de ello, se diseña en una computadora (el sistema huésped) y se ejecuta en una computadora separada (el sistema objetivo). Los sistemas huésped y objetivo son algunas veces del mismo tipo, aunque suelen ser completamente diferentes.

7.3.1 Reutilización

De la década de 1970 a la de 1990, gran parte del nuevo software se desarrolló desde cero, al escribir todo el código en un lenguaje de programación de alto nivel. La única reutilización o software significativo era la reutilización de funciones y objetos en las

librerías de lenguaje de programación. Sin embargo, los costos y la presión por fechas significaban que este enfoque se volvería cada vez más inviable, sobre todo para sistemas comerciales y basados en Internet. En consecuencia, surgió un enfoque al desarrollo basado en la reutilización del software existente y ahora se emplea generalmente para sistemas empresariales, software científico y, cada vez más, en ingeniería de sistemas embebidos.

La reutilización de software es posible en algunos niveles diferentes:

1. *El nivel de abstracción* En este nivel no se reutiliza el software directamente, sino más bien se utiliza el conocimiento de abstracciones exitosas en el diseño de su software. Los patrones de diseño y los arquitectónicos (tratados en el capítulo 6) son vías de representación del conocimiento abstracto para la reutilización.
2. *El nivel objeto* En este nivel se reutilizan directamente los objetos de una librería en vez de escribir uno mismo en código. Para implementar este tipo de reutilización, se deben encontrar librerías adecuadas y descubrir si los objetos y métodos ofrecen la funcionalidad que se necesita. Por ejemplo, si usted requiere procesar mensajes de correo en un programa Java, tiene que usar objetos y métodos de una librería JavaMail.
3. *El nivel componente* Los componentes son colecciones de objetos y clases de objetos que operan en conjunto para brindar funciones y servicios relacionados. Con frecuencia se debe adaptar y extender el componente al agregar por cuenta propia cierto código. Un ejemplo de reutilización a nivel componente es donde usted construye su interfaz de usuario mediante un marco. Éste es un conjunto de clases de objetos generales que aplica manipulación de eventos, gestión de despliegue, etcétera. Agrega conexiones a los datos a desplegar y escribe el código para definir detalles de despliegue específicos, como plantilla de la pantalla y colores.
4. *El nivel sistema* En este nivel se reutilizan sistemas de aplicación completos. Usualmente esto implica cierto tipo de configuración de dichos sistemas. Puede hacerse al agregar y modificar el código (si reutiliza una línea de producto de software) o al usar la interfaz de configuración característica del sistema. La mayoría de los sistemas comerciales se diseñan ahora de esta forma, donde se adapta y reutilizan sistemas COTS (comerciales) genéricos. A veces este enfoque puede incluir la reutilización de muchos sistemas diferentes e integrarlos para crear un nuevo sistema.

Al reutilizar el software existente, es factible desarrollar nuevos sistemas más rápidamente, con menos riesgos de desarrollo y también costos menores. Puesto que el software reutilizado se probó en otras aplicaciones, debe ser más confiable que el software nuevo. Sin embargo, existen costos asociados con la reutilización:

1. Los costos del tiempo empleado en la búsqueda del software para reutilizar y valorar si cubre sus necesidades o no. Es posible que deba poner a prueba el software para asegurarse de que funcionará en su entorno, especialmente si éste es diferente de su entorno de desarrollo.
2. Donde sea aplicable, los costos por comprar el software reutilizable. Para sistemas comerciales grandes, dichos costos suelen ser muy elevados.

3. Los costos por adaptar y configurar los componentes de software o sistemas reutilizables, con la finalidad de reflejar los requerimientos del sistema que se desarrolla.

4. Los costos de integrar elementos de software reutilizable unos con otros (si usa software de diferentes fuentes) y con el nuevo código que haya desarrollado. Integrar software reutilizable de diferentes proveedores suele ser difícil y costoso, ya que los proveedores podrían hacer conjeturas conflictivas sobre cómo se reutilizará su software respectivo.

Cómo reutilizar el conocimiento y el software existentes sería el primer punto a considerar cuando se inicie un proyecto de desarrollo de software. Hay que contemplar las posibilidades de reutilización antes de diseñar el software a detalle, pues tal vez usted quiera adaptar su diseño para reutilización de los activos de software existentes. Como se estudió en el capítulo 2, en un proceso de desarrollo orientado a la reutilización uno busca elementos reutilizables y, luego, modifica los requerimientos y el diseño para hacer un mejor uso de ellos.

Para un gran número de sistemas de aplicación, la ingeniería de software significa en realidad reutilización de software. En consecuencia, en este libro se dedican a este tema varios capítulos de la sección de tecnologías de software (capítulos 16, 17 y 19).

7.3.2 Administración de la configuración

En el desarrollo de software, los cambios ocurren todo el tiempo, de modo que la administración del cambio es absolutamente esencial. Cuando un equipo de individuos desarrolla software, hay que cerciorarse de que los miembros del equipo no interfieran con el trabajo de los demás. Esto es, si dos personas trabajan sobre un componente, los cambios deben coordinarse. De otro modo, un programador podría realizar cambios y sobrescribir en el trabajo de otro. También se debe garantizar que todos tengan acceso a las versiones más actualizadas de componentes de software; de lo contrario, los desarrolladores pueden rehacer lo ya hecho. Cuando algo salga mal con una nueva versión de un sistema, se debe poder retroceder a una versión operativa del sistema o componente.

Administración de la configuración es el nombre dado al proceso general de gestionar un sistema de software cambiante. La meta de la administración de la configuración es apoyar el proceso de integración del sistema, de modo que todos los desarrolladores tengan acceso en una forma controlada al código del proyecto y a los documentos, descubrir qué cambios se realizaron, así como compilar y vincular componentes para crear un sistema. Por lo tanto, hay tres actividades fundamentales en la administración de la configuración:

1. Gestión de versiones, donde se da soporte para hacer un seguimiento de las diferentes versiones de los componentes de software. Los sistemas de gestión de versiones incluyen facilidades para que el desarrollo esté coordinado por varios programadores. Esto evita que un desarrollador sobrescriba un código que haya sido enviado al sistema por alguien más.

2. Integración de sistema, donde se da soporte para ayudar a los desarrolladores a definir qué versiones de componentes se usan para crear cada versión de un sistema. Luego, esta descripción se utiliza para elaborar automáticamente un sistema al compilar y vincular los componentes requeridos.

3. Rastreo de problemas, donde se da soporte para que los usuarios reporten bugs y otros problemas, y también para que todos los desarrolladores sepan quién trabaja en dichos problemas y cuándo se corrigen.

Las herramientas de administración de la configuración de software soportan cada una de las actividades anteriores. Dichas herramientas pueden diseñarse para trabajar en conjunto en un sistema de gestión de cambio global, como ClearCase (Bellagio y Milligan, 2005). En los sistemas de administración de configuración integrada, se diseñan en conjunto las herramientas de gestión de versiones, integración de sistema y rastreo del problema. Comparten un estilo de interfaz de usuario y se integran a través de un repositorio de código común.

Alternativamente, pueden usarse herramientas por separado, instaladas en un entorno de desarrollo integrado. La gestión de versiones puede soportarse mediante un sistema de gestión de versiones como Subversion (Pilato *et al*., 2008), que puede soportar desarrollo multisitio o multiequipo. El soporte a la integración de sistema podría construirse en el lenguaje o apoyarse en un conjunto de herramientas por separado, como el sistema de construcción GNU. Esto incluye lo que quizá sea la herramienta de integración mejor conocida, hecha en Unix. El rastreo de bugs o los sistemas de rastreo de conflictos, como Bugzilla, se usan para reportar bugs y otros conflictos, así como para seguir la pista sobre si se corrigieron o no.

En virtud de su importancia en la ingeniería de software profesional, en el capítulo 25 se analizan con más detenimiento la administración de la configuración y del cambio.

7.3.3 Desarrollo huésped-objetivo

La mayoría del desarrollo de software se basa en un modelo huésped-objetivo. El software se desarrolla en una computadora (el huésped), aunque opera en una máquina separada (el objetivo). En un sentido más amplio, puede hablarse de una plataforma de desarrollo y una plataforma de ejecución. Una plataforma es más que sólo hardware. Incluye el sistema operativo instalado más otro software de soporte como un sistema de gestión de base de datos o, para plataformas de desarrollo, un entorno de desarrollo interactivo.

En ocasiones, las plataformas de desarrollo y ejecución son iguales, lo que posibilita diseñar el software y ponerlo a prueba en la misma máquina. Sin embargo, es más común que sean diferentes, de modo que es necesario mover el software desarrollado a la plataforma de ejecución para ponerlo a prueba, u operar un simulador en su máquina de desarrollo.

Los simuladores se usan con frecuencia al elaborar sistemas embebidos. Se simulan dispositivos de hardware, tales como sensores, y los eventos en el entorno donde el sistema se podrá en funcionamiento. Los simuladores aceleran el proceso de desarrollo para sistemas embebidos, pues cada desarrollador puede contar con su propia plataforma de ejecución, sin tener que descargar el software al hardware objetivo. No obstante, los simuladores son costosos de desarrollar y, por lo tanto, a menudo sólo se encuentran disponibles para las arquitecturas de hardware más conocidas.

Si el sistema objetivo tiene instalado middleware u otro software que necesite usar, en tal caso el sistema se debe poner a prueba utilizando dicho software. Tal vez no resulte práctico instalar dicho software en su máquina de desarrollo, incluso si es la misma que la plataforma objetivo, debido a restricciones de licencia. Ante tales circunstancias, usted necesita transferir su código desarrollado a la plataforma de ejecución, con la finalidad de poner a prueba el sistema.

Diagramas de despliegue UML

Los diagramas de despliegue UML muestran cómo los componentes de software se despliegan físicamente en los procesadores; es decir, el diagrama de despliegue muestra el hardware y el software en el sistema, así como el middleware usado para conectar los diferentes componentes en el sistema. En esencia, los diagramas de despliegue se pueden considerar como una forma de definir y documentar el entorno objetivo.

http://www.SoftwareEngineering-9.com/Web/Deployment/

Una plataforma de desarrollo de software debe ofrecer una variedad de herramientas para soportar los procesos de ingeniería de software. Éstas pueden incluir:

1. Un compilador integrado y un sistema de edición dirigida por sintaxis que le permitan crear, editar y compilar código.
2. Un sistema de depuración de lenguaje.
3. Herramientas de edición gráfica, tales como las herramientas para editar modelos UML.
4. Herramientas de prueba, como JUnit (Massol, 2003) que operen automáticamente un conjunto de pruebas sobre una nueva versión de un programa.
5. Herramientas de apoyo de proyecto que le ayuden a organizar el código para diferentes proyectos de desarrollo.

Al igual que dichas herramientas estándar, su sistema de desarrollo puede incluir herramientas más especializadas, como analizadores estáticos (que se estudian en el capítulo 15). Normalmente, los entornos de desarrollo para equipos también contemplan un servidor compartido que opera un sistema de administración del cambio y la configuración y, si acaso, un sistema para soportar la gestión de requerimientos.

Las herramientas de desarrollo de software se agrupan con frecuencia para crear un entorno de desarrollo integrado (IDE), que es un conjunto de herramientas de software que apoyan diferentes aspectos del desarrollo de software, dentro de cierto marco común e interfaz de usuario. Por lo común, los IDE se crean para apoyar el desarrollo en un lenguaje de programación específico, como Java. El lenguaje IDE puede elaborarse especialmente, o ser una ejemplificación de un IDE de propósito general, con herramientas de apoyo a lenguaje específico.

Un IDE de propósito general es un marco para colocar herramientas de software, que brinden facilidades de gestión de datos para el software a desarrollar, y mecanismos de integración, que permitan a las herramientas trabajar en conjunto. El entorno Eclipse (Carlson, 2005) es el IDE de propósito general mejor conocido. Este entorno se basa en una arquitectura de conexión (*plug-in*), de modo que pueda especializarse para diferentes lenguajes y dominios de aplicación (Clayberg y Rubel, 2006). Por consiguiente, es posible instalar Eclipse y personalizarlo según sus necesidades específicas al agregar plug-ins (enchufables o conectables), para soportar el desarrollo de sistemas en red en Java o ingeniería de sistemas embebidos usando C.

Como parte del proceso de desarrollo, se requiere tomar decisiones sobre cómo se desplegará el software desarrollado en la plataforma objetivo. Esto es directo para sistemas

embebidos, donde el objetivo es usualmente una sola computadora. Sin embargo, para sistemas distribuidos, es necesario decidir sobre las plataformas específicas donde se desplegarán los componentes. Los conflictos que hay que considerar al tomar esta decisión son:

1. *Los requerimientos de hardware y software de un componente* Si un componente se diseña para una arquitectura de hardware específica, o se apoya en algún otro sistema de software, tiene que desplegarse por supuesto en una plataforma que brinde el soporte requerido de hardware y software.
2. *Los requerimientos de disponibilidad del sistema* Los sistemas de alta disponibilidad pueden necesitar que los componentes se desplieguen en más de una plataforma. Esto significa que, en el caso de una falla de plataforma, esté disponible una implementación alternativa del componente.
3. *Comunicaciones de componentes* Si hay un alto nivel de tráfico de comunicaciones entre componentes, por lo general tiene sentido desplegarlos en la misma plataforma o en plataformas que estén físicamente cercanas entre sí. Esto reduce la latencia de comunicaciones, es decir, la demora entre el tiempo que transcurre desde el momento en que un componente envía un mensaje hasta que otro lo recibe.

Puede documentar sus decisiones sobre el despliegue de hardware y software usando diagramas de despliegue UML, que muestran cómo los componentes de software se distribuyen a través de plataformas de hardware.

Si desarrolla un sistema embebido, quizá deba tomar en cuenta las características del objetivo, como su tamaño físico, capacidades de poder, necesidad de respuestas en tiempo real para eventos de sensor, características físicas de los actuadores, y sistema operativo de tiempo real. En el capítulo 20 se estudia la ingeniería de los sistemas embebidos.

7.4 Desarrollo de código abierto

El desarrollo de código abierto es un enfoque al desarrollo de software en que se publica el código de un sistema de software y se invita a voluntarios a participar en el proceso de desarrollo (Raymond, 2001). Sus raíces están en la Free Software Foundation (http://www.fsf.org), que aboga porque el código fuente no debe ser propietario sino, más bien, tiene que estar siempre disponible para que los usuarios lo examinen y modifiquen como deseen. Existía la idea de que el código estaría controlado y sería desarrollado por un pequeño grupo central, en vez de por usuarios del código.

El software de código abierto extendió esta idea al usar Internet para reclutar a una población mucho mayor de desarrolladores voluntarios. La mayoría de ellos también son usuarios del código. En principio al menos, cualquier contribuyente a un proyecto de código abierto puede reportar y corregir bugs, así como proponer nuevas características y funcionalidades. Sin embargo, en la práctica, los sistemas exitosos de código abierto aún se apoyan en un grupo central de desarrolladores que controlan los cambios al software.

Desde luego, el producto mejor conocido de código abierto es el sistema operativo Linux, utilizado ampliamente como sistema servidor y, cada vez más, como un entorno de escritorio. Otros productos de código abierto importantes son Java, el servidor Web

Apache y el sistema de gestión de base de datos mySQL. Los protagonistas principales en la industria de cómputo, como IBM y Sun, soportan el movimiento de código abierto y basan su software en productos de código abierto. Existen miles de otros sistemas y componentes de código abierto menos conocidos que también podrían usarse.

Por lo general, es muy barato o incluso gratuito adquirir software de código abierto. Usualmente, el software de código abierto se descarga sin costo. Sin embargo, si usted quiere documentación y soporte, entonces tal vez deba pagar por ello; aún así, los costos son por lo común bastante bajos. El otro beneficio clave para usar productos de código abierto es que los sistemas de código abierto mayores son casi siempre muy confiables. La razón de esto es una gran población de usuarios que quiere corregir los problemas por sí misma, en lugar de reportarlos al desarrollador y esperar una nueva versión del sistema. Los bugs se descubren y reparan con más rapidez que lo que normalmente sería posible con software propietario.

Para una compañía que desarrolla software, existen dos conflictos de código abierto que debe considerar:

1. ¿El producto que se desarrollará deberá usar componentes de código abierto?
2. ¿Deberá usarse un enfoque de código abierto para el desarrollo del software?

Las respuestas a dichas preguntas dependen del tipo de software que se desarrollará, así como de los antecedentes y la experiencia del equipo de desarrollo.

Si usted diseña un producto de software para su venta, entonces resultan críticos tanto el tiempo en que sale al mercado como la reducción en costos. Si se desarrolla en un dominio donde estén disponibles sistemas en código abierto de alta calidad, puede ahorrar tiempo y dinero al usar dichos sistemas. Sin embargo, si usted desarrolla software para un conjunto específico de requerimientos organizativos, entonces quizás el uso de componentes de código abierto no sea una opción. Tal vez tenga que integrar su software con sistemas existentes que sean compatibles con los sistemas en código abierto disponibles. No obstante, incluso entonces podría ser más rápido y barato modificar el sistema en código abierto, en vez de volver a desarrollar la funcionalidad que necesita.

Cada vez más compañías de productos usan un enfoque de código abierto para el desarrollo. Sus modelos empresariales no dependen de la venta de un producto de software, sino de la comercialización del soporte para dicho producto. Consideran que involucrar a la comunidad de código abierto permitirá que el software se desarrolle de manera más económica, más rápida y creará una comunidad de usuarios para el software. A pesar de ello, de nuevo, esto sólo es aplicable realmente para productos de software en general, y no para aplicaciones específicas de la organización.

Muchas compañías creen que adoptar un enfoque de código abierto revelará conocimiento empresarial confidencial a sus competidores y, por consiguiente, son reticentes a adoptar tal modelo de desarrollo. No obstante, si usted trabaja en una pequeña compañía y abre la fuente de su software, esto puede garantizar a los clientes que podrán soportar el software en caso de que la compañía salga del mercado.

Publicar el código de un sistema no significa que la comunidad en general necesariamente ayudará con su desarrollo. Los productos más exitosos de código abierto han sido productos de plataforma, en vez de sistemas de aplicación. Hay un número limitado de desarrolladores que pueden interesarse en sistemas de aplicación especializados. En sí, elaborar un sistema de software en código abierto no garantiza la inclusión de la comunidad.

7.4.1 Licencia de código abierto

Aunque un principio fundamental del desarrollo en código abierto es que el código fuente debe estar disponible por entero, esto no significa que cualquiera puede hacer lo que desee con el código. Por ley, el desarrollador del código (una compañía o un individuo) todavía es propietario del código. Puede colocar restricciones sobre cómo se le utiliza al incluir condiciones legales en una licencia de software de código abierto (St. Laurent, 2004). Algunos desarrolladores de código abierto creen que si un componente de código abierto se usa para desarrollar un nuevo sistema, entonces dicho sistema también debe ser de código abierto. Otros están satisfechos de que su código se use sin esta restricción. Los sistemas desarrollados pueden ser propietarios y venderse como sistemas de código cerrado.

La mayoría de las licencias de código abierto se derivan de uno de tres modelos generales:

1. La licencia pública general GNU se conoce como licencia "recíproca"; de manera simple, significa que si usted usa software de código abierto que esté permitido bajo la licencia GPL, entonces debe hacer que dicho software sea de código abierto.
2. La licencia pública menos general GNU es una variante de la licencia anterior, en la que usted puede escribir componentes que se vinculen con el código abierto, sin tener que publicar el código de dichos componentes. Sin embargo, si cambia el componente permitido, entonces debe publicar éste como código abierto.
3. La licencia Berkeley Standard Distribution es una licencia no recíproca, lo cual significa que usted no está obligado a volver a publicar algún cambio o modificación al código abierto. Puede incluir el código en sistemas propietarios que se vendan. Si usa componentes de código abierto, debe reconocer al creador original del código.

Los temas sobre permisos son importantes porque si usa software de código abierto como parte de un producto de software, entonces tal vez esté obligado por los términos de la licencia a hacer que su propio producto sea de código abierto. Si trata de vender su software, quizá desee mantenerlo en secreto. Esto significa que tal vez quiera evitar el uso de software de código abierto con licencia GPL en su desarrollo.

Si construye un software que opere en una plataforma de código abierto, como Linux, en tal caso las licencias no son problema. Sin embargo, tan pronto como comience a incluir componentes de código abierto en su software, necesita establecer procesos y bases de datos para seguir la pista de lo que se usó y sus condiciones de licencia. Bayersdorfer (2007) sugiere que las compañías que administran proyectos que usan código abierto deben:

1. Establecer un sistema para mantener la información sobre los componentes de código abierto que se descargan y usan. Tienen que conservar una copia de la licencia para cada componente que sea válida al momento en que se usó el componente. Las licencias suelen cambiar, así que necesita conocer las condiciones acordadas.
2. Estar al tanto de los diferentes tipos de licencias y entender cómo está autorizado un componente antes de usarlo. Puede decidir el uso de un componente en un sistema, pero no en otro, porque planea usar dichos sistemas en diferentes formas.

3. Estar al tanto de las rutas de evolución para los componentes. Necesita conocer un poco sobre el proyecto de código abierto donde se desarrollaron los componentes, para entender cómo pueden cambiar en el futuro.
4. Educar al personal acerca del código abierto. No es suficiente tener procedimientos para asegurar el cumplimiento de las condiciones de la licencia. También es preciso educar a los desarrolladores sobre el código abierto y el permiso de éste.
5. Tener sistemas de auditoría. Los desarrolladores, con plazos ajustados, pueden sentirse tentados a quebrantar los términos de una licencia. Si es posible, debe tener software para detectar y evitar esto último.
6. Participar en la comunidad de código abierto. Si se apoya en productos de código abierto, debe participar en la comunidad y ayudar a apoyar su desarrollo.

El modelo empresarial de software está cambiando. Se ha vuelto cada vez más difícil edificar una empresa mediante la venta de sistemas de software especializado. Muchas compañías prefieren hacer su software en código abierto y entonces vender soporte y consultoría a los usuarios del software. Es probable que esta tendencia se incremente, con el uso creciente de software de código abierto y con cada vez más software disponible de esta forma.

PUNTOS CLAVE

- El diseño y la implementación del software son actividades entrelazadas. El nivel de detalle en el diseño depende del tipo de sistema a desarrollar y de si se usa un enfoque dirigido por un plan o uno ágil.
- Los procesos del diseño orientado a objetos incluyen actividades para diseñar la arquitectura del sistema, identificar objetos en el sistema, describir el diseño mediante diferentes modelos de objeto y documentar las interfaces de componente.
- Durante un proceso de diseño orientado a objetos, puede elaborarse una variedad de modelos diferentes. En ellos se incluyen modelos estáticos (modelos de clase, modelos de generalización, modelos de asociación) y modelos dinámicos (modelos de secuencia, modelos de máquina de estado).
- Las interfaces de componente deben definirse con precisión, de modo que otros objetos puedan usarlos. Para definir interfaces es posible usar un estereotipo de interfaz UML.
- Cuando se desarrolla software, siempre debe considerarse la posibilidad de reutilizar el software existente, ya sea como componentes, servicios o sistemas completos.
- La administración de la configuración es el proceso de gestionar los cambios a un sistema de software en evolución. Es esencial cuando un equipo de personas coopera para desarrollar software.

- La mayoría del desarrollo de software es desarrollo huésped-objetivo. Se usa un IDE en una máquina para desarrollar el huésped, que se transfiere a una máquina objetivo para su ejecución.
- El desarrollo de código abierto requiere hacer públicamente disponible el código fuente de un sistema. Esto significa que muchos individuos tienen la posibilidad de proponer cambios y mejoras al software.

LECTURAS SUGERIDAS

Design Patterns: Elements of Reusable Object-oriented Software. Éste es el manual original de patrones de hardware que introdujo los patrones de software a una amplia comunidad. (E. Gamma, R. Helm, R. Johnson y J. Vlissides, Addison-Wesley, 1995.)

Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd edition. Larman escribe claramente sobre el diseño orientado a objetos y también analiza el uso del UML. Ésta es una buena introducción al uso de patrones en el proceso de diseño. (C. Larman, Prentice Hall, 2004.)

Producing Open Source Software: How to Run a Successful Free Software Project. Este libro es una guía completa de los antecedentes del software de código abierto, los conflictos de permiso y las prácticas de operar un proyecto de desarrollo en código abierto. (K. Fogel, O'Reilly Media Inc., 2008.)

En el capítulo 16 se sugieren más lecturas acerca de la reutilización de software, y en el capítulo 25 otras referentes a la administración de la configuración.

EJERCICIOS

7.1. Con la notación estructurada que se muestra en la figura 7.3, especifique los casos de uso de la estación meteorológica para Report status (reporte de estatus) y Reconfigure (reconfigurar). Tiene que hacer conjeturas razonables acerca de la funcionalidad que se requiere aquí.

7.2. Suponga que el MHC-PMS se desarrollará usando un enfoque orientado a objetos. Dibuje un diagrama de caso de uso, que muestre al menos seis posibles casos de uso para este sistema.

7.3. Con la notación gráfica UML para clases de objetos, diseñe las siguientes clases de objetos, e identifique los atributos y las operaciones. Use su experiencia para decidir sobre los atributos y las operaciones que deban asociarse con estos objetos:

- Teléfono.
- Impresora para computadora personal.
- Sistema de estéreo personal.
- Cuenta bancaria.
- Catálogo de biblioteca.

7.4. Con los objetos de la estación meteorológica, identificados en la figura 7.6 como punto de partida, identifique más objetos que puedan usarse en este sistema. Diseñe una jerarquía de herencia para los objetos que haya identificado.

7.5. Desarrolle el diseño de la estación meteorológica para mostrar la interacción entre el subsistema de recolección de datos y los instrumentos que recolectan datos meteorológicos. Utilice diagramas de secuencia para mostrar esta interacción.

7.6. Identifique posibles objetos en los siguientes sistemas y desarrolle para ellos un diseño orientado a objetos. Puede hacer conjeturas razonables sobre los sistemas cuando derive el diseño.

- Un sistema de diario grupal y administración del tiempo tiene la intención de apoyar los horarios de las reuniones y citas de un grupo de compañeros de trabajo. Cuando se hace una cita para muchas personas, el sistema encuentra un espacio común en cada uno de sus diarios y arregla la cita para esa hora. Si no hay espacios comunes disponibles, interactúa con el usuario para reordenar su diario personal, con la finalidad de hacer espacio para la cita.
- Una estación de llenado (estación de gasolina) se configurará para operación completamente automatizada. Los conductores pasan su tarjeta de crédito a través de un lector conectado a la bomba de gasolina; la tarjeta se verifica mediante comunicación con una computadora en la compañía de crédito, y se establece un límite de combustible. Luego, el conductor puede tomar el combustible requerido. Cuando se completa la entrega de combustible y la manguera de la bomba regresa a su soporte, el costo del combustible surtido se carga a la cuenta de la tarjeta de crédito del conductor. La tarjeta de crédito se regresa después de la deducción. Si la tarjeta es inválida, la bomba la devuelve antes de despachar el combustible.

7.7. Dibuje un diagrama de secuencia que muestre las interacciones de los objetos en un sistema de diario grupal, cuando un grupo de individuos organizan una reunión.

7.8. Dibuje un diagrama de estado UML que señale los posibles cambios de estado en el diario grupal o en el sistema de llenado de la estación.

7.9. Con ejemplos, explique por qué es importante la administración de la configuración, cuando un equipo de individuos desarrolla un producto de software.

7.10. Una pequeña compañía desarrolló un producto especializado que se configura de manera especial para cada cliente. Los clientes nuevos, por lo general, tienen requerimientos específicos para incorporar en su sistema, y pagan para que esto se desarrolle. La compañía tiene oportunidad de licitar para un nuevo contrato, el cual representaría más del doble de su base de clientes. El nuevo cliente también quiere tener cierta participación en la configuración del sistema. Explique por qué, en estas circunstancias, puede ser buena idea que la compañía que posee el software lo convierta en código abierto.

REFERENCIAS

Abbott, R. (1983). "Program Design by Informal English Descriptions". *Comm. ACM,* **26** (11), 882–94.

Alexander, C., Ishikawa, S. y Silverstein, M. (1977). *A Pattern Language: Towns, Building, Construction.* Oxford: Oxford University Press.

Bayersdorfer, M. (2007). "Managing a Project with Open Source Components". *ACM Interactions,* **14** (6), 33–4.

Beck, K. y Cunningham, W. (1989). "A Laboratory for Teaching Object-Oriented Thinking". *Proc. OOPSLA' 89* (Conference on Object-oriented Programming, Systems, Languages and Applications), ACM Press. 1–6.

Bellagio, D. E. y Milligan, T. J. (2005). *Software Configuration Management Strategies and IBM Rational Clearcase: A Practical Introduction.* Boston: Pearson Education (IBM Press).

Buschmann, F., Henney, K. y Schmidt, D. C. (2007a). *Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.* New York: John Wiley & Sons.

Buschmann, F., Henney, K. y Schmidt, D. C. (2007b). *Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* New York: John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H. y Sommerlad, P. (1996). *Pattern-oriented Software Architecture Volume 1: A System of Patterns.* New York: John Wiley & Sons.

Carlson, D. (2005). *Eclipse Distilled.* Boston: Addison-Wesley.

Clayberg, E. y Rubel, D. (2006). *Eclipse: Building Commercial-Quality Plug-Ins.* Boston: Addison Wesley.

Coad, P. y Yourdon, E. (1990). *Object-oriented Analysis.* Englewood Cliffs, NJ: Prentice Hall.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley.

Harel, D. (1987). "Statecharts: A Visual Formalism for Complex Systems". *Sci. Comput. Programming,* **8** (3), 231–74.

Kircher, M. y Jain, P. (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.* New York: John Wiley & Sons.

Massol, V. (2003). *JUnit in Action.* Greenwich, CT: Manning Publications.

Pilato, C., Collins-Sussman, B. y Fitzpatrick, B. (2008). *Version Control with Subversion.* Sebastopol, Calif.: O'Reilly Media Inc.

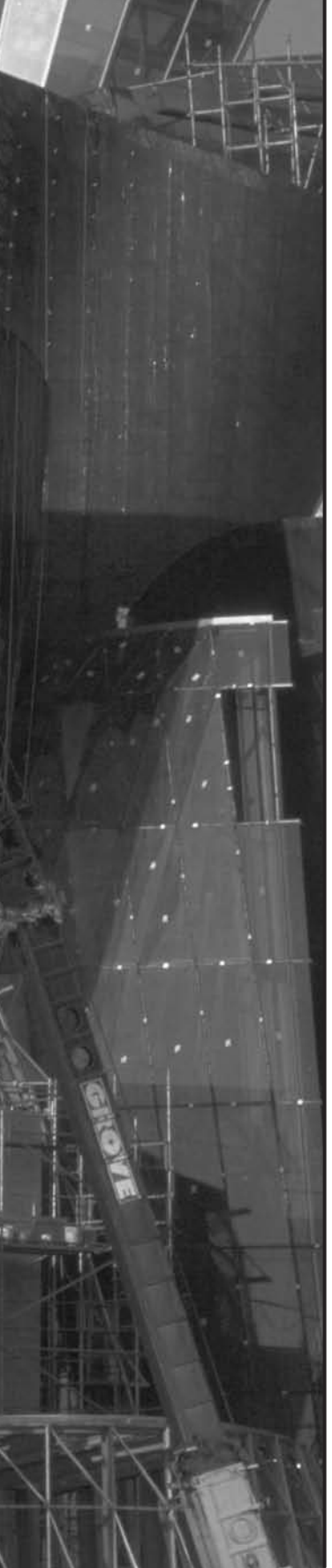
Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* Sebastopol. Calif.: O'Reilly Media, Inc.

Schmidt, D., Stal, M., Rohnert, H. y Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* New York: John Wiley & Sons.

Shlaer, S. y Mellor, S. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data.* Englewood Cliffs, NJ: Yourdon Press.

St. Laurent, A. (2004). *Understanding Open Source and Free Software Licensing.* Sebastopol, Calif.: O'Reilly Media Inc.

Wirfs-Brock, R., Wilkerson, B. y Weiner, L. (1990). *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice Hall.



8

Pruebas de software

Objetivos

El objetivo de este capítulo es introducirlo a las pruebas del software y los procesos necesarios para tales pruebas. Al estudiar este capítulo:

- comprenderá las etapas de las pruebas, desde las pruebas durante el desarrollo hasta la prueba de aceptación por los clientes del sistema;
- se introducirá en las técnicas que ayudan a elegir casos de prueba que se ponen en funcionamiento para descubrir los defectos del programa;
- entenderá el desarrollo de la primera prueba, donde se diseñan pruebas antes de escribir el código, las cuales operan automáticamente;
- conocerá las diferencias importantes entre pruebas de componente, de sistema y de liberación, y estará al tanto de los procesos y las técnicas de prueba del usuario.

Contenido

8.1 Pruebas de desarrollo
8.2 Desarrollo dirigido por pruebas
8.3 Pruebas de versión
8.4 Pruebas de usuario

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa antes de usarlo. Al probar el software, se ejecuta un programa con datos artificiales. Hay que verificar los resultados de la prueba que se opera para buscar errores, anomalías o información de atributos no funcionales del programa.

El proceso de prueba tiene dos metas distintas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos. Para el software personalizado, esto significa que en el documento de requerimientos debe haber, por lo menos, una prueba por cada requerimiento. Para los productos de software genérico, esto quiere decir que tiene que haber pruebas para todas las características del sistema, junto con combinaciones de dichas características que se incorporarán en la liberación del producto.
2. Encontrar situaciones donde el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación. Tales situaciones son consecuencia de defectos del software. La prueba de defectos tiene la finalidad de erradicar el comportamiento indeseable del sistema, como caídas del sistema, interacciones indeseadas con otros sistemas, cálculos incorrectos y corrupción de datos.

La primera meta conduce a la prueba de validación; en ella, se espera que el sistema se desempeñe de manera correcta mediante un conjunto dado de casos de prueba, que refleje el uso previsto del sistema. La segunda meta se orienta a pruebas de defectos, donde los casos de prueba se diseñan para presentar los defectos. Los casos de prueba en las pruebas de defecto pueden ser deliberadamente confusos y no necesitan expresar cómo se usa normalmente el sistema. Desde luego, no hay frontera definida entre estos dos enfoques de pruebas. Durante las pruebas de validación, usted descubrirá defectos en el sistema; en tanto que durante las pruebas de defecto algunas de las pruebas demostrarán que el programa cumple con sus requerimientos.

El diagrama de la figura 8.1 ayuda a explicar las diferencias entre las pruebas de validación y de defecto. Piense en el sistema que va a probar como si fuera una caja negra. El sistema acepta entradas desde algún conjunto de entradas I y genera salidas en un conjunto de salidas O. Algunas de las salidas serán erróneas. Son las salidas en el conjunto O_e que el sistema genera en respuesta a las entradas en el conjunto I_e. La prioridad en las pruebas de defecto es encontrar dichas entradas en el conjunto I_e porque ellas revelan problemas con el sistema. Las pruebas de validación involucran pruebas con entradas correctas que están fuera de I_e y estimulan al sistema para generar las salidas correctas previstas.

Las pruebas no pueden demostrar que el software esté exento de defectos o que se comportará como se especifica en cualquier circunstancia. Siempre es posible que una prueba que usted pase por alto descubra más problemas con el sistema. Como afirma de forma elocuente Edsger Dijkstra, uno de los primeros contribuyentes al desarrollo de la ingeniería de software (Dijkstra *et al*., 1972):

Las pruebas pueden mostrar sólo la presencia de errores, mas no su ausencia.

Las pruebas se consideran parte de un proceso más amplio de verificación y validación (V&V) del software. Aunque ambas no son lo mismo, se confunden con frecuencia. Barry

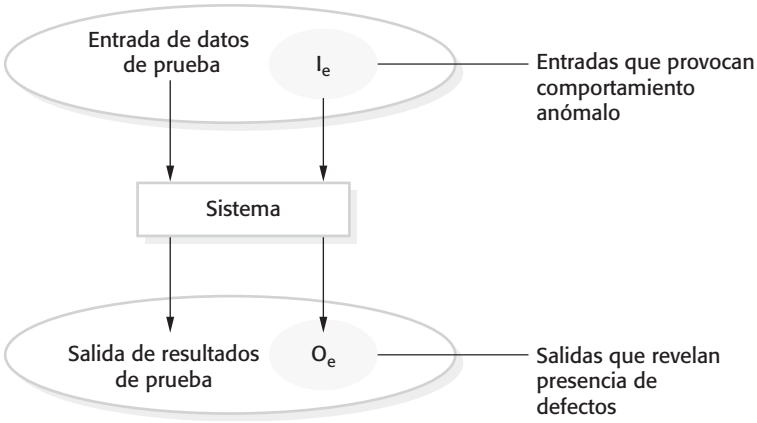


Figura 8.1 Modelo de entrada y salida de una prueba de programa

Boehm, pionero de la ingeniería de software, expresó de manera breve la diferencia entre las dos (Boehm, 1979):

- "Validación: ¿construimos el producto correcto?".
- "Verificación: ¿construimos bien el producto?".

Los procesos de verificación y validación buscan comprobar que el software por desarrollar cumpla con sus especificaciones, y brinde la funcionalidad deseada por las personas que pagan por el software. Dichos procesos de comprobación comienzan tan pronto como están disponibles los requerimientos y continúan a través de todas las etapas del proceso de desarrollo.

La finalidad de la verificación es comprobar que el software cumpla con su funcionalidad y con los requerimientos no funcionales establecidos. Sin embargo, la validación es un proceso más general. La meta de la validación es garantizar que el software cumpla con las expectativas del cliente. Va más allá del simple hecho de comprobar la conformidad con la especificación, para demostrar que el software hace lo que el cliente espera que haga. La validación es esencial pues, como se estudió en el capítulo 4, las especificaciones de requerimientos no siempre reflejan los deseos o las necesidades reales de los clientes y usuarios del sistema.

El objetivo final de los procesos de verificación y validación es establecer confianza de que el sistema de software es "adecuado". Esto significa que el sistema tiene que ser lo bastante eficaz para su uso esperado. El nivel de confianza adquirido depende tanto del propósito del sistema y las expectativas de los usuarios del sistema, como del entorno del mercado actual para el sistema:

1. *Propósito del software* Cuanto más crítico sea el software, más importante debe ser su confiabilidad. Por ejemplo, el nivel de confianza requerido para que se use el software en el control de un sistema crítico de seguridad es mucho mayor que el requerido para un prototipo desarrollado para demostrar nuevas ideas del producto.
2. *Expectativas del usuario* Debido a su experiencia con software no confiable y plagado de errores, muchos usuarios tienen pocas expectativas de la calidad del software, por lo que no se sorprenden cuando éste falla. Al instalar un sistema, los usuarios

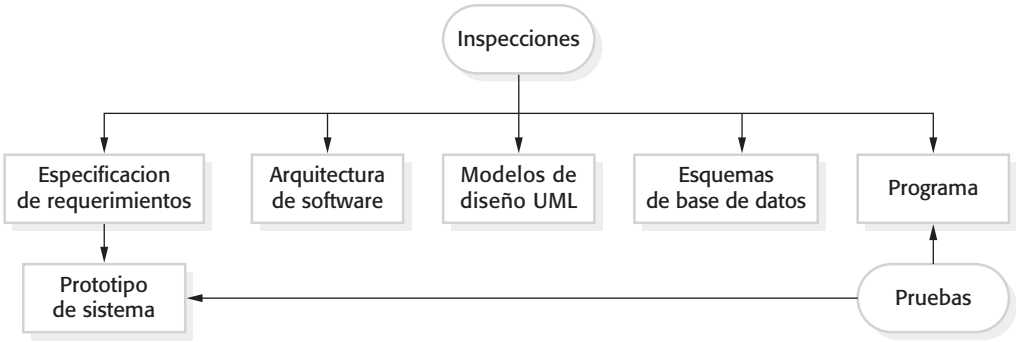


Figura 8.2 Inspecciones y pruebas

podrían soportar fallas, porque los beneficios del uso exceden los costos de la recuperación de fallas. Ante tales situaciones, no es necesario dedicar mucho tiempo en la puesta a prueba del software. Sin embargo, conforme el software se completa, los usuarios esperan que se torne más confiable, de modo que pueden requerirse pruebas exhaustivas en versiones posteriores.

3. *Entorno de mercado* Cuando un sistema se comercializa, los vendedores del sistema deben considerar los productos competitivos, el precio que los clientes están dispuestos a pagar por un sistema y la fecha requerida para entregar dicho sistema. En un ambiente competitivo, una compañía de software puede decidir lanzar al mercado un programa antes de estar plenamente probado y depurado, pues quiere que sus productos sean los primeros en ubicarse. Si un producto de software es muy económico, los usuarios tal vez toleren un nivel menor de fiabilidad.

Al igual que las pruebas de software, el proceso de verificación y validación implicaría inspecciones y revisiones de software. Estas últimas analizan y comprueban los requerimientos del sistema, los modelos de diseño, el código fuente del programa, e incluso las pruebas propuestas para el sistema. Éstas son las llamadas técnicas V&V "estáticas" donde no es necesario ejecutar el software para verificarlo. La figura 8.2 indica que las inspecciones y las pruebas del software soportan V&V en diferentes etapas del proceso del software. Las flechas señalan las etapas del proceso en que pueden usarse las técnicas.

Las inspecciones se enfocan principalmente en el código fuente de un sistema, aun cuando cualquier representación legible del software, como sus requerimientos o modelo de diseño, logre inspeccionarse. Cuando un sistema se inspecciona, se utiliza el conocimiento del sistema, su dominio de aplicación y el lenguaje de programación o modelado para descubrir errores.

Hay tres ventajas en la inspección del software sobre las pruebas:

1. Durante las pruebas, los errores pueden enmascarar (ocultar) otras fallas. Cuando un error lleva a salidas inesperadas, nunca se podrá asegurar si las anomalías de salida posteriores se deben a un nuevo error o son efectos colaterales del error original. Puesto que la inspección es un proceso estático, no hay que preocuparse por las interacciones entre errores. En consecuencia, una sola sesión de inspección descubriría muchos errores en un sistema.

Planeación de pruebas

La planeación de pruebas se interesa por la fecha y los recursos de todas las actividades durante el proceso de pruebas. Incluye la definición del proceso de pruebas, al tomar en cuenta tanto al personal como el tiempo disponible. Por lo general, se creará un plan de prueba que define lo que debe probarse, la fecha establecida de pruebas y cómo se registrarán éstas. Para sistemas críticos, el plan de prueba también puede incluir detalles de las pruebas que se van a correr en el software.

http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html

2. Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales. Si un programa está incompleto, entonces es necesario desarrollar equipos de prueba especializados para poner a prueba las partes disponibles. Evidentemente, esto genera costos para el desarrollo del sistema.

3. Además de buscar defectos de programa, una inspección puede considerar también atributos más amplios de calidad de un programa, como el cumplimiento con estándares, la portabilidad y la mantenibilidad. Pueden buscarse ineficiencias, algoritmos inadecuados y estilos de programación imitados que hagan al sistema difícil de mantener y actualizar.

Las inspecciones de programa son una idea antigua y la mayoría de estudios y experimentos indican que las inspecciones son más efectivas para el descubrimiento de defectos, que para las pruebas del programa. Fagan (1986) reportó que más del 60% de los errores en un programa se detectan mediante inspecciones informales de programa. En el proceso de Cleanroom (cuarto limpio) (Prowell *et al*., 1999) se afirma que más del 90% de los defectos pueden detectarse en inspecciones del programa.

Sin embargo, las inspecciones no sustituyen las pruebas del software, ya que no son eficaces para descubrir defectos que surjan por interacciones inesperadas entre diferentes partes de un programa, problemas de temporización o dificultades con el rendimiento del sistema. Más aún, en compañías o grupos de desarrollo pequeños, suele ser especialmente difícil y costoso reunir a un equipo de inspección separado, ya que todos los miembros potenciales del equipo también podrían ser desarrolladores de software. En el capítulo 24 (“Gestión de la calidad”) se estudian las revisiones e inspecciones con más detenimiento. En el capítulo 15 se explica el análisis estático automatizado, en el cual el texto fuente de un programa se analiza automáticamente para descubrir anomalías. Este capítulo se enfoca en las pruebas y los procesos de pruebas.

La figura 8.3 presenta un modelo abstracto del proceso de prueba “tradicional”, como se utiliza en el desarrollo dirigido por un plan. Los casos de prueba son especificaciones de las entradas a la prueba y la salida esperada del sistema (los resultados de la prueba), además de información sobre lo que se pone a prueba. Los datos de prueba son las entradas que se diseñaron para probar un sistema. En ocasiones pueden generarse automáticamente datos de prueba; no obstante, es imposible la generación automática de casos de prueba, pues debe estar implicada gente que entienda lo que se supone que tiene que hacer el sistema para especificar los resultados de prueba previstos. Sin embargo, es posible automatizar la ejecución de pruebas. Los resultados previstos se comparan automáticamente con los resultados establecidos, de manera que no haya necesidad de que un individuo busque errores y anomalías al correr las pruebas.

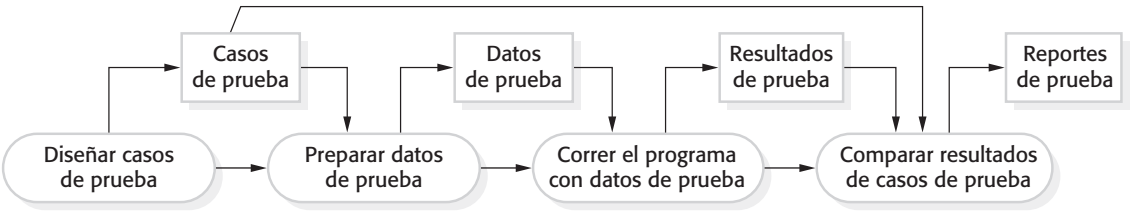


Figura 8.3 Modelo del proceso de pruebas de software

Por lo general, un sistema de software comercial debe pasar por tres etapas de pruebas:

1. Pruebas de desarrollo, donde el sistema se pone a prueba durante el proceso para descubrir errores (*bugs*) y defectos. Es probable que en el desarrollo de prueba intervengan diseñadores y programadores del sistema.
2. Versiones de prueba, donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios. La meta de la prueba de versión es comprobar que el sistema cumpla con los requerimientos de los participantes del sistema.
3. Pruebas de usuario, donde los usuarios reales o potenciales de un sistema prueban el sistema en su propio entorno. Para productos de software, el "usuario" puede ser un grupo interno de marketing, que decide si el software se comercializa, se lanza y se vende. Las pruebas de aceptación se efectúan cuando el cliente prueba de manera formal un sistema para decidir si debe aceptarse del proveedor del sistema, o si se requiere más desarrollo.

En la práctica, el proceso de prueba por lo general requiere una combinación de pruebas manuales y automatizadas. En las primeras pruebas manuales, un examinador opera el programa con algunos datos de prueba y compara los resultados con sus expectativas. Anota y reporta las discrepancias con los desarrolladores del programa. En las pruebas automatizadas, éstas se codifican en un programa que opera cada vez que se prueba el sistema en desarrollo. Comúnmente esto es más rápido que las pruebas manuales, sobre todo cuando incluye pruebas de regresión, es decir, aquellas que implican volver a correr pruebas anteriores para comprobar que los cambios al programa no introdujeron nuevos bugs.

El uso de pruebas automatizadas aumentó de manera considerable durante los últimos años. Sin embargo, las pruebas nunca pueden ser automatizadas por completo, ya que esta clase de pruebas sólo comprueban que un programa haga lo que supone que tiene que hacer. Es prácticamente imposible usar pruebas automatizadas para probar sistemas que dependan de cómo se ven las cosas (por ejemplo, una interfaz gráfica de usuario) o probar que un programa no presenta efectos colaterales indeseados.

8.1 Pruebas de desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que realiza el equipo que elabora el sistema. El examinador del software suele ser el programador que diseñó dicho software, aunque éste no es siempre el caso. Algunos procesos de desarrollo usan parejas de programador/examinador (Cusamano y Selby, 1998) donde cada programador

Depuración

Depuración (*debugging*) es el proceso para corregir los errores y problemas descubiertos por las pruebas. Al usar información de las pruebas del programa, los depuradores, para localizar y reparar el error del programa, emplean tanto su conocimiento del lenguaje de programación como el resultado que se espera de la prueba. Este proceso recibe con frecuencia apoyo de herramientas de depuración interactivas que brindan información adicional sobre la ejecución del programa.

http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html

tiene un examinador asociado que desarrolla pruebas y auxilia con el proceso de pruebas. Para sistemas críticos, puede usarse un proceso más formal, con un grupo de prueba independiente dentro del equipo de desarrollo. Son responsables del desarrollo de pruebas y del mantenimiento de registros detallados de los resultados de las pruebas.

Durante el desarrollo, las pruebas se realizan en tres niveles de granulación:

1. Pruebas de unidad, donde se ponen a prueba unidades de programa o clases de objetos individuales. Las pruebas de unidad deben enfocarse en comprobar la funcionalidad de objetos o métodos.
2. Pruebas del componente, donde muchas unidades individuales se integran para crear componentes compuestos. La prueba de componentes debe enfocarse en probar interfaces del componente.
3. Pruebas del sistema, donde algunos o todos los componentes en un sistema se integran y el sistema se prueba como un todo. Las pruebas del sistema deben enfocarse en poner a prueba las interacciones de los componentes.

Las pruebas de desarrollo son, ante todo, un proceso de prueba de defecto, en las cuales la meta consiste en descubrir bugs en el software. Por lo tanto, a menudo están entrelazadas con la depuración: el proceso de localizar problemas con el código y cambiar el programa para corregirlos.

8.1.1 Pruebas de unidad

Las pruebas de unidad son el proceso de probar componentes del programa, como métodos o clases de objetos. Las funciones o los métodos individuales son el tipo más simple de componente. Las pruebas deben llamarse para dichas rutinas con diferentes parámetros de entrada. Usted puede usar los enfoques para el diseño de casos de prueba que se estudian en la sección 8.1.2, con la finalidad de elaborar las pruebas de función o de método.

Cuando pone a prueba las clases de objetos, tiene que diseñar las pruebas para brindar cobertura a todas las características del objeto. Esto significa que debe:

- probar todas las operaciones asociadas con el objeto;
- establecer y verificar el valor de todos los atributos relacionados con el objeto;
- poner el objeto en todos los estados posibles. Esto quiere decir que tiene que simular todos los eventos que causen un cambio de estado.

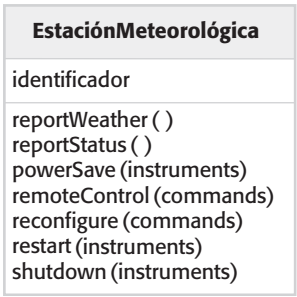


Figura 8.4 Interfaz de objeto de estación meteorológica

Considere, por ejemplo, el objeto de estación meteorológica del modelo analizado en el capítulo 7. La interfaz de este objeto se muestra en la figura 8.4. Tiene un solo atributo, que es su identificador (*identifier*). Ésta es una constante que se establece cuando se instala la estación meteorológica. Por consiguiente, sólo se necesita una prueba que demuestre si se configuró de manera adecuada. Usted necesita definir casos de prueba para todos los métodos asociados con el objeto, como reportWeather, reportStatus, etcétera. Aunque lo ideal es poner a prueba los métodos en aislamiento, en algunos casos son precisas ciertas secuencias de prueba. Por ejemplo, para someter a prueba el método que desactiva los instrumentos de la estación meteorológica (*shutdown*), se necesita ejecutar el método *restart* (reinicio).

La generalización o herencia provoca que sea más complicada la prueba de las clases de objetos. Usted no debe poner únicamente a prueba una operación en la clase donde se definió, ni suponer que funcionará como se esperaba en las subclases que heredan la operación. La operación que se hereda puede hacer conjeturas sobre otras operaciones y atributos. Es posible que no sean válidas en algunas subclases que hereden la operación. Por consiguiente, tiene que poner a prueba la operación heredada en todos los contextos en que se utilice.

Para probar los estados de la estación meteorológica, se usa un modelo de estado, tal como el que se muestra en la figura 7.8 del capítulo anterior. Al usar este modelo, identificará secuencias de transiciones de estado que deban probarse y definirá secuencias de eventos para forzar dichas transiciones. En principio, hay que probar cualquier secuencia posible de transición de estado, aunque en la práctica ello resulte muy costoso. Los ejemplos de secuencias de estado que deben probarse en la estación meteorológica incluyen:

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Siempre que sea posible, se deben automatizar las pruebas de unidad. En estas pruebas de unidad automatizadas, podría usarse un marco de automatización de pruebas (como JUnit) para escribir y correr sus pruebas de programa. Los marcos de pruebas de unidad ofrecen clases de pruebas genéricas que se extienden para crear casos de prueba específicos. En tal caso, usted podrá correr todas las pruebas que implementó y reportar, con frecuencia mediante alguna GUI, el éxito o el fracaso de las pruebas. Es común que toda una serie de pruebas completa opere en algunos segundos, de modo que es posible ejecutar todas las pruebas cada vez que efectúe un cambio al programa.

Un conjunto automatizado de pruebas tiene tres partes:

1. Una parte de configuración, en la cual se inicializa el sistema con el caso de prueba, esto es, las entradas y salidas esperadas.

2. Una parte de llamada (*call*), en la cual se llama al objeto o al método que se va a probar.
3. Una parte de declaración, en la cual se compara el resultado de la llamada con el resultado esperado. Si la información se evalúa como verdadera, la prueba tuvo éxito; pero si resulta falsa, entonces fracasó.

En ocasiones, el objeto que se prueba tiene dependencias de otros objetos que tal vez no se escribieron o que, si se utilizan, frenan el proceso de pruebas. Si su objeto llama a una base de datos, por ejemplo, esto requeriría un proceso de configuración lento antes de usarse. En tales casos, usted puede decidir usar objetos *mock* (simulados). Éstos son objetos con la misma interfaz como los usados por objetos externos que simulan su funcionalidad. Por ende, un objeto mock que aparenta una base de datos suele tener sólo algunos ítems de datos que se organizan en un arreglo. Por lo tanto, puede entrar rápidamente a ellos, sin las sobrecargas de llamar a una base de datos y acceder a discos. De igual modo, los objetos mock pueden usarse para simular una operación anormal o eventos extraños. Por ejemplo, si se pretende que el sistema tome acción en ciertas horas del día, su objeto mock simplemente regresará estas horas, independientemente de la hora real en el reloj.

8.1.2 Elección de casos de pruebas de unidad

Las pruebas son costosas y consumen tiempo, así que es importante elegir casos efectivos de pruebas de unidad. La efectividad significa, en este caso, dos cuestiones:

1. Los casos de prueba tienen que mostrar que, cuando se usan como se esperaba, el componente que se somete a prueba hace lo que se supone que debe hacer.
2. Si hay defectos en el componente, éstos deberían revelarse mediante los casos de prueba.

En consecuencia, hay que escribir dos tipos de casos de prueba. El primero debe reflejar una operación normal de un programa y mostrar que el componente funciona. Por ejemplo, si usted va a probar un componente que crea e inicia el registro de un nuevo paciente, entonces, su caso de prueba debe mostrar que el registro existe en una base de datos, y que sus campos se configuraron como se especificó. El otro tipo de caso de prueba tiene que basarse en probar la experiencia de donde surgen problemas comunes. Debe usar entradas anormales para comprobar que se procesan de manera adecuada sin colapsar el componente.

Aquí se discuten dos estrategias posibles que serían efectivas para ayudarle a elegir casos de prueba. Se trata de:

1. Prueba de partición, donde se identifican grupos de entradas con características comunes y se procesan de la misma forma. Debe elegir las pruebas dentro de cada uno de dichos grupos.
2. Pruebas basadas en lineamientos, donde se usan lineamientos para elegir los casos de prueba. Dichos lineamientos reflejan la experiencia previa de los tipos de errores que suelen cometer los programadores al desarrollar componentes.

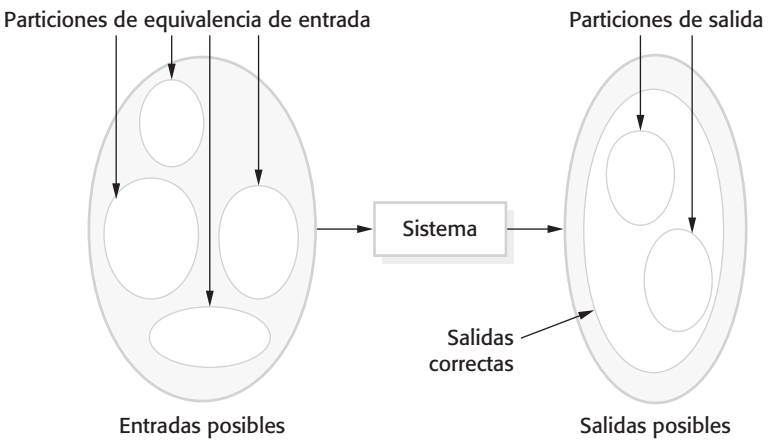


Figura 8.5 Partición de equivalencia

Los datos de entrada y los resultados de salida de un programa regularmente caen en un número de clases diferentes con características comunes. Los ejemplos de estas clases son números positivos, números negativos y selecciones de menú. Por lo general, los programas se comportan en una forma comparable a todos los miembros de una clase. Esto es, si usted prueba un programa que hace un cálculo y requiere dos números positivos, entonces esperaría que el programa se comportara de igual modo en todos los números positivos.

Debido a este comportamiento equivalente, dichas clases se llaman en ocasiones particiones de equivalencia o dominios (Bezier, 1990). Para el diseño de casos de prueba, un enfoque sistemático se basa en identificar todas las particiones de entrada y salida para un sistema o unos componentes. Los casos de prueba se elaboran de forma que las entradas o salidas se encuentren dentro de dichas particiones. La prueba de partición sirve para diseñar casos de prueba tanto para sistemas como para componentes.

En la figura 8.5, la elipse sombreada más grande ubicada en el lado izquierdo representa el conjunto de todas las entradas posibles al programa que se someterá a prueba. Las elipses más pequeñas sin sombrear constituyen particiones de equivalencia. Un programa que se ponga a prueba debe procesar de la misma forma todos los miembros de las particiones de equivalencia de entrada. Las particiones de equivalencia de salida son particiones dentro de las cuales todas las salidas tienen algo en común. En ocasiones hay un mapeo 1:1 entre particiones de equivalencia de entrada y salida. Sin embargo, éste no siempre es el caso; quizás usted necesite definir una partición de equivalencia de entrada independiente, donde la única característica común de las entradas sea que generan salidas dentro de la misma partición de salida. El área sombreada en la elipse izquierda representa excepciones que pueden ocurrir (es decir, respuestas a entradas inválidas).

Una vez identificado el conjunto de particiones, los casos de prueba se eligen de cada una de dichas particiones. Una buena regla empírica para la selección de casos de prueba es seleccionar casos de prueba en las fronteras de las particiones, además de casos cerca del punto medio de la partición. La razón es que diseñadores y programadores tienden a considerar valores típicos de entradas cuando se desarrolla un sistema. Éstos se prueban al elegir el punto medio de la partición. Los valores frontera son usualmente atípicos (por ejemplo, cero puede comportarse de manera diferente a otros números no negativos), de modo que a veces los desarrolladores los pasan por alto. Con frecuencia ocurren fallas del programa cuando se procesan estos valores atípicos.

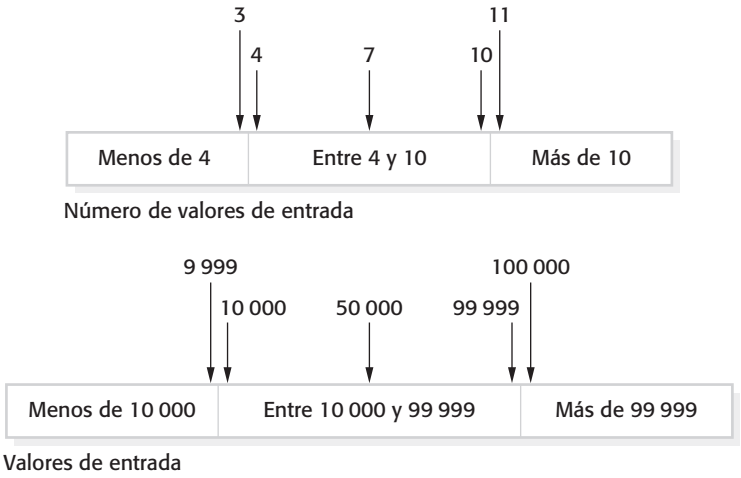


Figura 8.6 Particiones de equivalencia

Las particiones se identifican mediante la especificación del programa o la documentación del usuario y a partir de la experiencia, de donde se predicen las clases de valor de entrada que es probable que detecten errores. Por ejemplo, digamos que la especificación de un programa establece que el programa acepta de 4 a 8 entradas que son cinco dígitos enteros mayores que 10 000. Usted usa esta información para identificar las particiones de entrada y los posibles valores de entrada de prueba. Esto se muestra en la figura 8.6.

Cuando se usa la especificación de un sistema para reconocer particiones de equivalencia, se llama "pruebas de caja negra". Aquí no es necesario algún conocimiento de cómo funciona el sistema. Sin embargo, puede ser útil complementar las pruebas de caja negra con "pruebas de caja blanca", en las cuales se busca el código del programa para encontrar otras posibles pruebas. Por ejemplo, su código puede incluir excepciones para manejar las entradas incorrectas. Este conocimiento se utiliza para identificar "particiones de excepción": diferentes rangos donde deba aplicarse el mismo manejo de excepción.

La partición de equivalencia es un enfoque efectivo para las pruebas, porque ayuda a explicar los errores que cometen con frecuencia los programadores al procesar entradas en los bordes de las particiones. Usted también puede usar lineamientos de prueba para ayudarse a elegir casos de prueba. Los lineamientos encapsulan conocimiento sobre qué tipos de casos de prueba son efectivos para la detección de errores. Por ejemplo, cuando se prueban programas con secuencias, arreglos o listas, los lineamientos que pueden ayudar a revelar defectos incluyen:

1. Probar software con secuencias que tengan sólo un valor único. Los programadores naturalmente consideran a las secuencias como compuestas por muchos valores y, en ocasiones, incrustan esta suposición en sus programas. En consecuencia, si se presenta una secuencia de un valor único, es posible que un programa no funcione de manera adecuada.
2. Usar diferentes secuencias de diversos tamaños en distintas pruebas. Esto disminuye las oportunidades de que un programa con defectos genere accidentalmente una salida correcta, debido a algunas características accidentales de la entrada.
3. Derivar pruebas de modo que se acceda a los elementos primero, medio y último de la secuencia. Este enfoque revela problemas en las fronteras de la partición.

Pruebas de trayectoria

Las pruebas de trayectoria son una estrategia de prueba que se dirige principalmente a ejercitar cada trayectoria de ejecución independiente a través de un componente o programa. Si se ejecuta cualquier trayectoria independiente, entonces deben ejecutarse todos los enunciados en el componente al menos una vez. Todos los enunciados condicionales se prueban para los casos verdadero y falso. En un proceso de desarrollo orientado a objetos, la prueba de trayectoria puede usarse cuando se prueban los métodos asociados con los objetos.

http://www.SoftwareEngineering-9.com/Web/Testing/PathTest.html

El libro de Whittaker (2002) incluye muchos ejemplos de lineamientos que se pueden utilizar en el diseño de casos de prueba. Algunos de los lineamientos más generales que sugiere son:

- Elegir entradas que fuercen al sistema a generar todos los mensajes de error;
- Diseñar entradas que produzcan que los buffers de entrada se desborden;
- Repetir varias veces la misma entrada o serie de entradas;
- Forzar la generación de salidas inválidas;
- Forzar resultados de cálculo demasiado largos o demasiado pequeños.

Conforme adquiera experiencia con las pruebas, usted podrá desarrollar sus propios lineamientos sobre cómo elegir casos de prueba efectivos. En la siguiente sección de este capítulo se incluyen más ejemplos de lineamientos de prueba.

8.1.3 Pruebas de componentes

En general, los componentes de software son componentes compuestos constituidos por varios objetos en interacción. Por ejemplo, en el sistema de la estación meteorológica, el componente de reconfiguración incluye objetos que tratan con cada aspecto de la reconfiguración. El acceso a la funcionalidad de dichos objetos es a través de la interfaz de componente definida. Por consiguiente, la prueba de componentes compuestos tiene que enfocarse en mostrar que la interfaz de componente se comporta según su especificación. Usted puede suponer que dentro del componente se completaron las pruebas de unidad sobre el objeto individual.

La figura 8.7 ilustra la idea de la prueba de interfaz de componente. Suponga que los componentes A, B y C se integraron para crear un componente o subsistema más grande. Los casos de prueba no se aplican a los componentes individuales, sino más bien a la interfaz del componente compuesto, creado al combinar tales componentes. Los errores de interfaz en el componente compuesto quizá no se detecten al poner a prueba los objetos individuales, porque dichos errores resultan de interacciones entre los objetos en el componente.

Existen diferentes tipos de interfaz entre componentes de programa y, en consecuencia, distintos tipos de error de interfaz que llegan a ocurrir:

1. *Interfaces de parámetro* Son interfaces en que los datos, o en ocasiones referencias de función, pasan de un componente a otro. Los métodos en un objeto tienen una interfaz de parámetro.

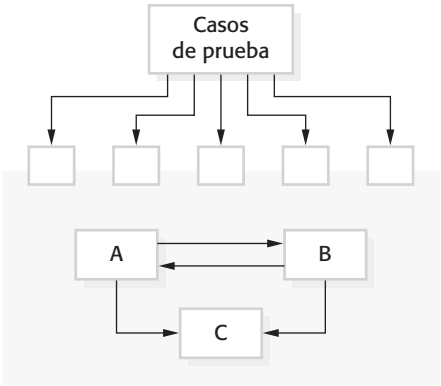


Figura 8.7 Prueba de interfaz

2. *Interfaces de memoria compartida* Son interfaces en que un bloque de memoria se reparte entre componentes. Los datos se colocan en la memoria de un subsistema y otros subsistemas los recuperan de ahí. Este tipo de interfaz se usa con frecuencia en sistemas embebidos, donde los sensores crean datos que se recuperan y son procesados por otros componentes del sistema.
3. *Interfaces de procedimiento* Son interfaces en que un componente encapsula un conjunto de procedimientos que pueden ser llamados por otros componentes. Los objetos y otros componentes reutilizables tienen esta forma de interfaz.
4. *Interfaces que pasan mensajes* Se trata de interfaces donde, al enviar un mensaje, un componente solicita un servicio de otro componente. El mensaje de retorno incluye los resultados para ejecutar el servicio. Algunos sistemas orientados a objetos tienen esta forma de interfaz, así como los sistemas cliente-servidor.

Los errores de interfaz son una de las formas más comunes de falla en los sistemas complejos (Lutz, 1993). Dichos errores caen en tres clases:

- *Uso incorrecto de interfaz* Un componente que llama a otro componente y comete algún error en el uso de su interfaz. Este tipo de error es común con interfaces de parámetro, donde los parámetros pueden ser del tipo equivocado, o bien, pasar en el orden o el número equivocados de parámetros.
- *Mala interpretación de interfaz* Un componente que malinterpreta la especificación de la interfaz del componente llamado y hace suposiciones sobre su comportamiento. El componente llamado no se comporta como se esperaba, lo cual entonces genera un comportamiento imprevisto en el componente que llama. Por ejemplo, un método de búsqueda binaria puede llamarse con un parámetro que es un arreglo desordenado. Entonces fallaría la búsqueda.
- *Errores de temporización* Ocurren en sistemas de tiempo real que usan una memoria compartida o una interfaz que pasa mensajes. El productor de datos y el consumidor de datos operan a diferentes niveles de rapidez. A menos que se tenga cuidado particular en el diseño de interfaz, el consumidor puede acceder a información obsoleta,

porque el productor de la información no actualizó la información de la interfaz compartida.

Las pruebas por defectos de interfaz son difíciles porque algunas fallas de interfaz sólo pueden manifestarse ante condiciones inusuales. Por ejemplo, se dice que un objeto implementa una cola como una estructura de datos de longitud fija. Un objeto que llama puede suponer que la cola se implementó como una estructura de datos infinita y no verificaría el desbordamiento de la cola, cuando se ingresa un ítem. Esta condición sólo se logra detectar durante las pruebas, al diseñar casos de prueba que fuercen el desbordamiento de la cola, y causen que el desbordamiento corrompa el comportamiento del objeto en cierta forma detectable.

Un problema posterior podría surgir derivado de interacciones entre fallas en diferentes módulos u objetos. Las fallas en un objeto sólo se detectan cuando algún otro objeto se comporta de una forma inesperada. Por ejemplo, un objeto llama a otro objeto para recibir algún servicio y supone que es correcta la respuesta; si el servicio de llamada es deficiente en algún modo, el valor devuelto puede ser válido pero equivocado. Esto no se detecta de inmediato, sino sólo se vuelve evidente cuando algún cálculo posterior sale mal.

Algunos lineamientos generales para las pruebas de interfaz son:

1. Examinar el código que se va a probar y listar explícitamente cada llamado a un componente externo. Diseñe un conjunto de pruebas donde los valores de los parámetros hacia los componentes externos estén en los extremos finales de sus rangos. Dichos valores extremos tienen más probabilidad de revelar inconsistencias de interfaz.
2. Donde los punteros pasen a través de una interfaz, pruebe siempre la interfaz con parámetros de puntero nulo.
3. Donde un componente se llame a través de una interfaz de procedimiento, diseñe pruebas que deliberadamente hagan que falle el componente. Diferir las suposiciones de falla es una de las interpretaciones de especificación equivocadas más comunes.
4. Use pruebas de esfuerzo en los sistemas que pasan mensajes. Esto significa que debe diseñar pruebas que generen muchos más mensajes de los que probablemente ocurran en la práctica. Ésta es una forma efectiva de revelar problemas de temporización.
5. Donde algunos componentes interactúen a través de memoria compartida, diseñe pruebas que varíen el orden en que se activan estos componentes. Tales pruebas pueden revelar suposiciones implícitas hechas por el programador, sobre el orden en que se producen y consumen los datos compartidos.

En ocasiones, las inspecciones y revisiones suelen ser más efectivas en costo que las pruebas para descubrir errores de interfaz. Las inspecciones pueden concentrarse en interfaces de componente e interrogantes sobre el comportamiento supuesto de la interfaz planteada durante el proceso de inspección. Un lenguaje robusto como Java permite que muchos errores de interfaz sean descubiertos por el compilador. Los analizadores estáticos (capítulo 15) son capaces de detectar un amplio rango de errores de interfaz.

Integración y pruebas incrementales

Las pruebas de sistema implican integrar diferentes componentes y, después, probar el sistema integrado que se creó. Siempre hay que usar un enfoque incremental para la integración y las pruebas (es decir, se debe incluir un componente, probar el sistema, integrar otro componente, probar de nuevo y así sucesivamente). Esto significa que, si ocurren problemas, quizá se deban a interacciones con el componente que se integró más recientemente.

La integración y las pruebas incrementales son fundamentales para los métodos ágiles como XP, donde las pruebas de regresión (véase sección 8.2) se efectúan cada vez que se integra un nuevo incremento.

http://www.SoftwareEngineering-9.com/Web/Testing/Integration.html

8.1.4 Pruebas del sistema

Las pruebas del sistema durante el desarrollo incluyen la integración de componentes para crear una versión del sistema y, luego, poner a prueba el sistema integrado. Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces. Evidentemente, se traslapan con las pruebas de componentes, pero existen dos importantes diferencias:

1. Durante las pruebas de sistema, los componentes reutilizables desarrollados por separado y los sistemas comerciales pueden integrarse con componentes desarrollados recientemente. Entonces se prueba el sistema completo.
2. Los componentes desarrollados por diferentes miembros del equipo o de grupos pueden integrarse en esta etapa. La prueba de sistema es un proceso colectivo más que individual. En algunas compañías, las pruebas del sistema implican un equipo de prueba independiente, sin la inclusión de diseñadores ni de programadores.

Cuando se integran componentes para crear un sistema, se obtiene un comportamiento emergente. Esto significa que algunos elementos de funcionalidad del sistema sólo se hacen evidentes cuando se reúnen los componentes. Éste podría ser un comportamiento emergente planeado que debe probarse. Por ejemplo, usted puede integrar un componente de autenticación con un componente que actualice información. De esta manera, tiene una característica de sistema que restringe la información actualizada de usuarios autorizados. Sin embargo, algunas veces, el comportamiento emergente no está planeado ni se desea. Hay que desarrollar pruebas que demuestren que el sistema sólo hace lo que se supone que debe hacer.

Por lo tanto, las pruebas del sistema deben enfocarse en poner a prueba las interacciones entre los componentes y los objetos que constituyen el sistema. También se prueban componentes o sistemas reutilizables para acreditar que al integrarse nuevos componentes funcionen como se esperaba. Esta prueba de interacción debe descubrir aquellos bugs de componente que sólo se revelan cuando lo usan otros componentes en el sistema. Las pruebas de interacción también ayudan a encontrar interpretaciones erróneas, cometidas por desarrolladores de componentes, acerca de otros componentes en el sistema.

Por su enfoque en las interacciones, las pruebas basadas en casos son un enfoque efectivo para la prueba del sistema. Normalmente, cada caso de uso es implementado por

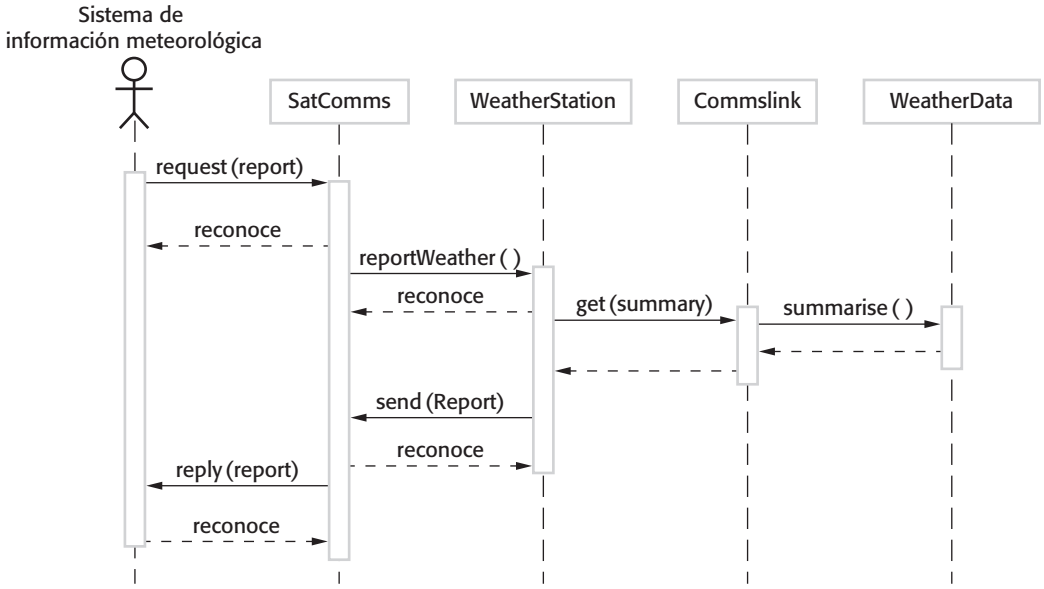


Figura 8.8 Gráfico de secuencia de recolección de datos meteorológicos

varios componentes u objetos en el sistema. Probar los casos de uso obliga a que ocurran estas interacciones. Si usted desarrolló un diagrama de secuencia para modelar la implementación de casos de uso, verá los objetos o componentes implicados en la interacción.

Para ilustrar lo anterior, se usa un ejemplo del sistema de estación meteorológica a campo abierto, donde se pide a la estación meteorológica reportar un resumen de datos a una computadora remota. El caso de uso para esto se describe en la figura 7.3 (capítulo anterior). La figura 8.8 (copia de la figura 7.7) muestra la secuencia de operaciones en la estación meteorológica, al responder a una petición de recolección de datos para el sistema de mapeo. Este diagrama sirve para identificar operaciones que se probarán y para ayudar a diseñar los casos de prueba para efectuar las pruebas. Por consiguiente, emitir una petición para un reporte dará como resultado la ejecución de la siguiente cadena de métodos:

SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary)
→ WeatherData:summarize

El diagrama de secuencia ayuda a diseñar los casos de prueba específicos necesarios, pues muestra cuáles entradas se requieren y cuáles salidas se crean:

1. Una entrada de una petición para un reporte tiene que contar con reconocimiento asociado. En última instancia, a partir de la petición debe regresarse un reporte. Durante las pruebas, se debe crear un resumen de datos que sirva para comprobar que el reporte se organiza correctamente.
2. Una petición de entrada para un reporte a WeatherStation da como resultado la generación de un reporte resumido. Usted puede probar esto en aislamiento, creando datos brutos correspondientes al resumen que preparó para la prueba de SatComms, y demostrar que el objeto WeatherStation produce este resumen. Tales datos brutos se usan también para probar el objeto WeatherData.

Desde luego, en la figura 8.8 se simplificó el diagrama de secuencia para que no muestre excepciones. Asimismo, una prueba completa de caso/escenario de uso considera esto y garantiza que los objetos manejen adecuadamente las excepciones.

Para la mayoría de sistemas es difícil saber cuántas pruebas de sistemas son esenciales y cuándo hay que dejar de hacer pruebas. Las pruebas exhaustivas, donde se pone a prueba cada secuencia posible de ejecución del programa, son imposibles. Por lo tanto, las pruebas deben basarse en un subconjunto de probables casos de prueba. De manera ideal, para elegir este subconjunto, las compañías de software cuentan con políticas, las cuales pueden basarse en políticas de prueba generales, como una política de que todos los enunciados del programa se ejecuten al menos una vez. Como alternativa, pueden basarse en la experiencia de uso de sistema y, a la vez, enfocarse en probar las características del sistema operativo. Por ejemplo:

1. Tienen que probarse todas las funciones del sistema que se ingresen a través de un menú.
2. Debe experimentarse la combinación de funciones (por ejemplo, formateo de texto) que se ingrese por medio del mismo menú.
3. Donde se proporcione entrada del usuario, hay que probar todas las funciones, ya sea con entrada correcta o incorrecta.

Por experiencia con los principales productos de software, como procesadores de texto u hojas de cálculo, es claro que lineamientos similares se usan por lo general durante la prueba del producto. Usualmente funcionan cuando las características del software se usan en aislamiento. Los problemas se presentan, dice Whittaker (2002), cuando las combinaciones de características de uso menos común no se prueban en conjunto. Él da el ejemplo de cómo, en un procesador de texto de uso común, el uso de notas al pie de página con una plantilla en columnas múltiples causa la distribución incorrecta del texto.

Las pruebas automatizadas del sistema suelen ser más difíciles que las pruebas automatizadas de unidad o componente. Las pruebas automatizadas de unidad se apoyan en la predicción de salidas y, luego, en la codificación de dichas predicciones en un programa. En tal caso, se compara el pronóstico con el resultado. Sin embargo, el punto de aplicar un sistema puede ser generar salidas que sean grandes o no logren predecirse con facilidad. Se tiene que examinar una salida y demostrar su credibilidad sin crearla necesariamente por adelantado.

8.2 Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (TDD, por las siglas de *Test-Driven Development*) es un enfoque al diseño de programas donde se entrelazan el desarrollo de pruebas y el de código (Beck, 2002; Jeffries y Melnik, 2007). En esencia, el código se desarrolla incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba. El desarrollo dirigido por pruebas se introdujo como parte de los métodos ágiles como la programación extrema. No obstante, se puede usar también en los procesos de desarrollo basados en un plan.

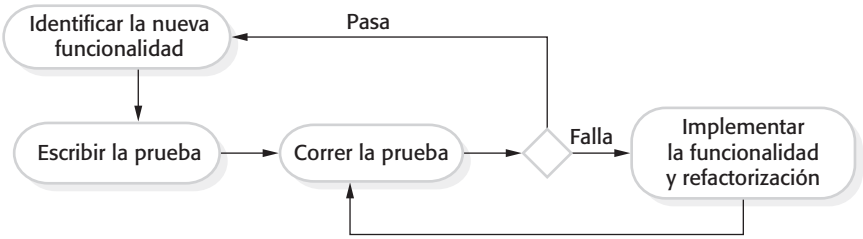


Figura 8.9 Desarrollo dirigido por pruebas

En la figura 8.9 se ilustra el proceso TDD fundamental. Los pasos en el proceso son los siguientes:

1. Se comienza por identificar el incremento de funcionalidad requerido. Éste usualmente debe ser pequeño y aplicable en pocas líneas del código.
2. Se escribe una prueba para esta funcionalidad y se implementa como una prueba automatizada. Esto significa que la prueba puede ejecutarse y reportarse, sin importar si aprueba o falla.
3. Luego se corre la prueba, junto con todas las otras pruebas que se implementaron. Inicialmente, no se aplica la funcionalidad, de modo que la nueva prueba fallará. Esto es deliberado, pues muestra que la prueba añade algo al conjunto de pruebas.
4. Luego se implementa la funcionalidad y se opera nuevamente la prueba. Esto puede incluir la refactorización del código existente, para perfeccionarlo y adicionar nuevo código a lo ya existente.
5. Una vez puestas en funcionamiento con éxito todas las pruebas, se avanza a la implementación de la siguiente funcionalidad.

Un entorno automatizado de pruebas, como el entorno JUnit que soporta pruebas del programa Java (Massol y Husted, 2003), es esencial para TDD. Conforme el código se desarrolla en incrementos muy pequeños, uno tiene la posibilidad de correr cada prueba, cada vez que se adiciona funcionalidad o se refactoriza el programa. Por consiguiente, las pruebas se incrustan en un programa independiente que corre las pruebas y apela al sistema que se prueba. Al usar este enfoque, en unos cuantos segundos se efectúan cientos de pruebas independientes.

Un argumento consistente con el desarrollo dirigido por pruebas es que ayuda a los programadores a aclarar sus ideas acerca de lo que realmente debe hacer un segmento de código. Para escribir una prueba, es preciso entender lo que se quiere, pues esta comprensión facilita la escritura del código requerido. Desde luego, si el conocimiento o la comprensión son incompletos, entonces no ayudará el desarrollo dirigido por pruebas. Por ejemplo, si su cálculo implica división, debería comprobar que no divide los números entre cero. En caso de que olvide escribir una prueba para esto, en el programa nunca se incluirá el código a comprobar.

Además de la mejor comprensión del problema, otros beneficios del desarrollo dirigido por pruebas son:

1. *Cobertura de código* En principio, cualquier segmento de código que escriba debe tener al menos una prueba asociada. Por lo tanto, puede estar seguro de que cual-

quier código en el sistema se ejecuta realmente. El código se prueba a medida que se escribe, de modo que los defectos se descubren con oportunidad en el proceso de desarrollo.

2. *Pruebas de regresión* Un conjunto de pruebas se desarrolla incrementalmente conforme se desarrolla un programa. Siempre es posible correr pruebas de regresión para demostrar que los cambios al programa no introdujeron nuevos bugs.

3. *Depuración simplificada* Cuando falla una prueba, debe ser evidente dónde yace el problema. Es preciso comprobar y modificar el código recién escrito. No se requieren herramientas de depuración para localizar el problema. Los reportes del uso del desarrollo dirigido por pruebas indican que difícilmente alguna vez se necesitará usar un depurador automatizado en el desarrollo dirigido por pruebas (Martin, 2007).

4. *Documentación del sistema* Las pruebas en sí actúan como una forma de documentación que describen lo que debe hacer el código. Leer las pruebas suele facilitar la comprensión del código.

Uno de los beneficios más importantes del desarrollo dirigido por pruebas es que reduce los costos de las pruebas de regresión. Estas últimas implican correr los conjuntos de pruebas ejecutadas exitosamente después de realizar cambios a un sistema. La prueba de regresión verifica que dichos cambios no hayan introducido nuevos bugs en el sistema, y que el nuevo código interactúa como se esperaba con el código existente. Las pruebas de regresión son muy costosas y, por lo general, poco prácticas cuando un sistema se prueba manualmente, pues son muy elevados los costos en tiempo y esfuerzo. Ante tales situaciones, usted debe ensayar y elegir las pruebas más relevantes para volver a correrlas, y es fácil perder pruebas importantes.

Sin embargo, las pruebas automatizadas, que son fundamentales para el desarrollo de primera prueba, reducen drásticamente los costos de las pruebas de regresión. Las pruebas existentes pueden volverse a correr de manera más rápida y menos costosa. Después de realizar cambios a un sistema en el desarrollo de la primera prueba, todas las pruebas existentes deben correr con éxito antes de añadir cualquier funcionalidad accesoria. Como programador, usted podría estar seguro de que la nueva funcionalidad que agregue no causará ni revelará problemas con el código existente.

El desarrollo dirigido por pruebas se usa más en el diseño de software nuevo, donde la funcionalidad se implementa en código nuevo o usa librerías estándar perfectamente probadas. Si se reutilizan grandes componentes en código o sistemas heredados, entonces se necesita escribir pruebas para dichos sistemas como un todo. El desarrollo dirigido por pruebas también puede ser ineficaz con sistemas multihilo. Los diferentes hilos pueden entrelazarse en diferentes momentos y en diversas corridas de pruebas y, por lo tanto, producirán resultados variados.

Si se usa el desarrollo dirigido por pruebas, se necesitará de un proceso de prueba del sistema para validar el sistema; esto es, comprobar que cumple con los requerimientos de todos los participantes del sistema. Las pruebas de sistema también demuestran rendimiento, confiabilidad y evidencian que el sistema no haga aquello que no debe hacer, como producir salidas indeseadas, etcétera. Andrea (2007) sugiere cómo pueden extenderse las herramientas de prueba para integrar algunos aspectos de las pruebas de sistema con TDD.

El desarrollo dirigido por pruebas resulta ser un enfoque exitoso para proyectos de dimensión pequeña y mediana. Por lo general, los programadores que adoptan dicho enfoque están contentos con él y descubren que es una forma más productiva de desarrollar

software (Jeffries y Melnik, 2007). En algunos ensayos, se demostró que conduce a mejorar la calidad del código; en otros, los resultados no son concluyentes. Sin embargo, no hay evidencia de que el TDD conduzca a un código con menor calidad.

8.3 Pruebas de versión

Las pruebas de versión son el proceso de poner a prueba una versión particular de un sistema que se pretende usar fuera del equipo de desarrollo. Por lo general, la versión del sistema es para clientes y usuarios. No obstante, en un proyecto complejo, la versión podría ser para otros equipos que desarrollan sistemas relacionados. Para productos de software, la versión sería para el gerente de producto, quien después la prepara para su venta.

Existen dos distinciones importantes entre las pruebas de versión y las pruebas del sistema durante el proceso de desarrollo:

1. Un equipo independiente que no intervino en el desarrollo del sistema debe ser el responsable de las pruebas de versión.
2. Las pruebas del sistema por parte del equipo de desarrollo deben enfocarse en el descubrimiento de bugs en el sistema (pruebas de defecto). El objetivo de las pruebas de versión es comprobar que el sistema cumpla con los requerimientos y sea suficientemente bueno para uso externo (pruebas de validación).

La principal meta del proceso de pruebas de versión es convencer al proveedor del sistema de que éste es suficientemente apto para su uso. Si es así, puede liberarse como un producto o entregarse al cliente. Por lo tanto, las pruebas de versión deben mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad especificados, y que no falla durante el uso normal. Deben considerarse todos los requerimientos del sistema, no sólo los de los usuarios finales del sistema.

Las pruebas de versión, por lo regular, son un proceso de prueba de caja negra, donde las pruebas se derivan a partir de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede determinarse por el estudio de entradas y salidas relacionadas. Otro nombre para esto es "prueba funcional", llamada así porque al examinador sólo le preocupa la funcionalidad y no la aplicación del software.

8.3.1 Pruebas basadas en requerimientos

Un principio general de buena práctica en la ingeniería de requerimientos es que éstos deben ser comprobables; esto es, los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento. Luego, un examinador comprueba que el requerimiento se cumpla. En consecuencia, las pruebas basadas en requerimientos son un enfoque sistemático al diseño de casos de prueba, donde se considera cada requerimiento y se deriva un conjunto de pruebas para éste. Las pruebas basadas en requerimientos son pruebas de validación más que de defecto: se intenta demostrar que el sistema implementó adecuadamente sus requerimientos.

Por ejemplo, considere los requerimientos relacionados para el MHC-PMS (presentado en el capítulo 1), que se enfocan a la comprobación de alergias a medicamentos:

> *Si se sabe que un paciente es alérgico a algún fármaco en particular, entonces la prescripción de dicho medicamento dará como resultado un mensaje de advertencia que se emitirá al usuario del sistema.*
>
> *Si quien prescribe ignora una advertencia de alergia, deberá proporcionar una razón para ello.*

Para comprobar si estos requerimientos se cumplen, tal vez necesite elaborar muchas pruebas relacionadas:

1. Configurar un registro de un paciente sin alergias conocidas. Prescribir medicamentos para alergias que se sabe que existen. Comprobar que el sistema no emite un mensaje de advertencia.
2. Realizar un registro de un paciente con una alergia conocida. Prescribir el medicamento al que es alérgico y comprobar que el sistema emite la advertencia.
3. Elaborar un registro de un paciente donde se reporten alergias a dos o más medicamentos. Prescribir dichos medicamentos por separado y comprobar que se emite la advertencia correcta para cada medicamento.
4. Prescribir dos medicamentos a los que sea alérgico el paciente. Comprobar que se emiten correctamente dos advertencias.
5. Prescribir un medicamento que emite una advertencia y pasar por alto dicha advertencia. Comprobar que el sistema solicita al usuario proporcionar información que explique por qué pasó por alto la advertencia.

A partir de esto se puede ver que probar un requerimiento no sólo significa escribir una prueba. Por lo general, usted deberá escribir muchas pruebas para garantizar que cubrió los requerimientos. También hay que mantener el rastreo de los registros de sus pruebas basadas en requerimientos, que vinculan las pruebas con los requerimientos específicos que se ponen a prueba.

8.3.2 Pruebas de escenario

Las pruebas de escenario son un enfoque a las pruebas de versión donde se crean escenarios típicos de uso y se les utiliza en el desarrollo de casos de prueba para el sistema. Un escenario es una historia que describe una forma en que puede usarse el sistema. Los escenarios deben ser realistas, y los usuarios reales del sistema tienen que relacionarse con ellos. Si usted empleó escenarios como parte del proceso de ingeniería de requerimientos (descritos en el capítulo 4), entonces podría reutilizarlos como escenarios de prueba.

En un breve ensayo sobre las pruebas de escenario, Kaner (2003) sugiere que una prueba de escenario debe ser una historia narrativa que sea creíble y bastante compleja. Tiene que motivar a los participantes; esto es, deben relacionarse con el escenario y creer que es importante que el sistema pase la prueba. También sugiere que debe ser fácil de

Kate es enfermera con especialidad en atención a la salud mental. Una de sus responsabilidades es visitar a domicilio a los pacientes, para comprobar la efectividad de su tratamiento y que no sufran de efectos colaterales del fármaco.

En un día de visitas domésticas, Kate ingresa al MHC-PMS y lo usa para imprimir su agenda de visitas domiciliarias para ese día, junto con información resumida sobre los pacientes por visitar. Solicita que los registros para dichos pacientes se descarguen a su laptop. Se le pide la palabra clave para cifrar los registros en la laptop.

Uno de los pacientes a quienes visita es Jim, quien es tratado con medicamentos antidepresivos. Jim siente que el medicamento le ayuda, pero considera que el efecto colateral es que se mantiene despierto durante la noche. Kate observa el registro de Jim y se le pide la palabra clave para descifrar el registro. Comprueba el medicamento prescrito y consulta sus efectos colaterales. El insomnio es un efecto colateral conocido, así que anota el problema en el registro de Jim y sugiere que visite la clínica para que cambien el medicamento. Él está de acuerdo, así que Kate ingresa un recordatorio para llamarlo en cuanto ella regrese a la clínica, para concertarle una cita con un médico. Termina la consulta y el sistema vuelve a cifrar el registro de Jim.

Más tarde, al terminar sus consultas, Kate regresa a la clínica y sube los registros de los pacientes visitados a la base de datos. El sistema genera para Kate una lista de aquellos pacientes con quienes debe comunicarse, para obtener información de seguimiento y concertar citas en la clínica.

Figura 8.10 Escenario de uso para el MHC-PMS

evaluar. Si hay problemas con el sistema, entonces el equipo de pruebas de versión tiene que reconocerlos. Como ejemplo de un posible escenario para el MHC-PMS, la figura 8.10 describe una forma de utilizar el sistema en una visita domiciliaria, que pone a prueba algunas características del MHC-PMS:

1. Autenticación al ingresar al sistema.
2. Descarga y carga registros de paciente específicos desde una laptop.
3. Agenda de visitas a domicilio.
4. Cifrado y descifrado de registros de pacientes en un dispositivo móvil.
5. Recuperación y modificación de registros.
6. Vinculación con la base de datos de medicamentos que mantenga información acerca de efectos colaterales.
7. Sistema para recordatorio de llamadas.

Si usted es examinador de versión, opere a través de este escenario, interprete el papel de Kate y observe cómo se comporta el sistema en respuesta a las diferentes entradas. Como "Kate", usted puede cometer errores deliberados, como ingresar la palabra clave equivocada para decodificar registros. Esto comprueba la respuesta del sistema ante los errores. Tiene que anotar cuidadosamente cualquier problema que surja, incluidos problemas de rendimiento. Si un sistema es muy lento, esto cambiará la forma en que se usa. Por ejemplo, si se tarda mucho al cifrar un registro, entonces los usuarios que tengan poco tiempo pueden saltar esta etapa. Si pierden su laptop, una persona no autorizada podría ver entonces los registros de los pacientes.

Cuando se usa un enfoque basado en escenarios, se ponen a prueba por lo general varios requerimientos dentro del mismo escenario. Por lo tanto, además de comprobar

requerimientos individuales, también demuestra que las combinaciones de requerimientos no causan problemas.

8.3.3 Pruebas de rendimiento

Una vez integrado completamente el sistema, es posible probar propiedades emergentes, como el rendimiento y la confiabilidad. Las pruebas de rendimiento deben diseñarse para garantizar que el sistema procese su carga pretendida. Generalmente, esto implica efectuar una serie de pruebas donde se aumenta la carga, hasta que el rendimiento del sistema se vuelve inaceptable.

Como con otros tipos de pruebas, las pruebas de rendimiento se preocupan tanto por demostrar que el sistema cumple con sus requerimientos, como por descubrir problemas y defectos en el sistema. Para probar si los requerimientos de rendimiento se logran, quizá se deba construir un perfil operativo. Un perfil operativo (capítulo 15) es un conjunto de pruebas que reflejan la mezcla real de trabajo que manejará el sistema. Por consiguiente, si el 90% de las transacciones en un sistema son del tipo A, el 5% del tipo B, y el resto de los tipos C, D y E, entonces habrá que diseñar el perfil operativo de modo que la gran mayoría de pruebas sean del tipo A. De otra manera, no se obtendrá una prueba precisa del rendimiento operativo del sistema.

Desde luego, este enfoque no necesariamente es el mejor para pruebas de defecto. La experiencia demuestra que una forma efectiva de descubrir defectos es diseñar pruebas sobre los límites del sistema. En las pruebas de rendimiento, significa estresar el sistema al hacer demandas que estén fuera de los límites de diseño del software. Esto se conoce como "prueba de esfuerzo". Por ejemplo, digamos que usted prueba un sistema de procesamiento de transacciones que se diseña para procesar hasta 300 transacciones por segundo. Comienza por probar el sistema con menos de 300 transacciones por segundo. Luego aumenta gradualmente la carga del sistema más allá de 300 transacciones por segundo, hasta que está muy por arriba de la carga máxima de diseño del sistema y el sistema falla. Este tipo de pruebas tiene dos funciones:

1. Prueba el comportamiento de falla del sistema. Pueden surgir circunstancias a través de una combinación inesperada de eventos donde la carga colocada en el sistema supere la carga máxima anticipada. Ante tales circunstancias, es importante que la falla del sistema no cause corrupción de datos o pérdida inesperada de servicios al usuario. Las pruebas de esfuerzo demuestran que la sobrecarga del sistema hace que "falle poco" en vez de colapsar bajo su carga.

2. Fuerza al sistema y puede hacer que salgan a la luz defectos que no se descubrirían normalmente. Aunque se puede argumentar que esos defectos probablemente no causen fallas en el sistema en uso normal, pudiera haber una serie de combinaciones inusuales de circunstancias normales que requieren pruebas de esfuerzo.

Las pruebas de esfuerzo son particularmente relevantes para los sistemas distribuidos basados en redes de procesadores. Dichos sistemas muestran con frecuencia degradación severa cuando se cargan en exceso. La red se empantana con la coordinación de datos que deben intercambiar los diferentes procesos. Éstos se vuelven cada vez más lentos conforme esperan los datos requeridos de otros procesos. Las pruebas de esfuerzo ayudan a descubrir cuándo comienza la degradación, de manera que se puedan adicionar comprobaciones al sistema para rechazar transacciones más allá de este punto.

8.4 Pruebas de usuario

Las pruebas de usuario o del cliente son una etapa en el proceso de pruebas donde los usuarios o clientes proporcionan entrada y asesoría sobre las pruebas del sistema. Esto puede implicar probar de manera formal un sistema que se comisionó a un proveedor externo, o podría ser un proceso informal donde los usuarios experimentan con un nuevo producto de software, para ver si les gusta y si hace lo que necesitan. Las pruebas de usuario son esenciales, aun cuando se hayan realizado pruebas abarcadoras de sistema y de versión. La razón de esto es que la influencia del entorno de trabajo del usuario tiene un gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema.

Es casi imposible que un desarrollador de sistema replique el entorno de trabajo del sistema, pues las pruebas en el entorno del desarrollador forzosamente son artificiales. Por ejemplo, un sistema que se pretenda usar en un hospital se usa en un entorno clínico donde suceden otros hechos, como emergencias de pacientes, conversaciones con familiares del paciente, etcétera. Todo ello afecta el uso de un sistema, pero los desarrolladores no pueden incluirlos en su entorno de pruebas.

En la práctica, hay tres diferentes tipos de pruebas de usuario:

1. Pruebas alfa, donde los usuarios del software trabajan con el equipo de diseño para probar el software en el sitio del desarrollador.
2. Pruebas beta, donde una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema.
3. Pruebas de aceptación, donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente.

En las pruebas alfa, los usuarios y desarrolladores trabajan en conjunto para probar un sistema a medida que se desarrolla. Esto significa que los usuarios pueden identificar problemas y conflictos que no son fácilmente aparentes para el equipo de prueba de desarrollo. Los desarrolladores en realidad sólo pueden trabajar a partir de los requerimientos, pero con frecuencia esto no refleja otros factores que afectan el uso práctico del software. Por lo tanto, los usuarios brindan información sobre la práctica que ayuda con el diseño de pruebas más realistas.

Las pruebas alfa se usan a menudo cuando se desarrollan productos de software que se venden como sistemas empaquetados. Los usuarios de dichos productos quizás estén satisfechos de intervenir en el proceso de pruebas alfa porque esto les da información oportuna acerca de las características del nuevo sistema que pueden explotar. También reduce el riesgo de que cambios no anticipados al software tengan efectos perturbadores para su negocio. Sin embargo, las pruebas alfa también se utilizan cuando se desarrolla software personalizado. Los métodos ágiles, como XP, abogan por la inclusión del usuario en el proceso de desarrollo y que los usuarios tengan un papel activo en el diseño de pruebas para el sistema.

Las pruebas beta tienen lugar cuando una versión temprana de un sistema de software, en ocasiones sin terminar, se pone a disposición de clientes y usuarios para evaluación.

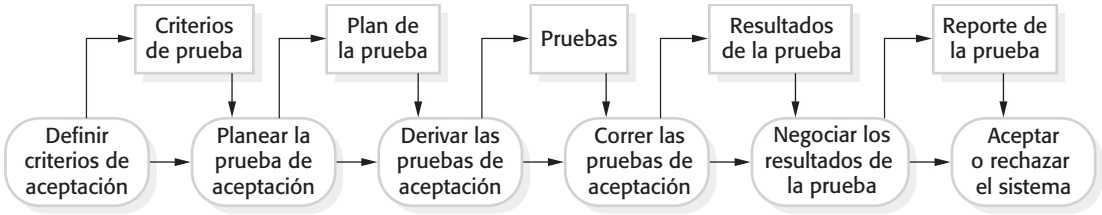


Figura 8.11 Proceso de prueba de aceptación

Los examinadores beta pueden ser un grupo selecto de clientes que sean adoptadores tempranos del sistema. De manera alternativa, el software se pone a disposición pública para uso de quienquiera que esté interesado en él. Las pruebas beta se usan sobre todo para productos de software que se emplean en entornos múltiples y diferentes (en oposición a los sistemas personalizados, que se utilizan por lo general en un entorno definido). Es imposible que los desarrolladores de producto conozcan y repliquen todos los entornos donde se usará el software. En consecuencia, las pruebas beta son esenciales para descubrir problemas de interacción entre el software y las características del entorno donde se emplea. Las pruebas beta también son una forma de comercialización: los clientes aprenden sobre su sistema y lo que puede hacer por ellos.

Las pruebas de aceptación son una parte inherente del desarrollo de sistemas personalizados. Tienen lugar después de las pruebas de versión. Implican a un cliente que prueba de manera formal un sistema, para decidir si debe o no aceptarlo del desarrollador del sistema. La aceptación implica que debe realizarse el pago por el sistema.

Existen seis etapas en el proceso de pruebas de aceptación, como se muestra en la figura 8.11. Éstas son:

1. *Definir los criterios de aceptación* Esta etapa debe, de manera ideal, anticiparse en el proceso, antes de firmar el contrato por el sistema. Los criterios de aceptación forman parte del contrato del sistema y tienen que convenirse entre el cliente y el desarrollador. Sin embargo, en la práctica suele ser difícil definir los criterios de manera tan anticipada en el proceso. Es posible que no estén disponibles requerimientos detallados y que haya cambios significativos en los requerimientos durante el proceso de desarrollo.

2. *Plan de pruebas de aceptación* Esto incluye decidir sobre los recursos, el tiempo y el presupuesto para las pruebas de aceptación, así como establecer un calendario de pruebas. El plan de pruebas de aceptación debe incluir también la cobertura requerida de los requerimientos y el orden en que se prueban las características del sistema. Tiene que definir riesgos al proceso de prueba, como caídas del sistema y rendimiento inadecuado, y resolver cómo mitigar dichos riesgos.

3. *Derivar pruebas de aceptación* Una vez establecidos los criterios de aceptación, tienen que diseñarse pruebas para comprobar si un sistema es aceptable o no. Las pruebas de aceptación deben dirigirse a probar tanto las características funcionales como las no funcionales del sistema (por ejemplo, el rendimiento). Lo ideal sería que dieran cobertura completa a los requerimientos del sistema. En la práctica, es difícil establecer criterios de aceptación completamente objetivos. Con frecuencia hay espacio para argumentar sobre si las pruebas deben mostrar o no que un criterio se cubre de manera definitiva.

4. *Correr pruebas de aceptación* Las pruebas de aceptación acordadas se ejecutan sobre el sistema. De manera ideal, esto debería ocurrir en el entorno real donde se usará el sistema, pero esto podría ser perturbador y poco práctico. En consecuencia, quizá deba establecerse un entorno de pruebas de usuario para efectuar dichas pruebas. Es difícil automatizar este proceso, ya que parte de las pruebas de aceptación podría necesitar poner a prueba las interacciones entre usuarios finales y el sistema. Es posible que se requiera cierta capacitación de los usuarios finales.

5. *Negociar los resultados de las pruebas* Es poco probable que se pasen todas las pruebas de aceptación definidas y que no haya problemas con el sistema. Si éste es el caso, entonces las pruebas de aceptación están completas y el sistema está listo para entregarse. Con mayor regularidad se descubrirán algunos problemas. En tales casos, el desarrollador y el cliente tienen que negociar para decidir si el sistema es suficientemente adecuado para ponerse en uso. También deben acordar sobre la respuesta del desarrollador para identificar problemas.

6. *Rechazo/aceptación del sistema* Esta etapa incluye una reunión entre los desarrolladores y el cliente para decidir si el sistema debe aceptarse o no. Si el sistema no es suficientemente bueno para usarse, entonces se requiere mayor desarrollo para corregir los problemas identificados. Una vez completo, se repite la fase de pruebas de aceptación.

En los métodos ágiles, como XP, las pruebas de aceptación tienen un significado un tanto diferente. En principio, comparten la noción de que son los usuarios quienes deciden si el sistema es aceptable o no. Sin embargo, en XP, el usuario forma parte del equipo de desarrollo (es decir, es un examinador alfa) y proporciona los requerimientos del sistema en términos de historias de usuario. También es responsable de definir las pruebas, que permiten determinar si el software desarrollado soporta o no la historia del usuario. Las pruebas son automatizadas y el desarrollo no avanza sino hasta que se pasan las pruebas de aceptación históricas. Por consiguiente, no hay una actividad separada de pruebas de aceptación.

Como se estudió en el capítulo 3, un problema con la participación del usuario es garantizar que quien se inserte en el equipo de desarrollo sea un usuario "típico" con conocimiento general de cómo se usará el sistema. Quizá sea difícil encontrar a tal usuario y, por lo tanto, las pruebas de aceptación en realidad tal vez no sean un verdadero reflejo de la práctica. Más aún, el requerimiento de pruebas automatizadas limita severamente la flexibilidad de los sistemas interactivos de pruebas. Para tales sistemas, las pruebas de aceptación podrían requerir que grupos de usuarios finales usen el sistema como si fuera parte de su trabajo cotidiano.

Usted puede considerar que las pruebas de aceptación son un conflicto contractual tajante. Si un sistema no pasa sus pruebas de aceptación, debe rechazarse y el pago no se realiza. Sin embargo, la realidad es más compleja. Los clientes quieren usar el software tan pronto como puedan debido a los beneficios de su despliegue inmediato. Ellos quizá compraron un nuevo hardware, capacitaron al personal y modificaron sus procesos. Tal vez están deseosos de aceptar el software, sin importar los problemas, ya que los costos por no usar el software serían mayores que los de trabajar en torno a los problemas. Por consiguiente, el resultado de las negociaciones podría ser la aceptación condicional del sistema. El cliente acepta tal sistema para comenzar el despliegue. El proveedor del sistema acuerda reparar los problemas urgentes y entregar una nueva versión al cliente tan rápido como sea posible.

PUNTOS CLAVE

- Las pruebas sólo pueden mostrar la presencia de errores en un programa. Si embargo, no pueden garantizar que no surjan fallas posteriores.
- Las pruebas de desarrollo son responsabilidad del equipo de desarrollo del software. Un equipo independiente debe responsabilizarse de probar un sistema antes de darlo a conocer a los clientes. En el proceso de pruebas de usuario, clientes o usuarios del sistema brindan datos de prueba y verifican que las pruebas sean exitosas.
- Las pruebas de desarrollo incluyen pruebas de unidad, donde se examinan objetos y métodos individuales; pruebas de componente, donde se estudian grupos de objetos relacionados; y pruebas del sistema, donde se analizan sistemas parciales o completos.
- Cuando pruebe software, debe tratar de "romperlo" mediante la experiencia y los lineamientos que elijan los tipos de casos de prueba que hayan sido efectivos para descubrir defectos en otros sistemas.
- Siempre que sea posible, se deben escribir pruebas automatizadas. Las pruebas se incrustan en un programa que puede correrse cada vez que se hace un cambio al sistema.
- El desarrollo de la primera prueba es un enfoque de desarrollo, donde las pruebas se escriben antes de que se pruebe el código. Se realizan pequeños cambios en el código, y éste se refactoriza hasta que todas las pruebas se ejecuten exitosamente.
- Las pruebas de escenario son útiles porque imitan el uso práctico del sistema. Implican trazar un escenario de uso típico y utilizarlo para derivar casos de prueba.
- Las pruebas de aceptación son un proceso de prueba de usuario, donde la meta es decidir si el software es suficientemente adecuado para desplegarse y utilizarse en su entorno operacional.

LECTURAS SUGERIDAS

"How to design practical test cases". Un artículo práctico sobre el diseño de casos de prueba elaborado por un publicista de una compañía japonesa, que tiene una muy buena reputación debido a que entrega el software con muy pocas fallas. (T. Yamaura, *IEEE Software*, **15** (6), noviembre 1998.) http://dx.doi.org/10.1109/52.730835.

How to Break Software: A Practical Guide to Testing. Se trata de un libro más práctico que teórico, sobre las pruebas de software. El autor presenta un conjunto de lineamientos basados en su experiencia relativa al diseño de pruebas, que probablemente sean efectivas en la detección de fallas del sistema. (J. A. Whittaker, Addison-Wesley, 2002.)

"Software Testing and Verification". Este número especial del *IBM Systems Journal* comprende algunos ensayos de pruebas, incluido un buen panorama. Además, incluye ensayos de métricas de prueba y automatización de pruebas. (*IBM Systems Journal*, **41** (1), enero 2002.)

"Test-driven development". Este número especial es acerca del desarrollo dirigido por pruebas, el cual incluye un buen panorama general del TDD, así como ensayos de experiencia sobre cómo se usó el TDD para diferentes tipos de software. (*IEEE Software*, **24** (3) mayo/junio 2007.)

EJERCICIOS

8.1. Explique por qué no es necesario que un programa esté completamente libre de defectos antes de entregarse a sus clientes.

8.2. Indique por qué las pruebas sólo pueden detectar la presencia de errores, pero no su ausencia.

8.3. Algunas personas argumentan que los desarrolladores no deben intervenir en las pruebas de su propio código, sino que todas las pruebas deben ser responsabilidad de un equipo independiente. Exponga argumentos en favor y en contra de las pruebas efectuadas por parte de los mismos desarrolladores.

8.4. Se pide al lector poner a prueba un método llamado "catWhiteSpace" en un objeto "Paragraph" que, dentro del párrafo, sustituye secuencias de caracteres blancos con un solo carácter blanco. Identifique las particiones de prueba para este ejemplo y derive un conjunto de pruebas para el método "catWhiteSpace".

8.5. ¿Qué es la prueba de regresión? Explique cómo el uso de pruebas automatizadas y un marco de pruebas como JUnit simplifican las pruebas de regresión.

8.6. El MHC-PMS se construyó al adaptar un sistema de información comercial. ¿Cuáles considera que son las diferencias entre probar tal sistema y probar el software que se desarrolló usando un lenguaje orientado a objetos como Java?

8.7. Diseñe un escenario que pueda usar para ayudarse a elaborar pruebas para el sistema de estación meteorológica en campo abierto.

8.8. ¿Qué entiende por "pruebas de esfuerzo"? Sugiera cómo puede hacer una prueba de esfuerzo del MHC-PMS.

8.9. ¿Cuáles son los beneficios de hacer participar a usuarios en las pruebas de versión en una etapa temprana del proceso de pruebas? ¿Hay desventajas en la implicación del usuario?

8.10. Un enfoque común a las pruebas del sistema es probar el sistema hasta que se agote el presupuesto de pruebas y, luego, entregar el sistema a los clientes. Discuta la ética de este enfoque para sistemas que se entregan a clientes externos.

REFERENCIAS

Andrea, J. (2007). "Envisioning the Next Generation of Functional Testing Tools". *IEEE Software*, 24 (3), 58–65.

Beck, K. (2002). *Test Driven Development: By Example*. Boston: Addison-Wesley.

Bezier, B. (1990). *Software Testing Techniques, 2nd edition*. New York: Van Nostrand Rheinhold.

Boehm, B. W. (1979). "Software engineering; R & D Trends and defense needs." In *Research Directions in Software Technology*. Wegner, P. (ed.). Cambridge, Mass.: MIT Press. 1–9.

Cusamano, M. y Selby, R. W. (1998). *Microsoft Secrets*. New York: Simon and Shuster.

Dijkstra, E. W., Dahl, O. J. y Hoare, C. A. R. (1972). *Structured Programming*. Londres: Academic Press.

Fagan, M. E. (1986). "Advances in Software Inspections". *IEEE Trans. on Software Eng*., S**E-12** (7), 744–51.

Jeffries, R. y Melnik, G. (2007). "TDD: The Art of Fearless Programming". *IEEE Software*, **24**, 24–30.

Kaner, C. (2003). "The power of 'What If . . .' and nine ways to fuel your imagination: Cem Kaner on scenario testing". *Software Testing and Quality Engineering*, **5** (5), 16–22.

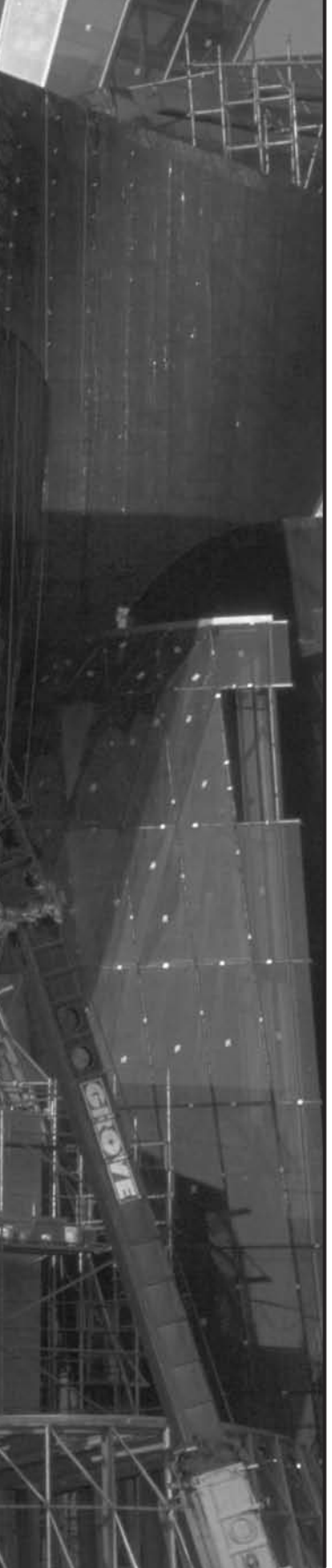
Lutz, R. R. (1993). "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems". RE'93, San Diego, Calif.: IEEE.

Martin, R. C. (2007). "Professionalism and Test-Driven Development". *IEEE Software*, **24** (3), 32–6.

Massol, V. y Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Prowell, S. J., Trammell, C. J., Linger, R. C. y Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.

Whittaker, J. W. (2002). *How to Break Software: A Practical Guide to Testing*. Boston: Addison-Wesley.



9

Evolución del software

Objetivos

Los objetivos de este capítulo son explicar por qué la evolución del software forma parte importante de la ingeniería de software, así como describir los procesos de evolución del software. Al estudiar este capítulo:

- comprenderá que el cambio es inevitable si los sistemas de software deben mantener su utilidad, y que el desarrollo y la evolución del software pueden integrarse en un modelo espiral;
- entenderá los procesos de evolución del software y las influencias sobre dichos procesos;
- aprenderá sobre los diferentes tipos de mantenimiento de software y los factores que afectan los costos de mantenimiento; y
- conocerá cómo pueden valorarse los sistemas heredados para decidir si se deben descartar, mantener, someter a reingeniería o sustituir.

Contenido

9.1 Procesos de evolución
9.2 Evolución dinámica del programa
9.3 Mantenimiento del software
9.4 Administración de sistemas heredados

El desarrollo del software no se detiene cuando un sistema se entrega, sino que continúa a lo largo de la vida de éste. Después de distribuir un sistema, inevitablemente debe modificarse, con la finalidad de mantenerlo útil. Tanto los cambios empresariales como los de las expectativas del usuario generan nuevos requerimientos para el software existente. Es posible que tengan que modificarse partes del software para corregir errores encontrados durante su operación, para adaptarlo a los cambios en su plataforma de software y hardware, y para mejorar su rendimiento u otras características no funcionales.

La evolución del software es importante porque las organizaciones invierten grandes cantidades de dinero en él y en la actualidad son completamente dependientes de dichos sistemas. Sus sistemas se consideran activos empresariales críticos, por lo que tienen que invertir en el cambio del sistema para mantener el valor de estos activos. En consecuencia, las compañías más grandes gastan más en conservar los sistemas existentes que en el desarrollo de sistemas nuevos. Con base en una encuesta industrial informal, Erlikh (2000) sugiere que entre el 85 y 90% de los costos del software organizacional son costos de evolución; mientras que otros estudios sugieren que éstos conforman alrededor de dos tercios de los costos del software. Desde luego, los costos del cambio del software representan una gran parte del presupuesto de TI de todas las compañías.

La evolución del software puede potenciarse al cambiar los requerimientos empresariales, con reportes de defectos del software o por cambios a otros sistemas en un entorno del sistema de software. Hopkins y Jenkins (2008) acuñaron el término "desarrollo de software abandonado" (subutilizado) para describir situaciones en que los sistemas de software tienen que desarrollarse y gestionarse en un ambiente donde dependen de muchos otros sistemas de software.

Por consiguiente, la evolución de un sistema rara vez puede considerarse en aislamiento. Los cambios al entorno conducen a cambios en el sistema que, a la vez, pueden generar más cambios en el entorno. Desde luego, el hecho de que los sistemas tengan que evolucionar en un ambiente "rico en sistemas" con frecuencia aumenta las dificultades y los costos de la evolución. Además de comprender y analizar el impacto de un cambio propuesto sobre el sistema en sí, también es probable que se deba valorar cómo esto afectaría a otros sistemas en el entorno operacional.

Por lo general, los sistemas de software útiles tienen una vida muy larga. Por ejemplo, los grandes sistemas militares o de infraestructura, como los de control de tráfico aéreo, llegan a durar 30 años o más; en tanto que los sistemas empresariales con frecuencia superan los 10 años. Puesto que el costo del software es elevado, una compañía debe usar un sistema de software durante muchos años para recuperar su inversión. Evidentemente, los requerimientos de los sistemas instalados cambian conforme lo hacen el negocio y su entorno. Por consiguiente, se crean a intervalos regulares nuevas versiones de los sistemas, las cuales incorporan cambios y actualizaciones.

Por ende, la ingeniería de software se debe considerar como un proceso en espiral, con requerimientos, diseño, implementación y pruebas continuas, a lo largo de la vida del sistema (figura 9.1). Esto comienza por crear la versión 1 del sistema. Una vez entregada, se proponen cambios y casi de inmediato comienza el desarrollo de la versión 2. De hecho, la necesidad de evolución puede volverse evidente incluso antes de que el sistema se distribuya, de manera que las futuras versiones del software estarían en desarrollo antes de que se libere la versión actual.

Este modelo de evolución de software implica que una sola organización es responsable tanto del desarrollo del software inicial como de la evolución del software. La mayoría de los productos de software empacados se desarrollan siguiendo este enfoque. Para el software personalizado, por lo general se utiliza un enfoque diferente. Una compañía de

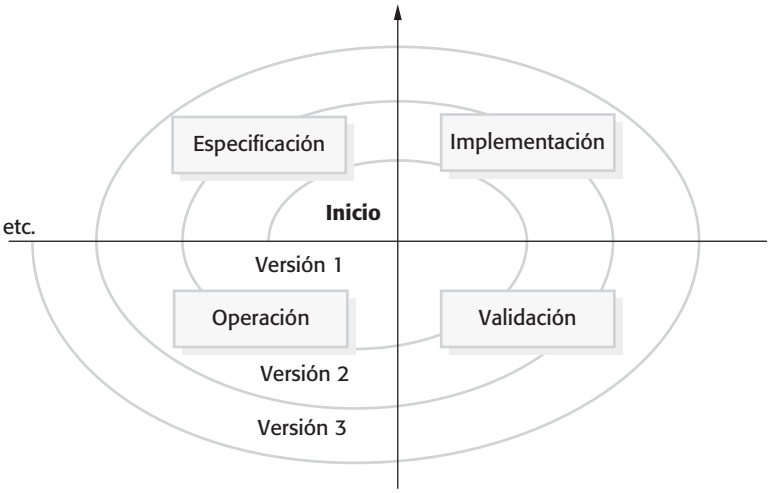


Figura 9.1 Modelo en espiral de desarrollo y evolución

software lo desarrolla para un cliente y, luego, el personal de desarrollo del propio cliente se hace cargo del sistema. Ellos son los responsables de la evolución del software. De forma alternativa, el cliente del software puede otorgar por separado un contrato a una compañía diferente, con la finalidad de que dé soporte al sistema y continuar su evolución.

En este caso, es probable que existan discontinuidades en el proceso espiral. Los documentos de requerimientos y diseño quizá no se compartan entre una compañía y otra. Éstas podrían fusionarse o reorganizarse y heredar el software de otras compañías, para luego descubrir que este último tiene que cambiarse. Cuando la transición del desarrollo a la evolución no es uniforme, el proceso de cambiar el software después de la entrega se conoce como "mantenimiento de software". Como se analizará más adelante en este capítulo, el mantenimiento incluye actividades de proceso adicionales, como la comprensión del programa, además de las actividades normales de desarrollo del software.

Rajlich y Bennett (2000) propusieron una visión alternativa del ciclo de vida de la evolución del software, como se indica en la figura 9.2. En ese modelo, distinguen entre evolución y servicio. La evolución es la fase donde es posible hacer cambios significativos a la arquitectura y la funcionalidad del software. Durante el servicio, los únicos cambios que se realizan son relativamente pequeños.

Durante la evolución, el software se usa con éxito y hay un flujo constante de propuestas de cambios a los requerimientos. Sin embargo, conforme el software se modifica, su estructura tiende a degradarse y los cambios se vuelven más y más costosos. Esto sucede con frecuencia después de algunos años de uso, cuando también se requieren otros cambios ambientales como el hardware y los sistemas operativos. En alguna etapa de su ciclo de vida, el software alcanza un punto de transición donde los cambios significativos que implementan nuevos requerimientos se vuelven cada vez menos rentables.

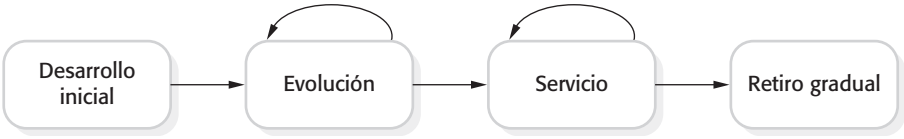


Figura 9.2 Evolución y servicio

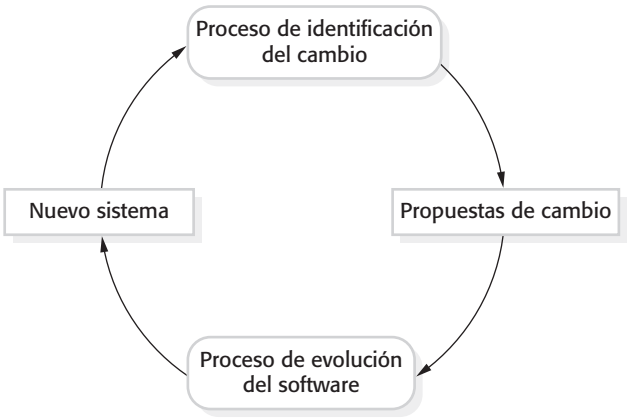


Figura 9.3 Identificación del cambio y procesos de evolución

En dicha fase, el software avanza de la evolución al servicio. Durante la fase de servicio, el software todavía es útil y se utiliza, pero sólo se le realizan pequeños cambios tácticos. Durante esta fase, la compañía normalmente considera cómo reemplazar el software. En la fase final, de retiro gradual (*Phase-out*), el software todavía puede usarse, aunque no se implementan más cambios. Los usuarios tienen que sobrellevar cualquier problema que descubran.

9.1 Procesos de evolución

Los procesos de evolución del software varían dependiendo del tipo de software que se mantiene, de los procesos de desarrollo usados en la organización y de las habilidades de las personas que intervienen. En algunas organizaciones, la evolución es un proceso informal, donde las solicitudes de cambios provienen sobre todo de conversaciones entre los usuarios del sistema y los desarrolladores. En otras compañías, se trata de un proceso formalizado con documentación estructurada generada en cada etapa del proceso.

Las propuestas de cambio al sistema son el motor para la evolución del sistema en todas las organizaciones. Estos cambios provienen de requerimientos existentes que no se hayan implementado en el sistema liberado, de peticiones de nuevos requerimientos, de reportes de bugs de los participantes del sistema, y de nuevas ideas para la mejora del software por parte del equipo de desarrollo del sistema. Los procesos de identificación de cambios y evolución del sistema son cíclicos y continúan a lo largo de la vida de un sistema (figura 9.3).

Las propuestas de cambio deben vincularse con los componentes del sistema que se van a modificar para implementar dichas propuestas. Esto permite que el costo y el impacto del cambio logren valorarse. Lo anterior forma parte del proceso general de la administración del cambio, que también debe garantizar la inclusión de versiones correctas de los componentes en cada versión del sistema. En el capítulo 25 se estudiará la administración del cambio y de la configuración.

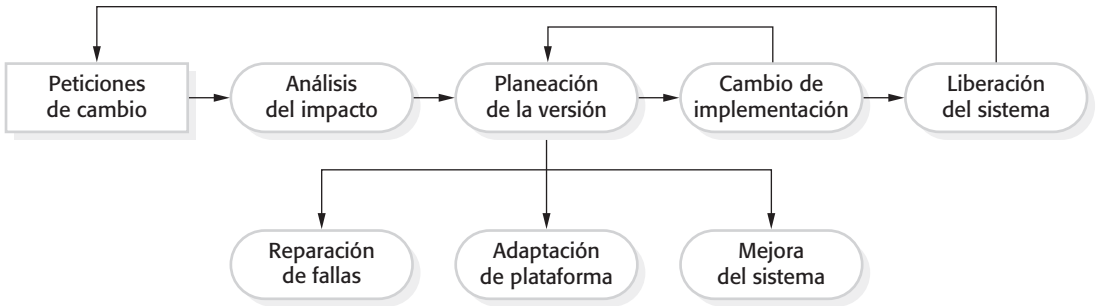


Figura 9.4 Proceso de evolución del software

La figura 9.4, adaptada de Arthur (1988), muestra un panorama del proceso de evolución, el cual incluye actividades fundamentales de análisis del cambio, planeación de la versión, implementación del sistema y su liberación a los clientes. El costo y el impacto de dichos cambios se valoran para saber qué tanto resultará afectado el sistema por el cambio y cuánto costaría implementarlo. Si los cambios propuestos se aceptan, se planea una nueva versión del sistema. Durante la planeación de la versión se consideran todos los cambios propuestos (reparación de fallas, adaptación y nueva funcionalidad). Entonces se toma una decisión acerca de cuáles cambios implementar en la siguiente versión del sistema. Después de implementarse, se valida y se libera una nueva versión del sistema. Luego, el proceso se repite con un conjunto nuevo de cambios propuestos para la siguiente liberación.

Es posible considerar la implementación del cambio como una iteración del proceso de desarrollo, donde las revisiones al sistema se diseñan, se aplican y se ponen a prueba. Sin embargo, una diferencia crítica es que la primera etapa de implementación del cambio puede involucrar la comprensión del programa, sobre todo si los desarrolladores del sistema original no son los responsables de implementar el cambio. Durante esta fase de comprensión del programa, hay que entender cómo está estructurado, cómo entrega funcionalidad y cómo lo afectaría el cambio propuesto. Se necesita de esta comprensión para asegurarse de que el cambio implementado no cause nuevos problemas, cuando se introduzca al sistema existente.

De manera ideal, la etapa de implementación del cambio de este proceso debe modificar la especificación, el diseño y la implementación del sistema para reflejar los cambios al mismo (figura 9.5). Se proponen, analizan y validan los nuevos requerimientos que reflejan los cambios al sistema. Los componentes del sistema se rediseñan e implementan, y el sistema vuelve a probarse. Si es adecuado, como parte del proceso de análisis de cambio, podrían elaborarse prototipos con los cambios propuestos.

Durante el proceso de evolución, se analizan a detalle los requerimientos y surgen implicaciones de los cambios que no eran aparentes al comienzo del proceso de análisis de cambio. Esto significa que los cambios propuestos pueden modificarse y quizá se requieran más pláticas con el cliente antes de la implementación.

En ocasiones, las peticiones de cambio se relacionan con problemas del sistema que tienen que enfrentarse de manera urgente. Estos cambios urgentes surgen básicamente por tres razones:

1. Si ocurre una falla seria del sistema que deba repararse para permitir que continúe la operación normal.

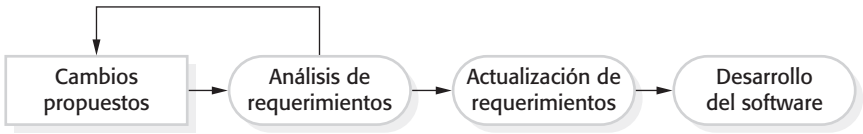


Figura 9.5 Implementación del cambio

Figura 9.6 Proceso de reparación de emergencia

2. Si los cambios a los sistemas que operan el entorno tienen efectos inesperados que perturban la operación normal.
3. Si hay cambios no anticipados a la empresa que opera el sistema, como el surgimiento de competidores nuevos o la introducción de una nueva legislación que afecte al sistema.

En tales casos, la necesidad de realizar el cambio rápidamente significa que quizá no pueda seguir el proceso formal de análisis de cambio. En vez de modificar los requerimientos y el diseño, se puede hacer una reparación de emergencia al programa para resolver el problema de inmediato (figura 9.6). Sin embargo, el riesgo es que los requerimientos, el diseño del software y el código se vuelvan inconsistentes. Aunque se trate de documentar el cambio en los requerimientos y el diseño, es posible que el software requiera reparaciones de emergencia adicionales, las cuales tienen prioridad sobre la documentación. Con el tiempo, el cambio original se olvida y la documentación y el código del sistema nunca se realinean.

Por lo general, las reparaciones de emergencia del sistema tienen que completarse tan rápido como sea posible. Se elige una solución rápida y viable, en lugar de la mejor solución en cuanto a la estructura del sistema. Esto acelera el proceso de degeneración del software, de modo que los cambios futuros se vuelven progresivamente más difíciles y, con ello, aumenta el costo de mantenimiento.

Idealmente, cuando se hacen reparaciones de emergencia al código, la petición de cambio debería quedar pendiente después de reparar las fallas de código. Entonces, se pueden volver a implementar de manera más cuidadosa después de mayor análisis. Desde luego, el código de la reparación es reutilizable. Una mejor solución al problema surgirá cuando haya más tiempo disponible para el análisis. Sin embargo, en la práctica es casi inevitable que dichas mejoras tengan baja prioridad. Con frecuencia se olvidan y, si se realizan más cambios al sistema, entonces se vuelve imposible reelaborar las reparaciones de emergencia.

Los métodos y procesos ágiles, que se estudiaron en el capítulo 3, se utilizan tanto para la evolución del programa como para su desarrollo. De hecho, puesto que dichos métodos se basan en el desarrollo incremental, hacer la transición del desarrollo ágil a la evolución posterior a la entrega no debería tener complicaciones. Las técnicas como las pruebas de regresión automatizadas son útiles cuando se realizan cambios al sistema. Los cambios pueden expresarse como historias de usuario, y el involucramiento del cliente priorizará los cambios que se requieran en un sistema operacional. En resumen, la evolución simplemente implica la continuación del proceso de desarrollo ágil.

Sin embargo, es posible que surjan problemas en situaciones donde haya transferencia de un equipo de desarrollo a un equipo separado responsable de la evolución. Existen dos situaciones potencialmente problemáticas:

1. Donde el equipo de desarrollo haya usado un enfoque ágil, pero el equipo de evolución no esté familiarizado con los métodos ágiles y prefiera un enfoque basado en un plan. Quizás el equipo de evolución espere documentación detallada para apoyar la evolución, y esto rara vez sucede en los procesos ágiles. Podría no haber un enunciado definitivo de los requerimientos del sistema que sea modificable conforme se realizan cambios al sistema.
2. Donde se haya usado un enfoque basado en un plan para el desarrollo, pero el equipo de evolución prefiera usar métodos ágiles. En este caso, el equipo de evolución tal vez deba comenzar desde cero para desarrollar pruebas automatizadas y, además, es posible que el código en el sistema no se haya refactorizado y simplificado como se espera en el desarrollo ágil. En este caso, tal vez se requiera algo de reingeniería para mejorar el código, antes de usarlo en un proceso de desarrollo ágil.

Poole y Huisman (2001) reportan sus experiencias en el uso de programación extrema, para mantener un sistema grande que originalmente se desarrolló usando un enfoque basado en un plan. Después de someter el sistema a reingeniería para mejorar su estructura, XP se usó con mucho éxito en el proceso de mantenimiento.

9.2 Evolución dinámica del programa

La dinámica de evolución del programa es el estudio del cambio al sistema. En las décadas de 1970 y 1980, Lehman y Belady (1985) realizaron varios estudios empíricos acerca del cambio al sistema, con una visión para entender más sobre las características de la evolución del software. El trabajo continuó en la década de 1990, conforme Lehman y sus colegas investigaron la importancia de la retroalimentación en los procesos de evolución (Lehman, 1996; Lehman *et al*., 1998; Lehman *et al*., 2001). A partir de estos estudios, propusieron las "leyes de Lehman" relacionadas al cambio del sistema (figura 9.7).

Lehman y Belady afirman que dichas leyes suelen ser verdaderas para todos los tipos de sistemas de software organizacional grandes (los llamados sistemas tipo E). Se trata de sistemas en los cuales los requerimientos se modifican para reflejar las necesidades cambiantes de la empresa. Las nuevas versiones del sistema son esenciales para que éste proporcione valor al negocio.

La primera ley afirma que el mantenimiento del sistema es un proceso inevitable. A medida que cambia el entorno del sistema, surgen nuevos requerimientos y el sistema debe modificarse. Cuando el sistema modificado se reintroduce al entorno, promueve más cambios ambientales, de manera que el proceso de evolución comienza de nuevo.

La segunda ley afirma que, conforme cambia un sistema, su estructura se degrada. La única manera de evitar que esto ocurra es invertir en mantenimiento preventivo. Se invierte tiempo mejorando la estructura del software sin agregar nada a su funcionalidad. Evidentemente, esto significa costos adicionales, por encima de los asignados para implementar los cambios requeridos al sistema.

Ley	Descripción
Cambio continuo	Un programa usado en un entorno real debe cambiar; de otro modo, en dicho entorno se volvería progresivamente inútil.
Complejidad creciente	A medida que cambia un programa en evolución, su estructura tiende a volverse más compleja. Deben dedicarse recursos adicionales para conservar y simplificar su estructura.
Evolución de programa grande	La evolución del programa es un proceso autorregulador. Los atributos del sistema, como tamaño, tiempo entre versiones y número de errores reportados, son casi invariantes para cada versión del sistema.
Estabilidad organizacional	Durante la vida de un programa, su tasa de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.
Conservación de familiaridad	A lo largo de la existencia de un sistema, el cambio incremental en cada liberación es casi constante.
Crecimiento continuo	La funcionalidad ofrecida por los sistemas tiene que aumentar continuamente para mantener la satisfacción del usuario.
Declive de calidad	La calidad de los sistemas declinará, a menos que se modifiquen para reflejar los cambios en su entorno operacional.
Sistema de retroalimentación	Los procesos de evolución incorporan sistemas de retroalimentación multiagente y multiciclo. Además, deben tratarse como sistemas de retroalimentación para lograr una mejora significativa del producto.

Figura 9.7 Leyes de Lehman

La tercera ley es, quizá, la más interesante y polémica de las leyes de Lehman. Sugiere que los sistemas grandes tienen una dinámica propia que se establece en una etapa temprana del proceso de desarrollo. Esto determina las grandes tendencias del proceso de mantenimiento del sistema y limita el número de cambios posibles al sistema. Lehman y Belady sugieren que esta ley es consecuencia de factores estructurales que influyen en el cambio al sistema y lo restringen, así como de factores organizacionales que afectan el proceso de evolución.

Los factores estructurales que afectan la tercera ley provienen de la complejidad de los sistemas grandes. Conforme cambia y se extiende un programa, su estructura tiende a degradarse. Esto es verdadero para todos los tipos de sistemas (no sólo para el software) y ocurre porque una estructura con un propósito específico se adapta para uno diferente. Esta degradación, si no se controla, hace cada vez más difícil realizar cambios ulteriores al programa. Hacer pequeños cambios reduce el alcance de la degradación estructural y, por consiguiente, aminora los riesgos de causar serios problemas a la confiabilidad del sistema. Si se realizan grandes cambios, hay una alta probabilidad de que se introduzcan nuevas fallas, los cuales impedirían más cambios al programa.

Los factores organizacionales que afectan la tercera ley reflejan el hecho de que, por lo general, los sistemas grandes se producen en organizaciones grandes. Estas compañías tienen burocracias internas que establecen los presupuestos de cambio para cada sistema y controlan el proceso de toma de decisiones. Las empresas deben tomar decisiones sobre los riesgos y el valor de los cambios, así como sobre los costos inherentes. Tales decisiones toman tiempo y, en ocasiones, tardan más para decidir acerca de los cambios por

realizar que para implementarlos. En consecuencia, la rapidez de los procesos de toma de decisiones de la organización controla la tasa de cambio del sistema.

La cuarta ley de Lehman sugiere que la mayoría de los grandes proyectos de programación funcionan en un estado "saturado". Es decir, un cambio en los recursos o en el personal tiene efectos imperceptibles en la evolución a largo plazo del sistema. Esto es congruente con la tercera ley, que sugiere que la evolución del programa es en gran medida independiente de las decisiones administrativas. Esta ley confirma que los grandes equipos de desarrollo de software con frecuencia son improductivos, porque los gastos en comunicación dominan el trabajo del equipo.

La quinta ley de Lehman se relaciona con los incrementos de cambio en cada versión del sistema. Agregar nueva funcionalidad a un sistema introduce inevitablemente nuevas fallas al mismo. Cuanto más funcionalidad se agregue en cada versión, más fallas habrá. Por consiguiente, un gran incremento en funcionalidad en una versión del sistema significa que esto tendrá que seguir en una versión ulterior, donde se reparen las fallas del nuevo sistema. A dicha versión debe agregarse relativamente poca funcionalidad. Esta ley sugiere que no se deben presupuestar grandes incrementos de funcionalidad en cada versión, sin tomar en cuenta la necesidad de reparación de las fallas.

Las primeras cinco leyes fueron las propuestas iniciales de Lehman; las leyes restantes se agregaron después de un trabajo posterior. Las leyes sexta y séptima son similares y, en esencia, indican que los usuarios de software se volverán cada vez más infortunados con el sistema, a menos que se le mantenga y se le agregue nueva funcionalidad. La ley final refleja el trabajo más reciente sobre los procesos de retroalimentación, aunque todavía no está claro cómo se aplica en el desarrollo de software práctico.

En general, las observaciones de Lehman parecen sensatas. Hay que tomarlas en cuenta cuando se planee el proceso de mantenimiento. Podría suceder que las consideraciones empresariales requieran ignorarlas en algún momento. Por ejemplo, por razones de marketing, quizá sea necesario realizar muchos cambios significativos al sistema en una sola versión. La consecuencia probable de esto es que tal vez se requieran una o más versiones dedicadas a la reparación del error. A menudo esto se observa en el software de computadoras personales, cuando una nueva gran versión de alguna aplicación con frecuencia viene seguida por una actualización para reparar un bug.

9.3 Mantenimiento del software

El mantenimiento del software es el proceso general de cambiar un sistema después de que éste se entregó. El término usualmente se aplica a software personalizado, en el que grupos de desarrollo separados intervienen antes y después de la entrega. Los cambios realizados al software van desde los simples para corregir errores de codificación, los más extensos para corregir errores de diseño, hasta mejorías significativas para corregir errores de especificación o incorporar nuevos requerimientos. Los cambios se implementan modificando los componentes del sistema existentes y agregándole nuevos componentes donde sea necesario.

Existen tres tipos de mantenimiento de software:

1. *Reparaciones de fallas* Los errores de codificación por lo general son relativamente baratos de corregir; los errores de diseño son más costosos, ya que quizás impliquen la reescritura de muchos componentes del programa. Los errores de

requerimientos son los más costosos de reparar debido a que podría ser necesario un extenso rediseño del sistema.

2. *Adaptación ambiental* Este tipo de mantenimiento se requiere cuando algún aspecto del entorno del sistema, como el hardware, la plataforma operativa del sistema u otro soporte, cambia el software. El sistema de aplicación tiene que modificarse para lidiar con dichos cambios ambientales.

3. *Adición de funcionalidad* Este tipo de mantenimiento es necesario cuando varían los requerimientos del sistema, en respuesta a un cambio organizacional o empresarial. La escala de los cambios requeridos en el software suele ser mucho mayor que en los otros tipos de mantenimiento.

En la práctica, no hay una distinción clara entre estos tipos de mantenimiento. Cuando se adapta el sistema a un nuevo entorno, se puede agregar funcionalidad para sacar ventaja de las nuevas características del entorno. Las fallas de desarrollo del software con frecuencia quedan expuestas debido a que los usuarios usan el sistema en formas no anticipadas. Cambiar el sistema para adaptar su forma de trabajar es la mejor forma de corregir dichas fallas.

Estos tipos de mantenimiento se reconocen comúnmente, pero diferentes personas en ocasiones les dan diferentes nombres. De manera universal se usa el término "mantenimiento correctivo" para referirse al mantenimiento para reparación de fallas de desarrollo. Por otro lado, "mantenimiento adaptativo" algunas veces quiere decir adaptarse a un nuevo entorno y otras veces significa adaptar el software a nuevos requerimientos. "Mantenimiento perfectivo" a veces significa perfeccionar el software al implementar nuevos requerimientos; en otros casos representa mantener la funcionalidad del sistema, pero mejorando su estructura y rendimiento. Debido a la incertidumbre en torno a la nomenclatura, en este capítulo se evitó el uso de tales términos.

Se han realizado varios estudios sobre el mantenimiento del software que observan las relaciones entre mantenimiento y desarrollo, así como entre las diferentes actividades de mantenimiento (Krogstie *et al*., 2005; Lientz y Swanson, 1980; Nosek y Palvia, 1990; Sousa, 1998). Debido a diferencias en la terminología, los detalles de estos estudios no pueden compararse. A pesar de los cambios en tecnología y los diferentes dominios de aplicación, parece que, desde la década de 1980, ha sido notable el cambio moderado que ha habido en la distribución del esfuerzo de evolución.

Los estudios concuerdan ampliamente en que el mantenimiento del software toma una proporción más alta de presupuestos de TI que el nuevo desarrollo (casi dos tercios en mantenimiento y un tercio en desarrollo). También coinciden en que una mayor parte del presupuesto de mantenimiento se destina a la implementación de nuevos requerimientos, y no a la reparación de bugs. La figura 9.8 muestra una distribución aproximada de los costos de mantenimiento. Evidentemente, los porcentajes específicos variarán de una organización a otra pero, de manera universal, reparar las fallas del desarrollo del sistema no es la actividad de mantenimiento más costosa. Evolucionar el sistema para enfrentar nuevos entornos y requerimientos nuevos o cambiantes consume más esfuerzo de mantenimiento.

Los costos relativos de mantenimiento y del nuevo desarrollo varían de un dominio de aplicación a otro. Guimaraes (1983) descubrió que los costos de mantenimiento para sistemas de aplicación empresarial son ampliamente comparables con los costos de desarrollo del sistema. Para sistemas embebidos de tiempo real, los costos de mantenimiento fueron hasta cuatro veces mayores que los costos de desarrollo. Los requerimientos de alta

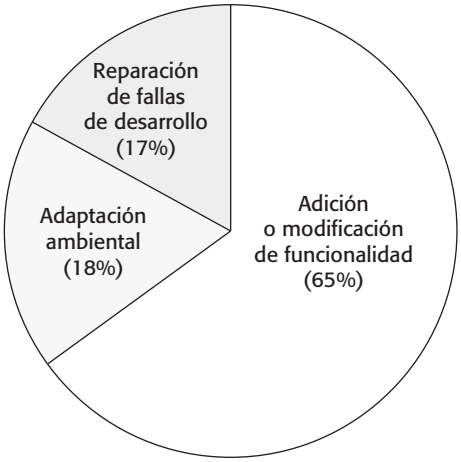


Figura 9.8 Distribución del esfuerzo de mantenimiento

fiabilidad y rendimiento de dichos sistemas significan que los módulos deben estar estrechamente ligados y, por lo tanto, son difíciles de cambiar. Aunque estas estimaciones tienen más de 25 años de antigüedad, es improbable que las distribuciones de costos para diferentes tipos de sistema hayan cambiado significativamente.

Por lo general, resulta efectivo en costo invertir esfuerzo en el diseño y la implementación de un sistema, con la finalidad de reducir los costos de cambios futuros. Agregar nueva funcionalidad después de la entrega es costoso porque toma tiempo aprender cómo funciona el sistema y analizar el impacto de los cambios propuestos. Por lo tanto, es posible que el trabajo realizado durante el desarrollo para hacer que el software sea más fácil de entender, y de cambiar, reduzca los costos de evolución. Las buenas técnicas de ingeniería de software, como la especificación precisa, el uso de desarrollo orientado a objetos y la administración de la configuración, contribuyen a reducir los costos de mantenimiento.

La figura 9.9 muestra cómo disminuyen los costos totales de la vida del sistema conforme se emplea más esfuerzo durante el desarrollo, para producir un sistema mantenible. Debido a la reducción potencial en costos de comprensión, análisis y pruebas, hay un significativo efecto multiplicador cuando el sistema se desarrolla para ser mantenible. Para el sistema 1, los $25,000 por costos adicionales de desarrollo se invirtieron para hacer al sistema más mantenible. Esto da como resultado un ahorro de $100 000 en costos de mantenimiento durante la vida del sistema. Esto supone que un aumento porcentual

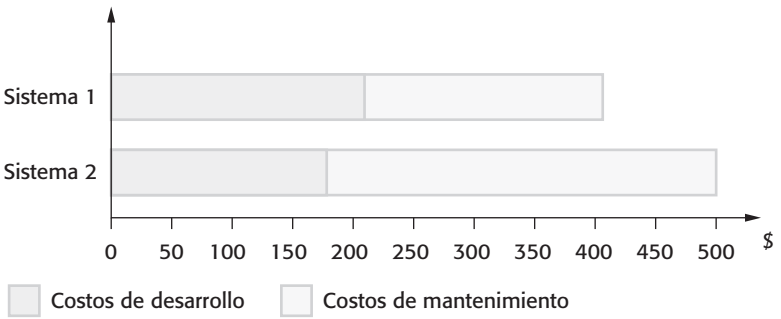


Figura 9.9 Costos de desarrollo y mantenimiento

Sistemas heredados

Los sistemas heredados son sistemas antiguos que todavía son útiles y en ocasiones críticos para la operación de la empresa. Pueden implementarse usando lenguajes y tecnología obsoletos, o utilizar otros sistemas que sean costosos de mantener. Normalmente su estructura se ha degradado por cambios y la documentación está extraviada o desactualizada. No obstante, quizá no sea efectivo en costo sustituir tales sistemas. Quizá sólo se usen algunas veces al año o sea demasiado riesgoso sustituirlos porque se hayan perdido las especificaciones.

http://www.SoftwareEngineering-9.com/Web/LegacySys/

en costos de desarrollo da como resultado una reducción porcentual comparable en costos totales del sistema.

Dichas estimaciones son hipotéticas; sin embargo, no hay duda de que desarrollar software para hacerlo más mantenible es efectivo en costo, si se toman en cuenta todos los costos de por vida. Ésta es la razón para refactorizar en el desarrollo ágil. Sin la refactorización, el código se vuelve cada vez más difícil y costoso de cambiar. Sin embargo, en el desarrollo basado en un plan, la realidad es que la inversión adicional en el mejoramiento del código rara vez se hace durante el desarrollo. Esto se debe principalmente a las formas en que la mayoría de las organizaciones aplican sus presupuestos. Invertir en la mantenibilidad conduce a aumentos de costo a corto plazo, los cuales son mensurables. Por desgracia, las ganancias a largo plazo no pueden medirse al mismo tiempo, de modo que las compañías son renuentes a gastar dinero en un rendimiento futuro incierto.

En general, resulta más costoso agregar funcionalidad después de que un sistema está en operación, que implementar la misma funcionalidad durante el desarrollo. Las razones son:

1. *Estabilidad del equipo* Después de que un sistema se entrega, es normal que el equipo de desarrollo se separe y que los individuos trabajen en nuevos proyectos. El nuevo equipo o los individuos responsables del mantenimiento del sistema no entienden el sistema o los antecedentes de las decisiones de diseño del mismo. Necesitan emplear tiempo para comprender el sistema existente, antes de implementar cambios en él.

2. *Práctica de desarrollo deficiente* El contrato para mantener un sistema por lo general está separado del contrato de desarrollo del sistema. El contrato de mantenimiento puede otorgarse a una compañía diferente, y no al desarrollador original del sistema. Este factor, junto con la falta de estabilidad del equipo, indica que no hay incentivo para que un equipo de desarrollo escriba software mantenible. Si el equipo de desarrollo puede buscar atajos para ahorrar esfuerzo durante el desarrollo, para ellos vale la pena hacerlo, incluso si esto significa que el software sea más difícil de cambiar en el futuro.

3. *Habilidades del personal* El personal de mantenimiento con frecuencia es relativamente inexperto y no está familiarizado con el dominio de aplicación. El mantenimiento tiene una mala imagen entre los ingenieros de software. Lo ven como un proceso que requiere menos habilidades que el desarrollo de sistemas y se asigna a menudo al personal más novato. Más aún, los sistemas antiguos pueden estar escritos en lenguajes de programación obsoletos. Es posible que el personal de mantenimiento no tenga mucha experiencia de desarrollo en estos lenguajes y debe aprenderlos para mantener el sistema.

Documentación

La documentación del sistema ayuda al proceso de mantenimiento al proporcionar a quienes dan mantenimiento información acerca de la estructura y la organización del sistema, así como de las características que ofrece a los usuarios del sistema. Aunque los defensores de los enfoques ágiles, como XP, sugieren que el código debe ser la principal documentación, los modelos de diseño de alto nivel y la información acerca de dependencias y restricciones facilitarán la comprensión y realización de cambios al código.

El autor escribió un capítulo aparte sobre documentación que el lector puede descargar.

http://www.SoftwareEngineering-9.com/Web/ExtraChaps/Documentation.pdf

4. *Antigüedad y estructura del programa* Conforme se realizan cambios al programa, su estructura tiende a degradarse. En consecuencia, a medida que los programas envejecen, se vuelven más difíciles de entender y cambiar. Algunos sistemas se desarrollaron sin técnicas modernas de ingeniería de software. Es posible que nunca hayan estado bien estructurados y tal vez estuvieron optimizados para eficiencia y no para comprensibilidad. La documentación del sistema puede estar perdida o ser inconsistente. Es posible que los sistemas antiguos no se hayan sujetado a una gestión rigurosa de configuración, de modo que se desperdicia tiempo para encontrar las versiones correctas de los componentes del sistema a cambiar.

Los primeros tres de estos problemas surgen del hecho de que muchas organizaciones todavía consideran el desarrollo y el mantenimiento como actividades separadas. El mantenimiento se ve como una actividad de segunda clase, y no hay incentivo para gastar dinero durante el desarrollo para reducir los costos del cambio de sistema. La única solución a largo plazo a este problema es aceptar que los sistemas rara vez tienen una vida definida pero continúan en uso, en cierta forma, durante un periodo indefinido. Como se sugirió en la introducción, se debe considerar que los sistemas evolucionan a lo largo de su vida durante un proceso de desarrollo continuo.

En el cuarto conflicto, el problema de la estructura degradada del sistema es el más sencillo de enfrentar. Las técnicas de reingeniería de software (que se describen más adelante en este capítulo) son aplicables para mejorar la estructura y comprensibilidad del sistema. Transformaciones arquitectónicas pueden adaptar el sistema a un hardware nuevo. La refactorización mejora la calidad del código del sistema y facilita el cambio.

9.3.1 Predicción de mantenimiento

Los gerentes aborrecen las sorpresas, sobre todo si derivan en costos inesperadamente elevados. Por consiguiente, se debe tratar de predecir qué cambios deben proponerse al sistema y qué partes del sistema es probable que sean las más difíciles de mantener. También hay que tratar de estimar los costos de mantenimiento globales para un sistema durante cierto lapso de tiempo. La figura 9.10 muestra dichas predicciones y las preguntas asociadas.

Predecir el número de peticiones de cambio para un sistema requiere un entendimiento de la relación entre el sistema y su ambiente externo. Algunos sistemas tienen una

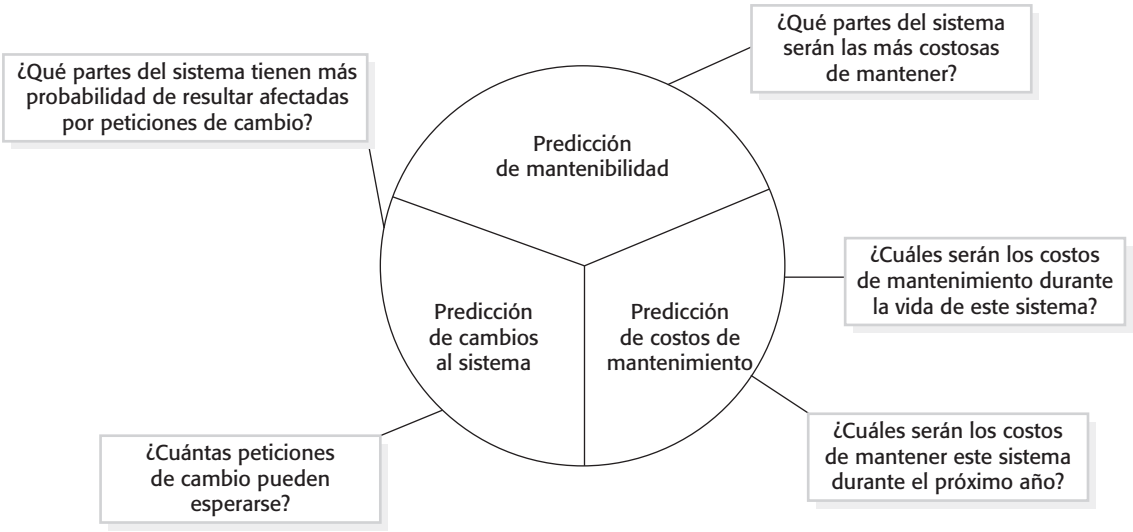


Figura 9.10
Predicción de mantenimiento

relación muy compleja con su ambiente externo, y los cambios a dicho entorno inevitablemente derivarán en cambios al sistema. Para evaluar las relaciones entre un sistema y su ambiente, se debe valorar:

1. *El número y la complejidad de las interfaces del sistema* Cuanto más grande sea el número de interfaces y más complejas sean dichas interfaces, más probable será que se requieran cambios de interfaz conforme se propongan nuevos requerimientos.
2. *El número de requerimientos de sistema inherentemente inestables* Como se estudió en el capítulo 4, es más probable que los requerimientos que reflejan políticas y procedimientos de la organización sean más inestables que los requerimientos que se basan en características de un dominio estable.
3. *Los procesos empresariales donde se usa el sistema* A medida que evolucionan los procesos empresariales, generan peticiones de cambio del sistema. Cuantos más procesos use un sistema, habrá más demandas de cambio del mismo.

Durante muchos años, los investigadores han observado las relaciones entre complejidad del programa, medidas por métricas como la complejidad ciclomática (McCabe, 1976) y la mantenibilidad (Banker *et al*., 1993; Coleman *et al*., 1994; Kafura y Reddy, 1987; Kozlov *et al*., 2008). No debería sorprender que dichos estudios descubrieran que, cuanto más complejo sea un sistema o componente, más costoso será darle mantenimiento. Las mediciones de complejidad son muy útiles para identificar componentes de programa que suelen ser costosos de mantener. Kafura y Reddy (1987) examinaron algunos componentes del sistema y descubrieron que el esfuerzo de mantenimiento tendía a enfocarse en un pequeño número de componentes complejos. Por ello, para reducir los costos de mantenimiento, se debe tratar de sustituir los componentes complejos del sistema con alternativas más sencillas.

Después de poner en servicio un sistema, se deben usar datos de proceso para auxiliarse a predecir la mantenibilidad. Los siguientes son ejemplos de métricas de proceso que sirven para valorar la mantenibilidad:

1. *Número de peticiones para mantenimiento correctivo* Un aumento en el número de reportes de bugs y fallas indicaría que se introdujeron más errores en el programa de los que se repararon durante el proceso de mantenimiento. Esto podría revelar un declive en la mantenibilidad.
2. *Tiempo promedio requerido para análisis del impacto* Refleja el número de componentes de programa que se ven afectados por la petición de cambio. Si este tiempo aumenta, implica que más componentes resultaron afectados y que la mantenibilidad decrece.
3. *Tiempo promedio tomado para implementar una petición de cambio* Éste no es el mismo que el tiempo para el análisis del impacto, aunque puede correlacionarse con él, sino más bien es la cantidad de tiempo que se necesita para modificar el sistema y su documentación, después de valorar cuáles componentes serán afectados. Un aumento en el tiempo necesario para implementar un cambio puede indicar un declive en la mantenibilidad.
4. *Número de peticiones de cambio pendientes* Con el tiempo, un aumento en este número implicaría un declive en la mantenibilidad.

La información predicha sobre las peticiones de cambio y las predicciones acerca de la mantenibilidad del sistema se usan para predecir los costos de mantenimiento. La mayoría de los gerentes combinan esta información con la intuición y la experiencia para estimar costos. El modelo COCOMO 2 de estimación de costos (Boehm *et al*., 2000), que se estudia en el capítulo 24, sugiere que una estimación del esfuerzo de mantenimiento del software puede basarse en el esfuerzo por comprender el código existente, así como en el esfuerzo para desarrollar el nuevo código.

9.3.2 Reingeniería de software

Como se estudió en la sección anterior, el proceso de evolución del sistema incluye comprender el programa que debe cambiarse y, luego, implementar dichos cambios. Sin embargo, muchos sistemas, especialmente los sistemas heredados más antiguos, son difíciles de entender y de cambiar. Es posible que los programas se hayan optimizado para rendimiento o utilización de espacio a expensas de la claridad o, con el tiempo, la estructura inicial del programa quizá se corrompió debido a una serie de cambios.

Para hacer que los sistemas de software heredados sean más sencillos de mantener, se pueden someter a reingeniería para mejorar su estructura y entendimiento. La reingeniería puede implicar volver a documentar el sistema, refactorizar su arquitectura, traducir los programas a un lenguaje de programación moderno, y modificar y actualizar la estructura y los valores de los datos del sistema. La funcionalidad del software no cambia y, normalmente, conviene tratar de evitar grandes cambios a la arquitectura de sistema.

Hay dos beneficios importantes de la reingeniería respecto de la sustitución:

1. *Reducción del riesgo* Hay un alto riesgo en el desarrollo de software empresarial crítico. Pueden cometerse errores en la especificación del sistema o tal vez haya problemas de desarrollo. Las demoras en la introducción del nuevo software podrían significar que la empresa está perdida y que se incurrirá en costos adicionales.

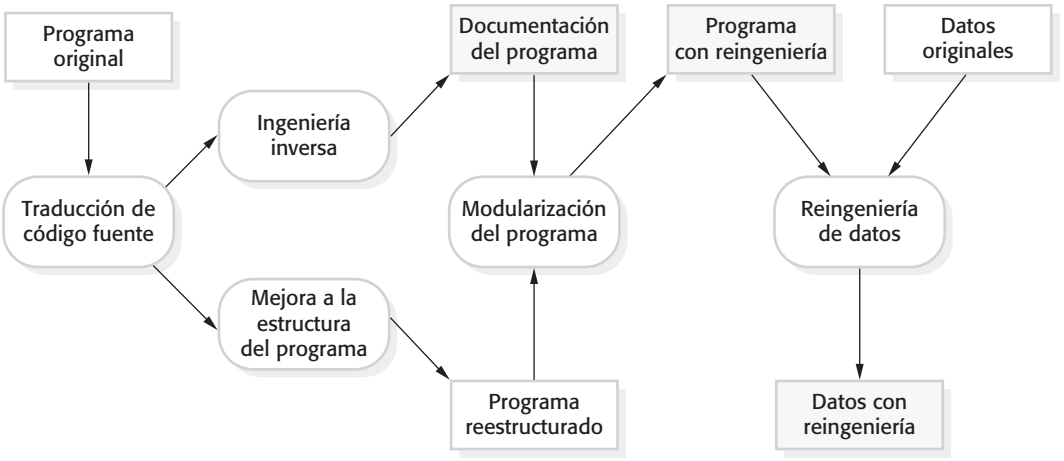


Figura 9.11 El proceso de reingeniería

2. *Reducción de costos* El costo de la reingeniería puede ser significativamente menor que el costo de desarrollar software nuevo. Ulrich (1990) cita un ejemplo de un sistema comercial para el cual los costos de reimplementación se estimaron en $50 millones. El sistema tuvo reingeniería exitosa por $12 millones. Con la tecnología de software moderna, el costo relativo de reimplementación quizá sea menor que esto, pero todavía superará considerablemente los costos de la reingeniería.

La figura 9.11 es un modelo general del proceso de reingeniería. La entrada al proceso es un sistema heredado, en tanto que la salida es una versión mejorada y reestructurada del mismo programa. Las actividades en este proceso de reingeniería son las siguientes:

1. *Traducción del código fuente* Con una herramienta de traducción, el programa se convierte de un lenguaje de programación antiguo a una versión más moderna del mismo lenguaje, o a un lenguaje diferente.
2. *Ingeniería inversa* El programa se analiza y se extrae información de él. Esto ayuda a documentar su organización y funcionalidad. De nuevo, este proceso es, por lo general, completamente automatizado.
3. *Mejoramiento de la estructura del programa* La estructura de control del programa se analiza y modifica para facilitar su lectura y comprensión, lo cual suele estar parcialmente automatizado, pero se requiere regularmente alguna intervención manual.
4. *Modularización del programa* Las partes relacionadas del programa se agrupan y, donde es adecuado, se elimina la redundancia. En algunos casos, esta etapa implicará refactorización arquitectónica (por ejemplo, un sistema que use muchos almacenes de datos diferentes puede refactorizarse para usar un solo depósito). Éste es un proceso manual.
5. *Reingeniería de datos* Los datos procesados por el programa cambian para reflejar cambios al programa. Esto puede significar la redefinición de los esquemas de bases de datos y convertir las bases de datos existentes a la nueva estructura. Por lo general,

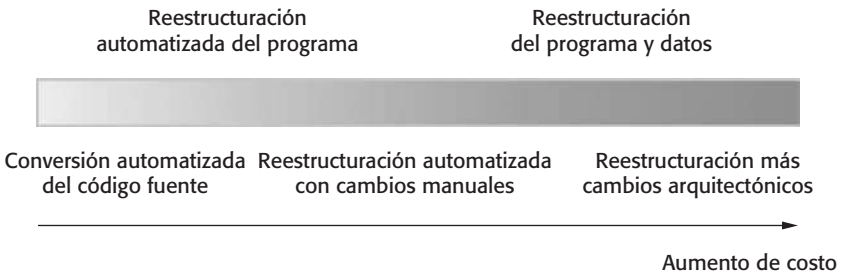


Figura 9.12 Enfoques de reingeniería

hay que limpiar los datos. Esto implica encontrar y corregir errores, eliminar registros duplicados, etcétera. Hay herramientas disponibles para auxiliar en la reingeniería de datos.

La reingeniería de programas no necesariamente requiere todos los pasos de la figura 9.11. No se necesita traducción del código fuente, si todavía se usa el lenguaje de programación de la aplicación. Si se logra hacer automáticamente toda la reingeniería, entonces quizá no sea necesaria la recuperación de documentación mediante ingeniería inversa. La reingeniería de datos sólo se requiere cuando las estructuras de datos en el programa cambian durante la reingeniería del sistema.

Para hacer que el sistema con reingeniería interopere con el nuevo software, tal vez se tengan que desarrollar servicios adaptadores, como se estudia en el capítulo 19. Éstos ocultan las interfaces originales del sistema de software y presentan las nuevas interfaces, mejor estructuradas, que pueden usarse con otros componentes. Este proceso de envolvimiento de los sistemas heredados es una importante técnica para desarrollar servicios reutilizables a gran escala.

Los costos de la reingeniería dependen evidentemente de la extensión del trabajo que se realiza. Existe un espectro de posibles enfoques a la reingeniería, como se muestra en la figura 9.12. Los costos aumentan de izquierda a derecha, de modo que la traducción del código fuente es la opción más barata. La reingeniería como parte de la migración arquitectónica es la más costosa.

El problema con la reingeniería de software es que existen límites prácticos a cuánto mejora un sistema gracias a la reingeniería. No es posible, por ejemplo, convertir un sistema escrito con un enfoque funcional, a un sistema orientado a objetos. Los grandes cambios arquitectónicos o la reorganización radical de la gestión de los datos del sistema no pueden realizarse automáticamente, de modo que son muy costosos. Aunque la reingeniería podría mejorar la mantenibilidad, el sistema con reingeniería probablemente no será tan mantenible como un sistema nuevo desarrollado usando modernos métodos de ingeniería de software.

9.3.3 Mantenimiento preventivo mediante refactorización

La refactorización es el proceso de hacer mejoras a un programa para frenar la degradación mediante el cambio (Opdyke y Johnson, 1990). Ello significa modificar un programa para mejorar su estructura, reducir su complejidad o hacerlo más fácil de entender. A veces se considera que la refactorización está limitada al desarrollo orientado a objetos, pero los principios son aplicables a cualquier enfoque de desarrollo. Mientras se refacto-

rice un programa, no se debe agregar funcionalidad, sino que hay que concentrarse en la mejora del programa. Por ende, se puede considerar la refactorización como el "mantenimiento preventivo" que reduce los problemas de cambios futuros.

Aunque la reingeniería y la refactorización tienen la intención de hacer el software más fácil de entender y cambiar, no son lo mismo. La reingeniería se lleva a cabo después de haber mantenido un sistema durante cierto tiempo y, por consiguiente, los costos de mantenimiento aumentan. Se usan herramientas automatizadas para procesar y someter a reingeniería un sistema heredado y así crear un nuevo sistema que sea más mantenible. La refactorización es un proceso continuo de mejoramiento debido al proceso de desarrollo y evolución. Tiene la intención de evitar la degradación de la estructura y el código que aumentan los costos y las dificultades por mantener un sistema.

La refactorización es una parte inherente de los métodos ágiles, como lo es la programación extrema, porque dichos métodos se basan en el cambio. En consecuencia, la calidad del programa es proclive a degradarse rápidamente, de modo que los desarrolladores ágiles con frecuencia refactorizan sus programas para evitar tal degradación. El énfasis en las pruebas de regresión en los métodos ágiles reduce el riesgo de introducir nuevos errores a través de la refactorización. Cualquier error que se introduzca debe ser detectable, ya que las pruebas anteriormente exitosas podrían fracasar. Sin embargo, la refactorización no depende de otras "actividades ágiles" y se utiliza con cualquier enfoque al desarrollo.

Fowler y sus colaboradores (1999) sugieren que existen situaciones estereotípicas (que se llaman "malos olores"), en las cuales el código de un programa es susceptible de mejorarse. Los ejemplos de malos olores que pueden mejorarse mediante refactorización incluyen:

1. *Código duplicado* El mismo de código muy similar puede incluirse en diferentes lugares de un programa. Éste se descarta o se implementa como un solo método o función que se llame cuando se requiera.
2. *Métodos largos* Si un método es demasiado largo, debe rediseñarse en varios métodos más cortos.
3. *Enunciados de switch (case)* Con frecuencia éstos implican duplicación, donde el cambio (*switch*) depende del tipo de algún valor. Los enunciados switch pueden dispersarse alrededor de un programa. En los lenguajes orientados a objetos, normalmente es posible usar un polimorfismo para lograr lo mismo.
4. *Aglomeración de datos* Las aglomeraciones de datos ocurren cuando el mismo grupo de objetos de datos (campos en clases, parámetros en métodos) vuelven a ocurrir en muchos lugares en un programa. Generalmente pueden sustituirse con un objeto que encapsule todos los datos.
5. *Generalidad especulativa* Esto ocurre cuando los desarrolladores incluyen generalidad en un programa, en caso de que se requiera en el futuro. Por lo general, esto simplemente puede eliminarse.

Fowler, en su libro y sitio Web, también sugiere algunas transformaciones primitivas de refactorización que pueden usarse de manera individual o en conjunto para lidiar con los malos olores. Los ejemplos de dichas transformaciones incluyen el método Extract (extraer), donde se eliminan duplicados y se crea un nuevo método; la expresión condicional Consolidate (consolidar), donde se sustituye una secuencia de pruebas con una sola prueba; y el método Pull up (subir), donde se sustituyen métodos similares en subclases con

un solo método en una superclase. Los entornos de desarrollo interactivo, como Eclipse, incluyen soporte para refactorización en sus editores. Esto facilita encontrar partes dependientes de un programa que deban cambiarse para implementar la refactorización.

La refactorización, realizada durante el desarrollo del programa, es una forma efectiva de reducir los costos a largo plazo en el mantenimiento de un programa. Sin embargo, si se toma la responsabilidad del mantenimiento de un programa, cuya estructura esté significativamente degradada, entonces sería casi imposible refactorizar sólo el código. Tal vez se tenga que considerar la refactorización del diseño, que probablemente resulte ser un problema más costoso y difícil. La refactorización del diseño implica identificar patrones de diseño relevantes (que se estudian en el capítulo 7) y sustituir el código existente con código que implemente dichos patrones de diseño (Kerievsky, 2004). Por cuestiones de espacio no se analiza esto aquí.

9.4 Administración de sistemas heredados

Para los sistemas de software nuevos desarrollados con modernos procesos de ingeniería de software, como el desarrollo incremental y el CBSE, es posible planear cómo integrar el desarrollo y la evolución del sistema. Cada vez con mayor frecuencia, las compañías empiezan a entender que el proceso de desarrollo del sistema es un proceso de todo el ciclo de vida y que no es útil una separación artificial entre el desarrollo del software y su mantenimiento. Sin embargo, todavía existen muchos sistemas heredados que son sistemas empresariales críticos. Éstos tienen que extenderse y adaptarse a las cambiantes prácticas del comercio electrónico.

La mayoría de las organizaciones, por lo general, tienen un portafolio de sistemas heredados, que se usan con un presupuesto limitado para mantenimiento y actualización. Deben decidir cómo obtener el mejor retorno de la inversión. Esto requiere hacer una valoración realista de sus sistemas heredados y, luego, decidir acerca de la estrategia más adecuada para hacer evolucionar dichos sistemas. Existen cuatro opciones estratégicas:

1. *Desechar completamente el sistema* Esta opción debe elegirse cuando el sistema no vaya a realizar una aportación efectiva a los procesos empresariales. Por lo general, esto ocurre cuando los procesos empresariales cambiaron desde la instalación del sistema y no se apoyan más en el sistema heredado.
2. *Dejar sin cambios el sistema y continuar el mantenimiento regular* Esta opción debe elegirse cuando el sistema todavía se requiera, pero sea bastante estable y los usuarios del sistema hagan relativamente pocas peticiones de cambio.
3. *Someter el sistema a reingeniería para mejorar su mantenibilidad* Esta opción debe elegirse cuando la calidad del sistema se haya degradado por el cambio y todavía se propone un nuevo cambio al sistema. Este proceso podría incluir el desarrollo de nuevos componentes de interfaz, de modo que el sistema original logre trabajar con otros sistemas más recientes.
4. *Sustituir todo o parte del sistema con un nuevo sistema* Esta opción tiene que elegirse cuando factores como hardware nuevo signifiquen que el viejo sistema no pueda continuar en operación o donde sistemas comerciales (*off-the-shelf*) permitirían al nuevo sistema desarrollarse a un costo razonable. En muchos casos se adopta

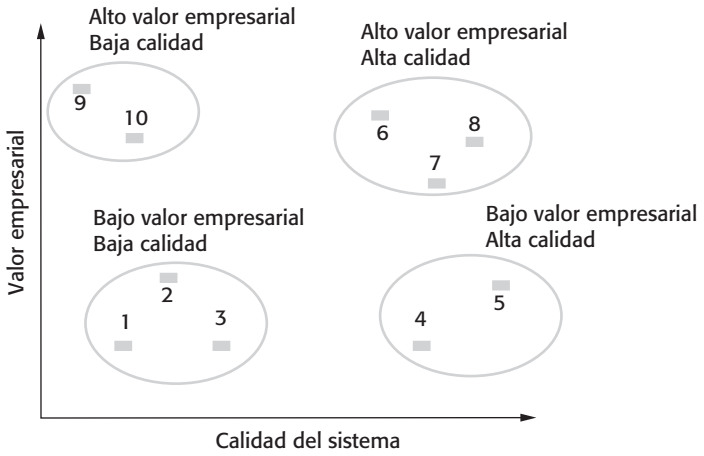


Figura 9.13 Ejemplo de valoración de un sistema heredado

una estrategia de sustitución evolutiva, en la cual grandes componentes del sistema se sustituyen con sistemas comerciales, y otros componentes se reutilizan siempre que sea posible.

Naturalmente, tales opciones no son excluyentes. Cuando un sistema está compuesto por muchos programas, es posible aplicar diferentes opciones a cada programa.

Cuando se valora un sistema heredado, tiene que observarse desde las perspectivas empresarial y técnica (Warren, 1998). Desde la perspectiva empresarial, se debe decidir si la compañía necesita realmente o no el sistema. Desde una perspectiva técnica, hay que valorar la calidad del software de aplicación, así como el hardware y software de soporte del sistema. Luego, se usa una combinación del valor empresarial y la calidad del sistema, para informar la decisión acerca de qué hacer con el sistema heredado.

Por ejemplo, suponga que una organización posee 10 sistemas heredados. Tiene que valorar la calidad y el valor empresarial de cada uno de dichos sistemas. Entonces se crea una gráfica que muestre el valor relativo empresarial y la calidad del sistema. Esto se ilustra en la figura 9.13.

En la figura 9.13 se observa que existen cuatro grupos de sistemas:

1. *Baja calidad, bajo valor empresarial* Mantener estos sistemas en operación será costoso y la tasa de retorno para la empresa será bastante pequeña. Estos sistemas deben descartarse.

2. *Baja calidad, alto valor empresarial* Estos sistemas realizan una importante aportación empresarial, de modo que no se pueden desechar. Sin embargo, su baja calidad significa que su mantenimiento resulta costoso. Dichos sistemas tienen que someterse a reingeniería para mejorar su calidad. Pueden sustituirse, si está disponible un sistema comercial adecuado.

3. *Alta calidad, bajo valor empresarial* Estos sistemas no aportan mucho a la empresa, pero su mantenimiento quizá no sea muy costoso. No vale la pena sustituir tales sistemas, así que puede continuarse el mantenimiento normal del sistema, si no se requieren cambios costosos y el hardware del sistema sigue en uso. Si los cambios costosos son necesarios, entonces el software puede desecharse.

4. *Alta calidad, alto valor empresarial* Estos sistemas deben mantenerse en operación. Sin embargo, su alta calidad significa que no se tiene que invertir en transformación ni sustitución del sistema. Hay que continuar con el mantenimiento normal del sistema.

Para calcular el valor empresarial de un sistema, se tiene que identificar a los participantes del sistema, como sus usuarios finales y sus administradores, y plantear una serie de preguntas acerca del sistema. Existen cuatro temas básicos que se deben analizar:

1. *El uso del sistema* Si los sistemas sólo se usan ocasionalmente o por un número pequeño de individuos, quizá tengan un bajo valor empresarial. En ocasiones un sistema heredado se desarrolló para satisfacer una necesidad empresarial que cambió o que ahora, de manera más efectiva, se satisface de otras formas. Sin embargo, se debe tener cuidado acerca del uso ocasional pero, a la vez, importante de los sistemas. Por ejemplo, en una universidad, un sistema de registro de estudiantes quizá sólo se utilice al comienzo de cada año escolar. Sin embargo, es un sistema esencial con un alto valor empresarial.
2. *Los procesos empresariales que se mantienen* Cuando se introduce un sistema, los procesos empresariales se diseñan para explotar las capacidades del sistema. Si el sistema es inflexible, sería casi imposible modificar dichos procesos empresariales. Sin embargo, conforme cambia el entorno, los procesos empresariales originales suelen volverse obsoletos. Por lo tanto, un sistema puede tener un bajo valor empresarial porque fuerza el uso de procesos empresariales ineficientes.
3. *Confiabilidad del sistema* La confiabilidad del sistema no sólo es un problema técnico, sino también un problema empresarial. Si un sistema no es confiable y los problemas afectan directamente a los clientes de la empresa o hacen que los trabajadores en la empresa se distraigan de otras tareas para resolver dichos problemas, el sistema tiene un valor empresarial bajo.
4. *Las salidas del sistema* Aquí el tema clave es la importancia de las salidas del sistema, para el funcionamiento exitoso de la empresa. Si la empresa depende de dichas salidas, entonces el sistema tiene un alto valor empresarial. Por el contrario, si tales salidas pueden generarse fácilmente en alguna otra forma, o si el sistema produce salidas que rara vez se utilizan, entonces su valor empresarial suele ser bajo.

Por ejemplo, suponga que una compañía ofrece un sistema de pedidos de viajes que usa el personal responsable de arreglar los viajes. Ellos pueden colocar los pedidos con un agente de viajes aprobado. Luego, los boletos se entregan y la compañía recibe una factura por ellos. Sin embargo, una evaluación del valor empresarial podría revelar que este sistema sólo se usa para colocar un porcentaje muy pequeño de pedidos de viaje. Las personas que hacen los arreglos de viaje descubren que es más barato y más conveniente tratar directamente con los proveedores de viajes mediante sus sitios Web. Este sistema todavía puede usarse, pero no hay razón real para conservarlo. La misma funcionalidad está disponible por parte de sistemas externos.

Por otro lado, suponga que una compañía desarrolló un sistema que lleva un seguimiento de todos los pedidos previos del cliente y, automáticamente, genera un recordatorio para que los clientes vuelvan a solicitar bienes. Esto da como resultado un gran número de pedidos repetidos y mantiene a los clientes satisfechos porque sienten que su

Factor	Preguntas
Estabilidad del proveedor	¿El proveedor todavía está en operación? ¿El proveedor es financieramente estable y es probable que continúe en operación? Si el proveedor ya no está en el negocio, ¿alguien más mantiene los sistemas?
Tasa de falla	¿El hardware tiene una alta tasa de fallas reportadas? ¿El software de soporte se cae y fuerza el reinicio del sistema?
Edad	¿Qué antigüedad tienen el hardware y el software? Cuanto más viejos sean el hardware y el software de soporte, más obsoletos serán. Quizá todavía funcionen correctamente, pero podría haber beneficios económicos y empresariales significativos al moverse hacia un sistema más moderno.
Rendimiento	¿El rendimiento del sistema es adecuado? ¿Los problemas de rendimiento tienen un efecto relevante sobre los usuarios del sistema?
Requerimientos de soporte	¿Qué apoyo local requieren el hardware y el software? Si existen altos costos asociados con este apoyo, valdría la pena considerar la sustitución del sistema.
Costos de mantenimiento	¿Cuáles son los costos del mantenimiento de hardware y de las licencias del software de soporte? El hardware más antiguo puede tener costos de mantenimiento más altos que los sistemas modernos. El software de soporte quizá tenga costos altos por licencia anual.
Interoperabilidad	¿Hay problemas de interfaz entre el sistema y otros sistemas? ¿Se pueden usar los compiladores, por ejemplo, con las versiones actuales del sistema operativo? ¿Se requiere emulación de hardware?

Figura 9.14 Factores usados en la valoración del entorno

proveedor está al tanto de sus necesidades. Las salidas de tal sistema son muy importantes para la empresa y, en consecuencia, este sistema tiene un alto valor empresarial.

Para valorar un sistema de software desde una perspectiva técnica, se necesita considerar tanto el sistema de aplicación en sí como el entorno donde opera el sistema. El entorno incluye el hardware y todo el software de soporte asociado (compiladores, entornos de desarrollo, etcétera) que se requieran para mantener el sistema. El entorno es importante porque muchos cambios del sistema resultan de cambios al entorno, como actualizaciones al hardware o al sistema operativo.

Si es posible, en el proceso de valoración ambiental se deben hacer mediciones del sistema y sus procesos de mantenimiento. Los ejemplos de datos que suelen ser útiles incluyen los costos de mantenimiento del hardware del sistema y del software de soporte, el número de fallas de hardware que ocurren durante algún periodo, y la frecuencia con la que se aplican "parches" y correcciones al software de soporte del sistema.

Los factores que se deben considerar durante la valoración del entorno se muestran en la figura 9.14. Observe que no son todas las características técnicas del entorno. También se debe considerar la fiabilidad de los proveedores del hardware y del software de soporte. Si dichos proveedores ya no están en el negocio, tal vez ya no haya soporte para sus sistemas.

Para valorar la calidad técnica de un sistema de aplicación, quizá se deban valorar diversos factores (figura 9.15) que se relacionan principalmente con la confiabilidad del sistema, las dificultades de mantener el sistema y su documentación. También se pueden

Factor	Preguntas
Entendimiento	¿Cuán difícil es entender el código fuente del sistema actual? ¿Cuán complejas son las estructuras de control que se utilizan? ¿Las variables tienen nombres significativos que reflejan su función?
Documentación	¿Qué documentación del sistema está disponible? ¿La documentación está completa, y es consistente y actualizada?
Datos	¿Existe algún modelo de datos explícito para el sistema? ¿En qué medida los datos se duplican a través de los archivos? ¿Los datos usados por el sistema están actualizados y son consistentes?
Rendimiento	¿El rendimiento de la aplicación es adecuado? ¿Los problemas de rendimiento tienen un efecto significativo sobre los usuarios del sistema?
Lenguaje de programación	¿Hay compiladores modernos disponibles para el lenguaje de programación usado para desarrollar el sistema? ¿El lenguaje de programación todavía se usa para el desarrollo de nuevos sistemas?
Administración de la configuración	¿Todas las versiones de la totalidad de las partes del sistema se administran mediante un sistema de administración de la configuración? ¿Existe una descripción explícita de las versiones de componentes que se usan en el sistema actual?
Datos de prueba	¿Existen datos de prueba para el sistema? ¿Hay un registro de pruebas de regresión realizadas cuando se agregaron nuevas características al sistema?
Habilidades del personal	¿Hay personal disponible que tenga las habilidades para mantener la aplicación? ¿Hay personal disponible que tenga experiencia con el sistema?

Figura 9.15 Factores usados en la valoración de la aplicación

recolectar datos que ayudarán a juzgar la calidad del sistema. Los datos que podrían ser útiles en la valoración de la calidad son:

1. *El número de peticiones de cambio del sistema* Los cambios al sistema, por lo general, corrompen la estructura del sistema y dificultan cambios futuros. Cuanto más alto sea este valor acumulado, más baja será la calidad del sistema.
2. *El número de interfaces de usuario* Éste es un factor importante en los sistemas basados en formas, donde cada forma puede considerarse como una interfaz de usuario separada. Cuanto más interfaces haya, más probabilidad habrá de que existan inconsistencias y redundancias en dichas interfaces.
3. *El volumen de datos usados por el sistema* Cuanto más alto sea el volumen de datos (número de archivos, tamaño de base de datos, etcétera), más probable será que haya inconsistencias de datos que reduzcan la calidad del sistema.

Idealmente, debe usarse una valoración objetiva para informar las decisiones acerca de qué hacer con un sistema heredado. Sin embargo, en muchos casos, las decisiones en realidad no son objetivas, sino que se basan en consideraciones políticas u organizacionales. Por ejemplo, si se fusionan dos empresas, el socio políticamente más poderoso por lo general conservará sus sistemas y desechará los demás. Si el gerente ejecutivo en una

organización decide moverse hacia una nueva plataforma de hardware, entonces esto puede requerir la sustitución de aplicaciones. Si no hay presupuesto disponible para la transformación del sistema en un año en particular, entonces puede continuar el mantenimiento del sistema, aunque esto dará como resultado costos más altos a largo plazo.

PUNTOS CLAVE

- El desarrollo y la evolución del software pueden considerarse como un proceso integrado e iterativo que se representa usando un modelo en espiral.
- Para sistemas personalizados, por lo general, los costos del mantenimiento de software superan a los de desarrollo.
- El proceso de evolución del software es conducido por peticiones de cambios e incluye análisis del impacto del cambio, planeación de las versiones e implementación del cambio.
- Las leyes de Lehman, como la noción de que el cambio es continuo, describen algunas percepciones derivadas de estudios a largo plazo de la evolución del sistema.
- Existen tres tipos de mantenimiento de software: reparación de bugs, modificación del software para trabajar en un nuevo entorno e implementación de requerimientos nuevos o diferentes.
- La reingeniería de software trata la reestructuración y la redocumentación de software para hacerlo más fácil de entender y cambiar.
- La refactorización, que hace pequeños cambios de programa sin alterar su funcionalidad, podría considerarse como mantenimiento preventivo.
- El valor empresarial de un sistema heredado y la calidad del software de aplicación y su entorno deben valorarse para determinar si el sistema tiene que sustituirse, transformarse o mantenerse.

LECTURAS SUGERIDAS

"Software Maintenance and Evolution: A Roadmap". Además de examinar los retos de la investigación, este ensayo es un adecuado y breve panorama del mantenimiento y la evolución del software por parte de investigadores líderes en esta área. Los problemas de investigación que identificaron todavía no se resuelven. (V. Rajlich y K.H. Bennett, *Proc. 20th Int. Conf. on Software Engineering*, IEEE Press, 2000.) http://doi.acm.org/10.1145/336512.336534.

Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Este excelente libro toca temas generales de mantenimiento y evolución del software, así como la migración de sistemas heredados. El libro se basa en un estudio de caso amplio de la transformación de un sistema COBOL a un sistema cliente-servidor basado en Java. (R. C. Seacord, D. Plakosh y G. A. Lewis, Addison-Wesley, 2003.)

Working Effectively with Legacy Code. Consejo prácticos sólidos acerca de los problemas y dificultades de lidiar con sistemas heredados. (M. Feathers, John Wiley & Sons, 2004.)

EJERCICIOS

9.1. Explique por qué un sistema de software que se usa en un entorno real debe cambiar o volverse progresivamente menos útil.

9.2. Exponga las razones subyacentes a las leyes de Lehman. ¿En qué circunstancias pueden fallar estas leyes?

9.3. A partir de la figura 9.4, observe que el análisis de impacto es un subproceso importante en el proceso de evolución del software. Con un diagrama, sugiera qué actividades habría en el análisis de impacto del cambio.

9.4. Como gerente de proyecto de software, en una compañía que se especializa en el desarrollo de software para la industria petrolera en el extranjero, se le asigna la tarea de descubrir los factores que afectan la mantenibilidad de los sistemas desarrollados por su compañía. Sugiera cómo configurar un programa para analizar el proceso de mantenimiento y descubrir métricas de mantenibilidad adecuadas para su compañía.

9.5. Describa brevemente los tres tipos principales de mantenimiento del software. ¿Por qué en ocasiones es difícil diferenciarlos?

9.6. ¿Cuáles son los principales factores que afectan los costos de la reingeniería de sistemas?

9.7. ¿En qué circunstancias una organización puede decidir desechar un sistema, cuando la valoración del sistema sugiere que es de alta calidad y de alto valor empresarial?

9.8. ¿Cuáles son las opciones estratégicas para la evolución de sistemas heredados? ¿En que situaciones sustituiría parte o todo un sistema en vez de continuar el mantenimiento del software?

9.9. Explique por qué los problemas con el software de soporte significarían que una organización tiene que sustituir sus sistemas heredados.

9.10. ¿Los ingenieros de software tienen responsabilidad profesional para producir código que pueda mantenerse y cambiarse, incluso si su empleador no lo solicita de manera explícita?

REFERENCIAS

Arthur, L. J. (1988). *Software Evolution*. New York: John Wiley & Sons.

Banker, R. D., Datar, S. M., Kemerer, C. F. y Zweig, D. (1993). "Software Complexity and Maintenance Costs". *Comm. ACM*, **36** (11), 81–94.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. y Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.

Coleman, D., Ash, D., Lowther, B. y Oman, P. (1994). "Using Metrics to Evaluate Software System Maintainability". *IEEE Computer*, **27** (8), 44–49.

Erlikh, L. (2000). "Leveraging legacy system dollars for E-business". *IT Professional*, **2** (3), mayo/junio 2000, 17–23.

Fowler, M., Beck, K., Brant, J., Opdyke, W. y Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Guimaraes, T. (1983). "Managing Application Program Maintenance Expenditures". *Comm. ACM*, **26** (10), 739–46.

Hopkins, R. y Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.

Kafura, D. y Reddy, G. R. (1987). "The use of software complexity metrics in software maintenance". *IEEE Trans. on Software Engineering*, **SE-13** (3), 335–43.

Kerievsky, J. (2004). *Refactoring to Patterns*. Boston: Addison Wesley.

Kozlov, D., Koskinen, J., Sakkinen, M. y Markkula, J. (2008). "Assessing maintainability change over multiple software releases". *J. of Software Maintenance and Evolution*, **20** (1), 31–58.

Krogstie, J., Jahr, A. y Sjoberg, D. I. K. (2005). "A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation". *Information and Software Technology*, **48** (11), 993–1005.

Lehman, M. M. (1996). "Laws of Software Evolution Revisited". Proc. European Workshop on Software Process Technology (EWSPT'96), Springer-Verlag. 108–24.

Lehman, M. M. y Belady, L. (1985). *Program Evolution: Processes of Software Change*. London: Academic Press.

Lehman, M. M., Perry, D. E. y Ramil, J. F. (1998). "On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution". Proc. Metrics '98, Bethesda. Maryland: IEEE Computer Society Press. 84–8.

Lehman, M. M., Ramil, J. F. y Sandler, U. (2001). "An Approach to Modelling Long-term Growth Trends in Software Systems". Proc. Int. Conf. on Software Maintenance, Florencia, Italia: 219–28.

Lientz, B. P. y Swanson, E. B. (1980). *Software Maintenance Management*. Reading, Mass.: Addison-Wesley.

McCabe, T. J. (1976). "A complexity measure". *IEEE Trans. on Software Engineering*., **SE-2** (4), 308–20.

Nosek, J. T. y Palvia, P. (1990). "Software maintenance management: changes in the last decade". *Software Maintenance: Research and Practice*, **2** (3), 157–74.

Opdyke, W. F. y Johnson, R. E. (1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA '90), Poughkeepsie, New York.

Poole, C. y Huisman, J. W. (2001). "Using Extreme Programming in a Maintenance Environment". *IEEE Software*, **18** (6), 42–50.

Rajlich, V. T. y Bennett, K. H. (2000). "A Staged Model for the Software Life Cycle". *IEEE Computer*, **33** (7), 66–71.

Sousa, M. J. (1998). "A Survey on the Software Maintenance Process". 14th IEEE International Conference on Software Maintenance (ICSM '98), Washington, D.C.: 265–74.

Ulrich, W. M. (1990). "The Evolutionary Growth of Software Reengineering and the Decade Ahead". *American Programmer*, **3** (10), 14–20.

Warren, I. E. (1998). *The Renaissance of Legacy Systems*. Londres: Springer.

PARTE 2

Confiabilidad y seguridad

A medida que los sistemas de software crecen en tamaño y complejidad, se cree firmemente que el reto más significativo que enfrenta la ingeniería de software consiste en garantizar la seguridad en dichos sistemas. Para confiar en un sistema se debe tener certeza de que estará disponible cuando se requiera y que rendirá lo esperado. Tiene que ser confiable para que las computadoras y los datos no estén amenazados por él. Ello significa que los temas de confiabilidad y seguridad del sistema con frecuencia son más importantes que los detalles de la funcionalidad del mismo. Por consiguiente, esta parte del libro se diseñó para introducir a los estudiantes y profesionales de la ingeniería de software en los importantes temas de confiabilidad y seguridad.

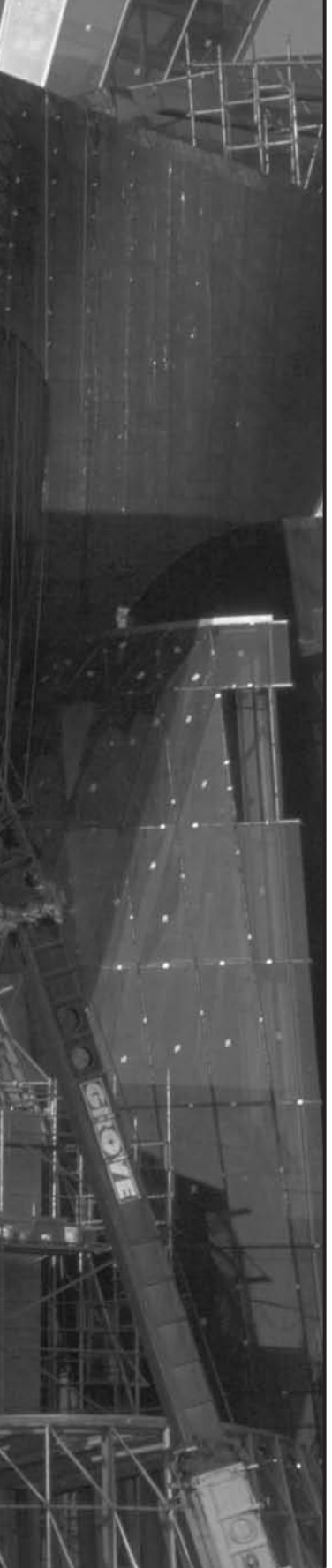
El primer capítulo de esta sección, capítulo 10, se ocupa de los sistemas sociotécnicos que, a primera vista, parecerían no tener mucho que ver con la confiabilidad del software. Sin embargo, muchas fallas de seguridad y confiabilidad surgen por causas humanas y de organización, que no se pueden ignorar al considerar la confiabilidad y la seguridad del sistema. Los ingenieros de software deben estar al tanto de esto y rechazar la idea de que mejores técnicas y tecnología pueden garantizar que los sistemas sean completamente confiables y seguros.

El capítulo 11 introduce los conceptos básicos de confiabilidad y seguridad, y explica los principios fundamentales de evasión, detección y

recuperación usados para construir sistemas confiables. El capítulo 12 complementa el capítulo 4, que cubre la ingeniería de requerimientos, con un análisis de los enfoques específicos que se utilizan para derivar y especificar requerimientos de seguridad y confiabilidad del sistema. En el capítulo 12 se accede brevemente al uso de la especificación formal; asimismo, en la Web está disponible un capítulo adicional sobre este tema.

Los capítulos 13 y 14 tratan las técnicas de la ingeniería de software para el desarrollo de sistemas confiables y seguros. Aun cuando las ingenierías de la confiabilidad y de la seguridad se tratan por separado, ambas tienen mucho en común. También se estudia la importancia de las arquitecturas de software, y se presentan lineamientos de diseño y técnicas de programación que ayudan a lograr la confiabilidad y seguridad. Se explica por qué es importante usar redundancia y diversidad, para asegurar que los sistemas sean capaces de enfrentar fallas operacionales y ataques externos. Se introduce al cada vez más importante tema de la supervivencia o resiliencia del software, que permite a los sistemas continuar con la entrega de servicios esenciales, aunque su seguridad se vea amenazada.

Finalmente, en esta sección, el capítulo 15 trata sobre la garantía de confiabilidad y seguridad. Se explica el uso del análisis estático y la comprobación del modelo para la verificación del sistema y la detección de fallas en el desarrollo. Dichas técnicas se han empleado con éxito en la ingeniería de sistemas críticos. También se analizan los enfoques específicos para probar la confiabilidad y seguridad de los sistemas, y exponer por qué sería necesario un caso de confiabilidad para convencer a un supervisor externo de que un sistema es seguro y confiable.



10

Sistemas sociotécnicos

Objetivos

Los objetivos de este capítulo son introducirlo al concepto de sistema sociotécnico (un sistema que incluye personal, software y hardware), así como mostrarle la necesidad de tener una perspectiva de los sistemas sobre seguridad y confiabilidad. Al estudiar este capítulo:

- conocerá qué se entiende por sistema sociotécnico, y entenderá la diferencia entre un sistema técnico basado en computadora y un sistema sociotécnico;
- se introducirá al concepto de propiedades de sistema emergente, como fiabilidad, rendimiento, seguridad y protección;
- identificará las actividades de procuración, desarrollo y operación que intervienen en el proceso de ingeniería de sistemas;
- sabrá por qué la confiabilidad y la seguridad del software no deben considerarse por separado y cómo éstas resultan afectadas por conflictos de sistemas, como errores del operador.

Contenido

10.1 Sistemas complejos
10.2 Ingeniería de sistemas
10.3 Procuración del sistema
10.4 Desarrollo del sistema
10.5 Operación del sistema

En un sistema de cómputo, el software y el hardware son interdependientes. Sin hardware, un sistema de software sería una abstracción, es decir, sería simplemente una representación o idea de cierto conocimiento humano. Mientras que sin el software, el hardware es sólo un conjunto de dispositivos electrónicos inertes. No obstante, si ambos se juntan para formar un sistema, entonces se crea una máquina capaz de realizar cálculos complejos y transferir los resultados de dichos cálculos a su entorno.

Esto ilustra una de las características fundamentales de un sistema: es más que la suma de sus partes. Los sistemas tienen propiedades que sólo se vuelven evidentes cuando sus componentes se integran y operan en conjunto. Por consiguiente, la ingeniería de software no es una actividad independiente, sino una parte intrínseca más general de procesos de ingeniería de sistemas. Los sistemas de software no son sistemas aislados, sino componentes esenciales de sistemas más extensos que tienen cierto propósito humano, social u organizacional.

Por ejemplo, el software del sistema meteorológico a campo abierto controla los instrumentos en una estación meteorológica. Se comunica con otros sistemas de software y forma parte de sistemas más amplios de predicción del clima, nacionales e internacionales. Además del hardware y el software, estos sistemas incluyen procesos para predecir el clima, y el personal que opera el sistema y analiza sus salidas. El sistema también abarca las organizaciones que dependen de él para ayudarlas a dar predicciones meteorológicas a individuos, gobiernos, industrias, etcétera. Estos sistemas más amplios en ocasiones se llaman sistemas sociotécnicos. Incluyen elementos no técnicos como individuos, procesos, regulaciones, etcétera, así como componentes técnicos, por ejemplo, computadoras, software y otro equipo.

Los sistemas sociotécnicos son tan complejos que es prácticamente imposible entenderlos como un todo. En vez de ello, deben verse como capas, como se muestra en la figura 10.1. Tales capas constituyen la columna de los sistemas sociotécnicos:

1. *La capa de equipo* Está compuesta de dispositivos de hardware, algunos de los cuales pueden ser computadoras.
2. *La capa del sistema operativo* Ésta interactúa con el hardware y ofrece un conjunto de facilidades comunes para capas de software superiores en el sistema.
3. *La capa de comunicaciones y gestión de datos* Extiende las facilidades del sistema operativo y ofrece una interfaz que permite la interacción con funcionalidad más extensa, como acceso a sistemas remotos, a una base de datos de un sistema, etcétera. En ocasiones, esto se llama *middleware*, pues se halla entre la aplicación y el sistema operativo.
4. *La capa de aplicaciones* Entrega la funcionalidad específica de la aplicación que se requiere. En esta capa puede haber varios programas de aplicación diferentes.
5. *La capa de proceso empresarial* En este nivel se definen y establecen los procesos empresariales de la organización que usan el software del sistema.
6. *La capa de la organización* Incluye procesos estratégicos de alto nivel, así como reglas, políticas y normas de la empresa que deben seguirse al usar el sistema.
7. *La capa social* En ella se definen las leyes y regulaciones de la sociedad que rigen la operación del sistema.

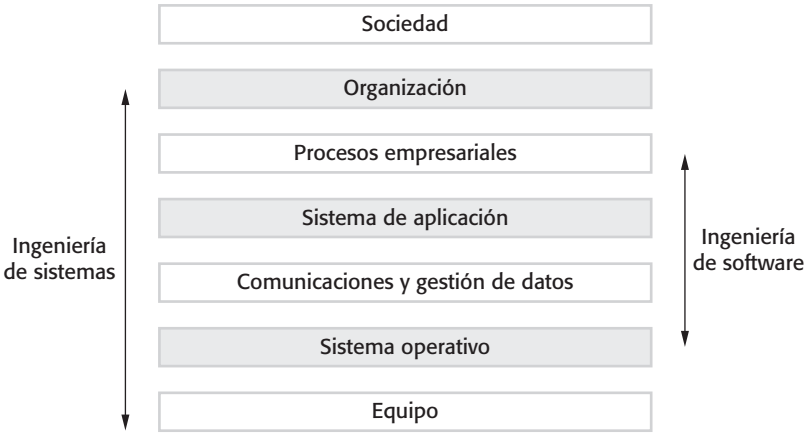


Figura 10.1 Capas de los sistemas sociotécnicos

En principio, la mayoría de las interacciones están entre capas contiguas, y cada capa oculta el detalle de la capa de abajo de la capa superior. En la práctica, éste es no siempre el caso. Entre las capas quizás haya interacciones inesperadas, lo cual derivaría en problemas para el sistema como un todo. Por ejemplo, imagine que hay un cambio en la ley que administra el acceso a la información personal. Esto proviene de la capa social. Ello conduce a nuevos procedimientos organizacionales y cambios para los procesos empresariales. Sin embargo, el sistema de aplicación quizá no sea capaz de brindar el nivel de privacidad requerido, de modo que deben implementarse cambios en la capa de comunicaciones y gestión de datos.

Deliberar de manera holística sobre los sistemas, en vez de considerar simplemente el software en aislamiento, es esencial cuando se considera la seguridad y confiabilidad del software. Las fallas de operación del software, en sí, rara vez tienen consecuencias serias, porque el software es intangible y, aun cuando se dañe, su restauración es fácil y económica. No obstante, cuando dichas fallas de operación de software caen en cascada por otras partes del sistema, afectan el entorno físico y humano del software. Aquí, las consecuencias de la falla de operación son más significativas. Es posible que las personas deban realizar trabajo extra para contener o recuperarse de la falla de operación; por ejemplo, pueden tener daños físicos en el equipo, pérdida o corrupción de datos, o fallas en la confidencialidad con consecuencias desconocidas.

Por lo tanto, hay que tomar una perspectiva de nivel del sistema cuando se diseñe software que tenga que ser seguro y confiable. Usted necesita entender las consecuencias de las fallas de operación del software para otros elementos en el sistema. También debe entender cómo esos otros elementos del sistema causan las fallas de operación del software, y cómo ayudarían a proteger al software y a recuperar éste contra tales fallas.

Por consiguiente, el problema existente es una falla de operación del sistema, más que del software. Esto significa que es preciso examinar cómo interactúa el software con su entorno inmediato para garantizar que:

1. Las fallas de operación del software estén, tanto como sea posible, contenidas dentro de las capas del sistema y no afecten seriamente la operación de capas adjuntas. En particular, las fallas de operación del software no deben conducir a fallas de operación del sistema.

2. Se entiende cómo las fallas de desarrollo y de operación en las capas, que no son del software, llegan a afectar al software. Usted también puede considerar cómo elaborar pruebas en el software con la finalidad de ayudar a detectar dichas fallas, y cómo dar soporte para recuperarse de las mismas.

Puesto que el software es inherentemente flexible, se deja por lo general a los ingenieros de software solucionar los problemas imprevistos del sistema. Suponga que una instalación de radar está ubicada de tal modo que pasan espectros en la imagen del radar. No es práctico mover el radar a un sitio con menos interferencia, así que los ingenieros de sistemas deben encontrar otra forma de eliminar tales espectros. La solución puede ser mejorar las capacidades de procesamiento de imágenes del software para eliminar los espectros de las imágenes. Esto quizás haga lento el software, de tal forma que su rendimiento se vuelve inadmisible. El problema se caracterizará entonces como una "falla de operación del software"; mientras que, de hecho, es una falla en el proceso de diseño para el sistema como un todo.

Es muy común este tipo de situación, donde los ingenieros de software tienen el problema de mejorar las capacidades del software, sin aumentar los costos del hardware. Muchas de las llamadas fallas de operación del software no son consecuencia de problemas inherentes de software, sino el resultado de buscar cambiar el software para instalar requerimientos modificados de ingeniería del sistema. Un buen ejemplo de esto fue la falla del sistema de equipaje del aeropuerto de Denver (Swartz, 1996), donde se esperaba que el software controlador lidiara con las limitaciones del equipo utilizado.

La ingeniería de sistemas (Stevens *et al*., 1998; Thayer, 2002; Thomé, 1993; White *et al*., 1993) es el proceso para diseñar sistemas completos, no sólo el software en estos sistemas. El software es el elemento controlador e integrador en estos sistemas, y los costos de la ingeniería de software con frecuencia son el principal componente de costo en los sistemas globales de costos. Como ingeniero de software, ayuda el hecho de tener un amplio conocimiento de cómo interactúa el software con otros sistemas de hardware y software, y cómo se supone que se usará. Este conocimiento es útil para comprender los límites del software, para diseñar mejor software y participar en un grupo de ingeniería de sistemas.

10.1 Sistemas complejos

El término "sistema" se usa de forma universal. Se habla de sistemas de cómputo, sistemas operativos, sistemas de pago, sistema educativo, sistema de gobierno y cosas por el estilo. Indiscutiblemente, se trata de usos muy diferentes de la palabra "sistema", aunque de algún modo comparten la característica de que el sistema es más que simplemente la suma de sus partes.

Los sistemas abstractos, como el sistema de gobierno, están fuera del ámbito de este libro. En cambio, el enfoque está en los sistemas que incluyen computadoras y que tienen algún propósito específico, como habilitar la comunicación, soportar la navegación o calcular los salarios. La siguiente es una definición de trabajo útil sobre estos tipos de sistemas:

> *Un sistema es una colección intencionada de componentes interrelacionados, de diferentes tipos, que trabajan en conjunto para lograr algún objetivo.*

Esta definición general abarca una amplia gama de sistemas; por ejemplo, un sistema simple, como un apuntador láser, que generalmente incluye algunos componentes de hardware, además de una pequeña cantidad de software de control. En contraste, un

sistema de control de tráfico incluye miles de componentes de hardware y software, aparte de usuarios que toman decisiones con base en información de dicho sistema de cómputo.

Una característica de todos los sistemas complejos es que las propiedades y el comportamiento de los componentes del sistema están estrechamente vinculados. El funcionamiento exitoso de cada componente del sistema depende del funcionamiento de los otros componentes. Por ende, el software sólo opera si el procesador está en funcionamiento. El procesador sólo realiza cálculos si el sistema de software —que define dichos cálculos— se instaló de manera exitosa.

Comúnmente, los sistemas complejos son jerárquicos y, por ende, incluyen otros sistemas. Por ejemplo, un sistema de comando y control policiaco incluye un sistema de información geográfica para proporcionar detalles de la ubicación de los incidentes. A tales sistemas incluidos se les conoce como "subsistemas", los cuales pueden operar por cuenta propia como sistemas independientes. Muestra de ello es que el mismo sistema de información geográfica puede ser utilizado en sistemas de control y orden de emergencias en logística de transporte.

Los sistemas que incluyen software se dividen en dos categorías:

1. *Sistemas técnicos basados en computadora* Se trata de sistemas que incluyen componentes de hardware y software, aunque no incluyen procedimientos y procesos. Los ejemplos de sistemas técnicos abarcan televisores, teléfonos móviles y otros equipos con software embebido. La mayoría del software para PC, juegos de computadora, etcétera, también se ubica en esta categoría. Los individuos y las organizaciones usan los sistemas técnicos para una finalidad específica, aunque el conocimiento de este propósito no forme parte del sistema. Por ejemplo, el procesador de texto que se usó para escribir este libro desconoce si se usó para escribir un libro.

2. *Sistemas sociotécnicos* Éstos incluyen uno o más sistemas técnicos, pero también incluyen individuos que entienden el propósito del sistema dentro del sistema en sí. Los sistemas sociotécnicos tienen procesos operacionales definidos y las personas (los operadores) son partes inherentes del sistema. Están administrados por políticas y reglas organizacionales, y podrían verse afectados por restricciones externas como leyes nacionales y políticas reguladoras. Por ejemplo, este libro se creó mediante un sistema de edición sociotécnico que incluye varios procesos y sistemas técnicos.

Los sistemas sociotécnicos son sistemas empresariales que intentan auxiliar para alcanzar una meta de negocio. Ésta puede ser incrementar las ventas, reducir el uso de material en la fabricación, recolectar impuestos, mantener un espacio aéreo seguro, etcétera. Puesto que están incrustados en un entorno organizacional, la procuración,* el desarrollo y el uso de dichos sistemas tienen influencia de las políticas y los procedimientos de la organización, así como de su cultura laboral. Los usuarios del sistema son individuos que están influidos por la forma en que se administra la organización, así como por sus interacciones con otras personas dentro y fuera de la organización.

Cuando se trata de desarrollar sistemas sociotécnicos, es necesario entender el entorno de la organización donde se utiliza. Si no se hace, los sistemas quizá no cubran las necesidades empresariales y, en consecuencia, los usuarios y sus administradores rechazarán el sistema.

---

*Nota del revisor técnico: En inglés, el término *procurement* (procuración) abarca las actividades de compra, elaboración del plan inicial de desarrollo y contratación para el desarrollo. Éste es el sentido del término procuración en este texto.

Los factores organizacionales del entorno del sistema que pueden afectar los requerimientos, el diseño y la operación de un sistema sociotécnico incluyen:

1. *Cambios de procesos* El sistema puede requerir cambios en los procesos de trabajo del entorno. Si es así, ciertamente se requerirá capacitación. Si los cambios son significativos, o si implican que la gente pierda su empleo, hay riesgo de que los usuarios se resistan a la introducción del sistema.
2. *Cambios laborales* Los nuevos sistemas pueden reemplazar las habilidades de los usuarios en un entorno, o bien, hacer que cambien la forma como trabajan. Si es así, los usuarios se opondrían activamente a la introducción del sistema en la organización. Los diseños que requieran que los directivos deban cambiar su forma de trabajar, para acoplarse a un nuevo sistema de cómputo, a menudo son molestos. Los directivos quizá crean que su estatus en la organización decrece por el sistema.
3. *Cambios en la organización* El sistema podría cambiar la estructura política de poder en una organización. Por ejemplo, si una organización depende de un sistema complejo, quienes controlan el acceso a dicho sistema tienen, por lo tanto, mayor poder político.

Los sistemas sociotécnicos poseen tres características que son particularmente importantes al considerar la seguridad y la confiabilidad:

1. Tienen propiedades emergentes que son propiedades del sistema como un todo, en vez de asociarse con partes individuales del sistema. Las propiedades emergentes dependen tanto de los componentes del sistema como de las relaciones entre ellos. Debido a esta complejidad, las propiedades emergentes sólo se evalúan cuando el sistema está ensamblado. Seguridad y confiabilidad son propiedades emergentes del sistema.
2. A menudo suelen ser no deterministas. Ello quiere decir que, cuando se les presenta una entrada específica, es posible que no produzcan siempre la misma salida. El comportamiento del sistema depende del personal que lo opera y los individuos no siempre reaccionan de la misma forma. Más aún, el uso del sistema puede crear nuevas relaciones entre los componentes del sistema y, por consiguiente, cambiar su comportamiento emergente. De ahí que los fallas de desarrollo y operación del sistema puedan ser transitorias, y las personas quizá no estén de acuerdo en si realmente ocurrió o no una falla de operación.
3. La amplitud con que el sistema apoya los objetivos de la organización no sólo depende del sistema en sí. También radica en la estabilidad de dichos objetivos, las relaciones y los conflictos entre los objetivos de la organización, y cómo la gente de la organización interpreta dichos objetivos. Una nueva administración podría reinterpretar los objetivos de la organización, a los cuales debería apoyar el sistema diseñado, de modo que un sistema "exitoso" se vería entonces como una "falla de operación".

Las consideraciones sociotécnicas son críticas por lo general en la determinación de si un sistema cumplió o no exitosamente con sus objetivos. Por desgracia, tomar en cuenta esto es muy difícil para los ingenieros con poca experiencia en estudios sociales o culturales. Para ayudar a entender los efectos de los sistemas sobre las organizaciones, se

Propiedad	Descripción
Volumen	El volumen de un sistema (el total de espacio ocupado) varía en función de cómo se ordenan y conectan los componentes ensamblados.
Fiabilidad	La fiabilidad del sistema depende de la fiabilidad del componente, aun cuando interacciones inesperadas lleguen a causar nuevos tipos de fallas de operación y, por consiguiente, afecten la fiabilidad del sistema.
Seguridad	La seguridad del sistema (capacidad para resistir ataques) es una propiedad compleja que no se mide con facilidad. Pueden concebirse ataques que no anticiparon los diseñadores del sistema y ello quizá logre vencer las protecciones internas.
Reparabilidad	Esta propiedad refleja qué tan fácilmente se corrige un problema con el sistema una vez descubierto. Depende de diagnosticar el problema, acceder a los componentes que están fallando y modificar o sustituir dichos componentes.
Usabilidad	Esta propiedad refleja la sencillez con la que se usa el sistema. Depende de los componentes técnicos del sistema, de sus operadores y de su entorno operacional.

Figura 10.2 Ejemplos de propiedades emergentes

desarrollaron varias metodologías, como las sociotécnicas de Mumford (1989) y la metodología de sistemas blandos de Checkland (1981; Checkland y Scholes, 1990). También hay estudios sociológicos acerca de los efectos de los sistemas basados en computadoras sobre el trabajo (Ackroyd *et al*., 1992; Anderson *et al*., 1989; Suchman, 1987).

10.1.1 Propiedades emergentes del sistema

Las relaciones complejas entre los componentes en un sistema significan que un sistema es más que simplemente la suma de sus partes. Tiene propiedades que son propiedades del sistema como un todo. Tales "propiedades emergentes" (Checkland, 1981) no pueden atribuirse a alguna parte específica del sistema. En cambio, sólo surgen una vez que se integran los componentes del sistema. Algunas de tales propiedades, como el peso, suelen derivarse directamente de propiedades comparables de subsistemas. Sin embargo, muy a menudo son resultado de interrelaciones complejas entre subsistemas. La propiedad del sistema no puede calcularse directamente a partir de las propiedades de los componentes individuales del mismo. En la figura 10.2 se muestran ejemplos de algunas propiedades emergentes.

Hay dos tipos de propiedades emergentes:

1. Propiedades emergentes funcionales, cuando el propósito de un sistema sólo surge después de integrar sus componentes. Por ejemplo, una bicicleta tiene la propiedad funcional de ser un dispositivo de transporte, una vez que se ensamblan sus componentes.
2. Propiedades emergentes no funcionales, que se relacionan con el comportamiento del sistema en su entorno operacional. Fiabilidad, rendimiento, seguridad y protección son ejemplos de propiedades emergentes. Éstas son críticas para los sistemas basados en computadora, pues las fallas para lograr un nivel mínimo definido en dichas propiedades suelen hacer inútil al sistema. Algunos usuarios tal vez no necesiten varias

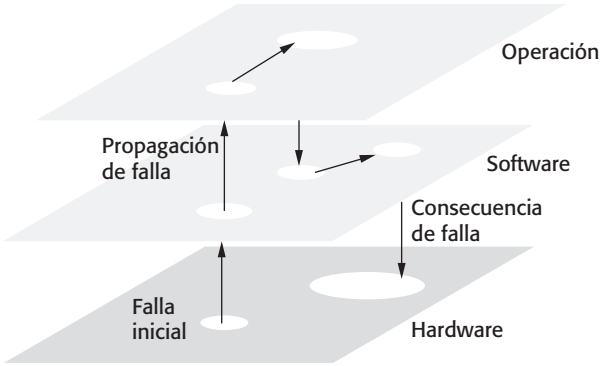


Figura 10.3 Propagación de falla

de las funciones del sistema, de modo que sin ellas el sistema también sería aceptable. Sin embargo, un sistema que no sea confiable o sea demasiado lento tiene amplias probabilidades de ser rechazado por todos los usuarios.

Las propiedades emergentes de confiabilidad, como la fiabilidad, dependen tanto de las propiedades de los componentes individuales como de sus interacciones. Los componentes en un sistema son interdependientes. Las fallas en un componente pueden propagarse a lo largo del sistema y afectar la operación de otros componentes. A pesar de ello, con frecuencia es difícil anticiparse a cómo las fallas en dichos componentes afectarán a otros componentes. Por ello, es prácticamente imposible estimar la fiabilidad global del sistema a partir de datos sobre la fiabilidad de los componentes del sistema.

En un sistema sociotécnico se necesita considerar la fiabilidad desde tres perspectivas:

1. *Fiabilidad del hardware* ¿Cuál es la probabilidad de que los componentes de hardware fallen y cuánto tiempo tardaría en repararse un componente averiado?
2. *Fiabilidad del software* ¿Cuál es la probabilidad de que un componente de software produzca una salida incorrecta? La falla del software es distinta de la falla del hardware, pues el software no se desgasta. Con frecuencia, las fallas de software son transitorias. El sistema sigue trabajando después de producir un resultado incorrecto.
3. *Fiabilidad del operador* ¿Cuán probable es que el operador de un sistema cometa un error y proporcione una entrada incorrecta? ¿Cuán probable es que el software falle para detectar este error y lo propague?

Las fiabilidades del hardware, software y el operador no son independientes. La figura 10.3 muestra cómo las fallas en un nivel podrían propagarse a otros niveles en el sistema. La falla en el hardware quizá genere señales falsas que están fuera del rango de las entradas esperadas por el software. Entonces, el software puede comportarse de manera impredecible y producir salidas imprevistas. Esto suele confundir y, en consecuencia, estresar al operador del sistema.

El error del operador es más probable cuando éste se siente estresado. De este modo, una falla de hardware significaría en tal caso que el operador del sistema cometerá

errores que, a la vez, conducirían a más problemas de software o de procesamiento adicional. Esto podría sobrecargar el hardware causando más fallas. Por lo tanto, la falla inicial, que podría ser recuperable, se convertiría rápidamente en un problema serio que dé como resultado una parálisis completa del sistema.

La fiabilidad de un sistema depende del contexto en que se use. Sin embargo, el entorno del sistema no se puede especificar por completo, ni los diseñadores del sistema pueden colocar restricciones en dicho entorno para sistemas operacionales. Los diferentes sistemas que operan dentro de un entorno tal vez reaccionen a los problemas de formas impredecibles, lo cual en consecuencia afectaría la fiabilidad de todos esos sistemas.

Por ejemplo, imagine que un sistema se diseña para operar a una temperatura ambiente normal. Para permitir variaciones y condiciones excepcionales, los componentes electrónicos de un sistema se diseñan para operar dentro de cierto intervalo de temperaturas, por ejemplo, de 0 a 45 grados. Fuera de este intervalo de temperatura, los componentes se comportarán en forma impredecible. Ahora suponga que este sistema se instala cerca del aire acondicionado. Si éste falla y arroja gas caliente sobre el dispositivo electrónico, entonces el sistema podría sobrecalentarse. Siendo así, los componentes y, en consecuencia, todo el sistema, fallarían.

Si este sistema se hubiera instalado en alguna otra parte de ese ambiente, tal problema no ocurriría. Cuando el aire acondicionado funciona adecuadamente no hay problemas. No obstante, debido a la cercanía física de dichas máquinas, existió entre ellas una relación no prevista que condujo a la falla del sistema.

Como la fiabilidad, las propiedades emergentes como rendimiento o usabilidad son difíciles de valorar, aunque pueden medirse después de poner en funcionamiento el sistema. Sin embargo, propiedades como la seguridad y la protección no son mensurables. Aquí, usted no está interesado simplemente por los atributos que se relacionan con el comportamiento del sistema, sino también con el comportamiento no deseado o inaceptable. Un sistema seguro es aquel que impide el acceso no autorizado a sus datos. A pesar de ello, resulta claramente imposible predecir todos los posibles modos de acceso y prohibirlos de forma explícita. Por consiguiente, sólo sería posible valorar dichas propiedades “no se debe”. Esto es, sólo se sabe que un sistema no es seguro cuando alguien trata de penetrarlo.

10.1.2 No determinismo

Un sistema determinista es uno completamente predecible. Si se ignoran los problemas de temporización, los sistemas de software que se ejecutan en un hardware confiable por completo y los que presentan una secuencia de entradas producirán siempre la misma secuencia de salidas. Por supuesto, no hay nada como un hardware completamente fiable; sin embargo, el hardware es por lo general bastante fiable para considerar a los sistemas de hardware como deterministas.

Por otro lado, los individuos no son deterministas. Cuando se les presenta exactamente la misma entrada (es decir, un requerimiento para completar una tarea), sus respuestas dependerán de sus estados emocional y físico, de la persona que hace la petición, de otros individuos en el entorno y de cualquier otra cuestión que estén haciendo. En algunas ocasiones, se sentirán satisfechos por hacer el trabajo, aunque en otras se negarán a realizarlo.

Por un lado, los sistemas sociotécnicos no son deterministas, ya que incluyen personas y, por otro, debido a que los cambios al hardware, al software y a los datos en dichos sistemas son muy frecuentes. Las interacciones entre tales cambios son complejas y, por lo tanto, el comportamiento del sistema se vuelve impredecible. Éste no es un problema en sí mismo, pero, desde una perspectiva de confiabilidad, sería difícil decidir si ocurrió o no una falla del sistema, así como estimar la periodicidad de las fallas del mismo.

Por ejemplo, suponga que a un sistema se le presenta un conjunto de 20 entradas de prueba. Éste procesa dichas entradas y registra los resultados. Algún tiempo después, se procesan las mismas 20 entradas de prueba y se comparan los efectos con los resultantes almacenados anteriormente. Cinco de ellos son diferentes. ¿Esto significa que existen cinco fallas? ¿O las diferencias simplemente son variaciones razonables en el comportamiento del sistema? Se descubrirá esto sólo al observar los resultados con más detenimiento y hacer juicios sobre la forma en que el sistema manejó cada entrada.

10.1.3 Criterios de éxito

Por lo general, los sistemas sociotécnicos complejos se desarrollan para enfrentar lo que en ocasiones se conoce como "problemas malvados" (Rittel y Webber, 1973). Un problema malvado es aquel que es tan complejo y que implica tantas entidades relacionadas que no hay especificación definitiva del problema. Diferentes participantes ven el problema de diversas formas y ninguna tiene una comprensión integral del problema como un todo. La verdadera naturaleza del problema sólo surge conforme se desarrolla una solución. Un ejemplo extremo de un problema malvado es la planeación en caso de terremotos. Nadie es capaz de predecir con exactitud dónde estará el epicentro de un sismo, a qué hora ocurrirá o qué efecto tendrá sobre el ambiente local. Es imposible especificar con detalle cómo hacer frente a un gran terremoto.

Esto dificulta la definición de los criterios de éxito para un sistema. ¿Cómo determinar si un nuevo sistema contribuye, con base en la forma en que fue planteado, con las metas empresariales de la organización que pagó por el sistema? El juicio de éxito no se hace por lo general con base en las razones originales al procurar y desarrollar el sistema. En vez de ello, se basa en si el sistema es efectivo o no lo es cuando se implementa. Puesto que el ambiente empresarial suele variar muy rápidamente, las metas empresariales quizá cambien de manera significativa durante el desarrollo del sistema.

La situación es incluso más compleja cuando existen múltiples metas en conflicto, que se interpretan de modo diferente por diversos participantes. Por ejemplo, el sistema en que se basa el MHC-PMS (estudiado en el capítulo 1) se diseñó para brindar apoyo a dos metas empresariales distintas:

1. Mejorar la calidad de atención para quienes padecen enfermedades mentales.
2. Aumentar el ingreso al ofrecer reportes detallados de la atención brindada y de los costos de esa atención.

Por desgracia, se demostró que éstas eran metas en conflicto porque la información requerida para satisfacer la meta de reportar significaba que los médicos y las enfermeras debían proporcionar información adicional, más allá de los registros sanitarios que generalmente se conservan. Esto reducía la calidad de atención a los pacientes, pues

significaba que el personal clínico tenía menos tiempo para hablar con ellos. Desde una perspectiva médica, este sistema no era una mejora sobre el sistema manual anterior, pero sí lo era desde una perspectiva administrativa.

La naturaleza de los atributos de seguridad y confiabilidad incluso hace en ocasiones más difícil decidir si un sistema es exitoso. La intención de un sistema nuevo será, digamos, mejorar la seguridad al sustituir un sistema existente con un entorno de datos más seguro. Suponga que, después de la instalación, el sistema es atacado, que sobreviene una brecha de seguridad y algunos datos se corrompen. ¿Esto significa que el sistema resultó un fracaso? Es imposible decirlo, porque no se conoce la dimensión de las pérdidas que ocurrirían con el sistema antiguo, dados los mismos ataques.

10.2 Ingeniería de sistemas

La ingeniería de sistemas abarca todas las actividades que hay en la procuración, la especificación, el diseño, la implementación, la validación, el despliegue, la operación y el mantenimiento de los sistemas sociotécnicos. Los ingenieros de sistemas no sólo se preocupan por el software, sino también por el hardware y las interacciones del sistema con los usuarios y su entorno. Ellos deben pensar en los servicios que ofrece el sistema, las restricciones con las cuales deben construir y operar el mismo, así como las formas en las cuales el sistema se usa para cumplir con su(s) propósito(s).

Existen tres etapas que se traslapan (figura 10.4) en la vida de los sistemas sociotécnicos grandes y complejos:

1. *Procuración o adquisición* Durante esta etapa se decide el propósito de un sistema; se establecen los requerimientos de alto nivel del sistema; se toman decisiones sobre cómo se distribuirá la funcionalidad a través del hardware, el software y el personal; y se adquieren los componentes que constituirán el sistema.
2. *Desarrollo* Durante esta etapa se diseña el sistema. Los procesos de desarrollo incluyen todas las actividades que intervienen en el desarrollo del sistema, como definición de requerimientos, diseño del sistema, ingeniería del hardware y el software, integración y pruebas del sistema. Se definen los procesos operacionales y se diseñan los cursos de capacitación para los usuarios del sistema.
3. *Operación* En esta etapa se implementa el sistema, se capacita a los usuarios y se pone en funcionamiento el sistema. Constantemente, tienen entonces que cambiar los procesos operacionales planeados para reflejar el entorno de trabajo real donde se usa el sistema. Con el tiempo, el sistema evoluciona a medida que se identifican nuevos requerimientos. A final de cuentas, el sistema declina en valor y se retira del servicio activo para reemplazarse.

Estas etapas no son independientes. Una vez que el sistema está en operación, quizá sea posible que se procure nuevo equipo y software para sustituir los componentes obsoletos del sistema, ofrecer nueva funcionalidad, o bien, hacer frente al aumento de la demanda. De igual modo, las peticiones de cambios durante la operación requieren más desarrollo del sistema.

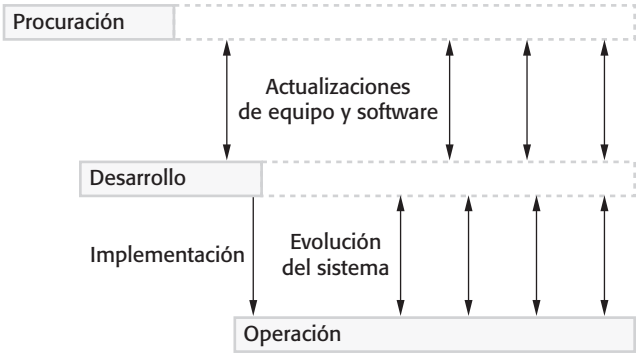


Figura 10.4 Etapas de ingeniería de sistemas

La seguridad y la confiabilidad global de un sistema tienen influencia en las actividades durante todas estas etapas. Las opciones de diseño pueden restringirse por decisiones de procuración en el ámbito del sistema, así como en el hardware y software. Quizá sea imposible implementar ciertos tipos de protecciones del sistema. Ello introduciría vulnerabilidades que podrían conducir a futuras fallas del sistema. Los errores humanos cometidos durante las fases de especificación, diseño y desarrollo significarían que se introducen fallas de desarrollo en el sistema. Las pruebas inadecuadas significarían que las fallas de desarrollo no se descubren antes de que se implemente un sistema. Durante la operación, los errores en la configuración del sistema para su implementación podrían conducir a más vulnerabilidades. Los operadores del sistema también pueden cometer errores en el uso del sistema. Quizá las suposiciones realizadas durante la procuración original se olviden cuando se hacen cambios al sistema y, de nueva cuenta, se podrían introducir vulnerabilidades al sistema.

Una diferencia importante entre la ingeniería de sistemas y de software es la inclusión de una variedad de disciplinas profesionales a lo largo de la vida del sistema. Por ejemplo, en la figura 10.5 se muestran las disciplinas técnicas que intervienen en la procuración y el desarrollo de un nuevo sistema para el manejo de tráfico aéreo. Están implicados arquitectos e ingenieros civiles, porque los nuevos sistemas de manejo del tráfico aéreo, por lo general, tienen que instalarse en un nuevo edificio. Participan ingenieros eléctricos y mecánicos para especificar y mantener la energía eléctrica y el aire acondicionado. Los ingenieros en electrónica se ocupan de las computadoras, los radares y otros equipos. Los ergonomistas diseñan las estaciones de trabajo de los controladores, en tanto que los ingenieros de software y diseñadores de interfaz de usuario se encargan del software del sistema.

Es esencial la inclusión de varias disciplinas profesionales, ya que existen demasiados aspectos diferentes en los sistemas sociotécnicos complejos. Sin embargo, las variedades entre disciplinas pueden introducir vulnerabilidades en los sistemas y, por consiguiente, comprometer la seguridad y la confiabilidad del sistema que se va a desarrollar:

1. Diferentes disciplinas usan las mismas palabras que significan distintas cuestiones. Las interpretaciones equivocadas son comunes entre ingenieros con antecedentes diversos. Si no se descubren y resuelven durante el desarrollo del sistema, podrían conducir a errores en los sistemas entregados. Por ejemplo, un ingeniero electrónico que sepa poco acerca de la programación C#, tal vez no entienda que un método en Java es comparable a una función en C.

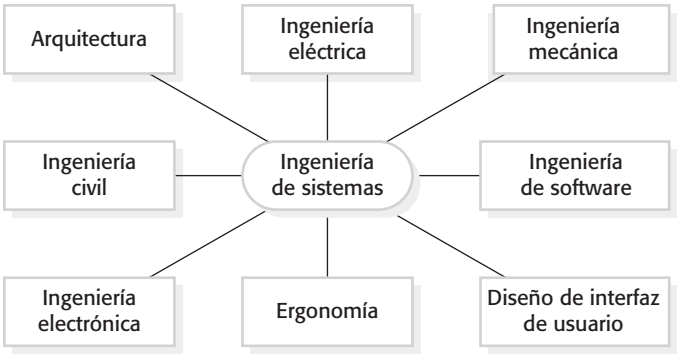


Figura 10.5 Disciplinas profesionales que intervienen en la ingeniería de sistemas

2. Cada disciplina hace suposiciones sobre lo que puede hacer o no otra disciplina. Tales suposiciones se basan con frecuencia en una comprensión inadecuada de lo que es posible realmente. Por ejemplo, un diseñador de interfaz de usuario propondrá una UI (*User Interface*) gráfica para un sistema embebido que requiera mayor cantidad de procesamiento y, en consecuencia, sobrecargará el procesador del sistema.
3. Las disciplinas tratan de proteger sus fronteras profesionales y justifican ciertas decisiones de diseño porque éstas requerirán de su experiencia profesional. En consecuencia, un ingeniero de software puede cuestionar en favor de un sistema de cierre de puertas de un edificio basado en software, aun cuando sería más fiable un sistema mecánico basado en llaves.

10.3 Procuración del sistema

La fase inicial de la ingeniería de sistemas es la procuración del sistema (llamada en ocasiones adquisición del sistema). En esta etapa se toman decisiones sobre el ámbito de un sistema que se adquirirá, los presupuestos y plazos del sistema, así como sobre los requerimientos de alto nivel del sistema. Al usar esta información, se toman más decisiones sobre si se procura un sistema, el tipo de sistema requerido y el (los) proveedor(es) del sistema. Los controladores para tales decisiones son:

1. *El estado de otros sistemas de la organización* Si la organización tiene una mezcla de sistemas que no logran comunicarse con facilidad o que son costosos de mantener, entonces la procuración de un sistema de reemplazo conduciría a beneficios empresariales significativos.
2. *La necesidad de cumplir con regulaciones externas* Cada vez más, las empresas están reguladas y deben demostrar el cumplimiento con regulaciones definidas de manera externa (por ejemplo, las regulaciones de contabilidad Sarbanes-Oxley en Estados Unidos). Esto podría requerir la sustitución de los sistemas que no cumplen o la provisión de nuevos sistemas específicamente para monitorizar el cumplimiento.
3. *Competencia externa* Si una empresa necesita competir de forma más efectiva o mantener una posición competitiva, sería conveniente realizar la inversión en nuevos

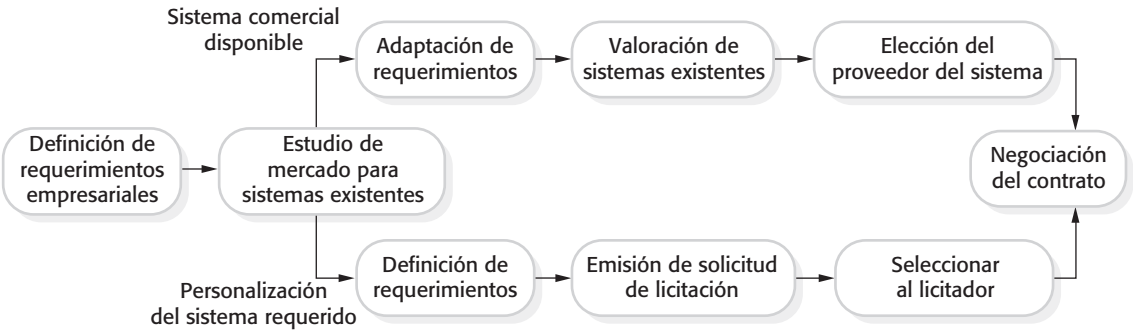


Figura 10.6 Procesos de procuración del sistema

sistemas que mejoren la eficiencia de los procesos empresariales. Para sistemas militares, la necesidad de mejorar la capacidad ante las nuevas amenazas es una importante razón para procurar sistemas nuevos.

4. *Reorganización empresarial* Las empresas y otras organizaciones se reestructuran frecuentemente con la intención de mejorar la eficiencia y/o el servicio al cliente. Las reorganizaciones conducen a cambios en los procesos empresariales que requieren soporte de nuevos sistemas.

5. *Presupuesto disponible* El presupuesto disponible es un factor innegable en la determinación del ámbito de los nuevos sistemas que pueden procurarse.

Además, con frecuencia se procuran nuevos sistemas gubernamentales que reflejan los cambios y principios políticos. Por ejemplo, en ocasiones los políticos deciden comprar sistemas de vigilancia nuevos que, afirman, combatirán el terrorismo. Al comprar tales sistemas muestran a los electores que toman acciones. Sin embargo, dichos sistemas se gestionan a menudo sin un análisis de costo/beneficio, donde se comparen los beneficios que resultan de las diferentes opciones de gastos.

Los sistemas grandes y complejos consisten por lo general en una mezcla de componentes comerciales y especialmente construidos. Una razón por la que se incluye cada vez más software en los sistemas es que permite mayor uso de los componentes de hardware existentes, en que el software actúa como "pegamento" (*glue*) para hacer que dichos componentes de hardware funcionen en conjunto de manera efectiva. La necesidad por desarrollar este "glueware" es una razón por la cual en ocasiones los ahorros para usar componentes comerciales (*off-the-shelf*) no son tan cuantiosos como se anticipaba.

La figura 10.6 ilustra un modelo simplificado del proceso de procuración, tanto para componentes del sistema COTS como para componentes del sistema que se diseñaron y desarrollaron a la medida. Los puntos importantes sobre el proceso que se muestra en este diagrama son:

1. Los componentes comerciales regularmente no cubren con exactitud los requerimientos, a menos que éstos se hayan escrito con tales componentes en mente. Por ello, elegir un sistema significa que debe encontrarse la coincidencia más cercana entre los requerimientos del sistema y las funcionalidades ofrecidas por los sistemas comerciales. Entonces, quizá tengan que modificarse los requerimientos. Esto podría tener efectos secundarios sobre otros subsistemas.

2. Cuando un sistema se construye a la medida, la especificación de requerimientos forma parte del contrato para el sistema que se va a adquirir. Por consiguiente, se trata de un documento tanto legal como técnico.

3. Después de seleccionar a un contratista para construir un sistema, hay un periodo de negociación del contrato en que quizá se tengan que negociar más cambios a los requerimientos, y discutir temas como el costo de los cambios al sistema. De igual modo, una vez seleccionado un sistema COTS, posiblemente se deba negociar con el proveedor acerca de costos, condiciones de licencia, posibles cambios al sistema, etcétera.

El software y el hardware en los sistemas sociotécnicos los desarrolla por lo general una organización diferente (el proveedor) de la organización que procura el sistema sociotécnico global. La razón para esto es que la empresa del cliente rara vez desarrolla software, de manera que sus empleados no tienen las habilidades necesarias para diseñar los sistemas por sí mismos. De hecho, muy pocas compañías tienen las capacidades para diseñar, fabricar y poner a prueba todos los componentes de un gran sistema sociotécnico complejo.

En consecuencia, el proveedor del sistema, que comúnmente se conoce como el contratista principal, con frecuencia subcontrata el desarrollo de diferentes subsistemas a algunos subcontratistas. Para sistemas grandes, como los sistemas de control del tráfico aéreo, un grupo de proveedores pueden formar un consorcio para licitar por el contrato. El consorcio debe incluir todas las capacidades requeridas para este tipo de sistema. Esto incluye proveedores de hardware de computadora, desarrolladores de software, proveedores de periféricos y de equipo especial, como sistemas de radar.

El comprador trata con el contratista y no con los subcontratistas, de modo que hay una sola interfaz comprador/proveedor. Los subcontratistas diseñan y construyen partes del sistema bajo una especificación que produce el contratista principal. Una vez completado, el contratista principal integra los diferentes componentes y los entrega al cliente. Dependiendo del contrato, el comprador puede permitir que el contratista principal seleccione libremente a los subcontratistas, o bien, solicitar al contratista principal que elija subcontratistas de entre una lista autorizada.

La toma de decisiones y elecciones durante la procuración del sistema tiene una gran influencia sobre la seguridad y confiabilidad de un sistema. Por ejemplo, si se toma una decisión para procurar un sistema comercial, entonces la organización debería aceptar que tienen influencia muy limitada sobre los requerimientos de seguridad y confiabilidad de este sistema. Ello depende básicamente de las decisiones tomadas por los proveedores del sistema. Además, los sistemas comerciales pueden tener debilidades de seguridad conocidas o requerir configuración compleja. Los errores de configuración, donde los puntos de entrada al sistema no se aseguran de manera adecuada, son una fuente principal de problemas de seguridad.

Por otro lado, una decisión para procurar un sistema personalizado significa que debe dedicarse un esfuerzo considerable para entender y definir los requerimientos de seguridad y confiabilidad. Si una compañía tiene experiencia limitada en el área, ésta será una labor muy difícil de realizar. Si debe lograrse el nivel de confiabilidad requerido, así como un rendimiento aceptable del sistema, entonces tal vez sea necesario extender el tiempo de desarrollo y aumentar el presupuesto.

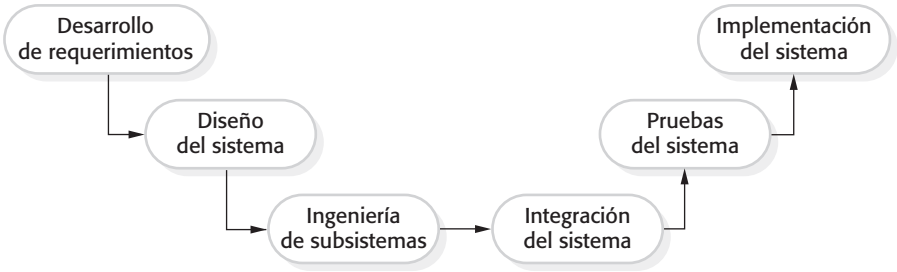


Figura 10.7 Desarrollo de sistemas

10.4 Desarrollo del sistema

Las metas del proceso de desarrollo del sistema son diseñar o adquirir todos los componentes de un sistema y, después, integrar dichos componentes para elaborar el sistema final. Los requerimientos son el puente entre los procesos de procuración y de desarrollo. Durante la procuración se definen los requerimientos empresariales y los requerimientos funcionales y no funcionales de alto nivel del sistema. Se puede considerar esto como el inicio del desarrollo, de ahí el traslape de los procesos que se muestran en la figura 10.4. Una vez acordados los contratos para los componentes del sistema, tiene lugar la ingeniería más detallada de los requerimientos.

La figura 10.7 es un modelo del proceso de desarrollo de sistemas. Este proceso de ingeniería de sistemas fue una importante influencia en el modelo "en cascada" del proceso de software estudiado en el capítulo 2. Aunque ahora se acepta que el modelo "en cascada" en general no es adecuado para el desarrollo del software, la mayoría de los procesos de desarrollo de sistemas son procesos derivados de un plan que todavía siguen dicho modelo.

Los procesos derivados de un plan se utilizan en la ingeniería de sistemas, ya que diferentes partes del sistema se desarrollan al mismo tiempo. Para sistemas que incluyen hardware y otro equipo, los cambios durante el desarrollo suelen ser muy costosos o, en ocasiones, prácticamente imposibles. Por consiguiente, es esencial que los requerimientos del sistema estén completamente entendidos antes de comenzar el desarrollo o el trabajo de construcción del hardware. Rara vez es posible reformar el diseño del sistema para resolver problemas de hardware. Por tal razón, cada vez más funcionalidad del sistema se asigna al software del sistema. Esto permite algunos cambios durante el desarrollo del sistema, en respuesta a los nuevos requerimientos del sistema que surgen inevitablemente.

Uno de los aspectos que más confunden en la ingeniería de sistemas es que las compañías usan una terminología diferente para cada etapa del proceso. También varía la estructura del proceso. En algunas ocasiones, la ingeniería de requerimientos es parte del proceso de desarrollo y, en otras, es una actividad independiente. Sin embargo, en esencia hay seis actividades fundamentales en cuanto al desarrollo de sistemas:

1. *Desarrollo de requerimientos* Los requerimientos de alto nivel y empresariales identificados durante el proceso de procuración deben desarrollarse con más detalle. Los requerimientos deben asignarse al hardware, al software o a los procesos, y hay que priorizar su implementación.

2. *Diseño del sistema* Este proceso se superpone significativamente con el proceso de desarrollo de requerimientos. Implica el establecimiento de la arquitectura global del sistema, al identificar los diferentes componentes del sistema y entender las relaciones entre ellos.
3. *Ingeniería de subsistemas* Esta etapa implica el desarrollo de los componentes de software del sistema; la configuración del hardware y software comerciales y, si es necesario, el diseño de hardware para un propósito especial; la definición de los procesos operacionales para el sistema, y el rediseño de los procesos empresariales esenciales.
4. *Integración del sistema* Durante esta etapa, los componentes se reúnen para crear un nuevo sistema. Sólo entonces se vuelven evidentes las propiedades emergentes del sistema.
5. *Pruebas del sistema* Por lo general, ésta es una actividad extensa y prolongada, donde se detectan los problemas. Se regresa a las fases de ingeniería de subsistemas y de integración del sistema para reparar dichos problemas, corregir el rendimiento del sistema e implementar nuevos requerimientos. Las pruebas del sistema pueden requerir tanto las pruebas por parte del desarrollador del sistema, como pruebas de aceptación/usuario por parte de la organización que procuró el sistema.
6. *Implementación del sistema* Éste es el proceso de poner el sistema a disposición de sus usuarios, de transferir datos de los sistemas existentes y establecer comunicaciones con otros sistemas en el entorno. El proceso culmina con una "inauguración", después de la cual los usuarios comienzan a usar el sistema para apoyar su trabajo.

Aunque el proceso global está derivado de un plan, los procesos de desarrollo de requerimientos y de diseño del sistema se encuentran inevitablemente ligados. Los requerimientos y el diseño de alto nivel se desarrollan de manera concurrente. Las restricciones impuestas por los sistemas existentes llegan a limitar las opciones de diseño y éstas pueden especificarse en los requerimientos. Probablemente se deba realizar cierto diseño inicial para estructurar y organizar el proceso de ingeniería de requerimientos. Conforme continúa el proceso de diseño, se descubrirían problemas con los requerimientos existentes y surgirían nuevos requerimientos. En consecuencia, se considera que estos procesos se vinculan de forma espiral, como se indica en la figura 10.8.

La espiral refleja la realidad de que los requerimientos afectan las decisiones de diseño y viceversa y, por lo tanto, tiene sentido entremezclar dichos procesos. A partir del centro, cada vuelta de la espiral puede agregar detalle a los requerimientos y al diseño. Algunas vueltas suelen enfocarse en los requerimientos y otras en el diseño. A veces, el nuevo conocimiento recolectado durante los requerimientos y el proceso de diseño significa que debe cambiarse el enunciado del problema en sí.

Para casi todos los sistemas, existen muchos diseños posibles que cumplen con los requerimientos. Éstos cubren una variedad de soluciones que combinan el hardware, el software y las operaciones humanas. La solución que usted elija para más desarrollo sería la solución técnica más adecuada que cubra los requerimientos. Sin embargo, consideraciones organizativas y políticas más amplias llegan a influir en la opción de solución. Por ejemplo, un cliente gubernamental quizá prefiera para su sistema el uso de proveedores nacionales en vez de extranjeros, incluso si los productos nacionales son técnicamente inferiores. Dichas influencias tienen por lo general efecto en la fase de revisión y valoración del modelo en espiral, donde diseños y requerimientos pueden aceptarse o rechazarse.

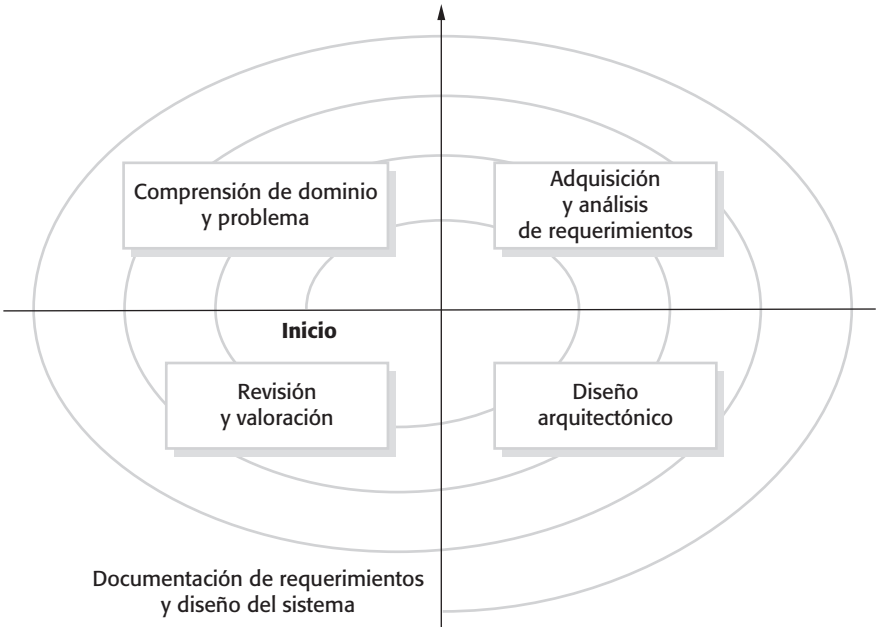


Figura 10.8 Espiral de requerimientos y diseño

El proceso termina cuando en una revisión se decide que los requerimientos y el diseño de alto nivel están suficientemente detallados para la especificación y el diseño de subsistemas.

En la fase de ingeniería de subsistemas se implementan los componentes de hardware y software del sistema. Para algunos tipos de sistema, como las aeronaves, todos los componentes de hardware y software pueden diseñarse y construirse durante el proceso de desarrollo. Sin embargo, en la mayoría de los sistemas, algunos componentes son sistemas comerciales (COTS). Suele ser mucho más barato comprar productos existentes que desarrollar componentes con un propósito especial.

Por lo general, los subsistemas se desarrollan en paralelo. Cuando se encuentran problemas que cruzan a través de las fronteras del subsistema, debe hacerse una petición para modificar el sistema. Cuando los sistemas requieran una extensa ingeniería del hardware, realizar modificaciones después de que comenzó la fabricación generalmente es muy costoso. A menudo, para compensar el problema tienen que encontrarse "soluciones". Éstas casi siempre implican cambios de software, debido a la flexibilidad inherente del mismo.

Durante la integración de sistemas se toman los subsistemas desarrollados de manera independiente y se reúnen para constituir un sistema completo. Esta integración puede formarse al usar un enfoque de "big-bang", donde todos los subsistemas se integran al mismo tiempo. Sin embargo, por razones técnicas y administrativas, la mejor opción es un proceso de integración incremental en que los subsistemas se integren uno a la vez:

1. Generalmente es imposible programar el desarrollo de los subsistemas de modo que todos se terminen al mismo tiempo.
2. La integración incremental reduce el costo de la localización del error. Si numerosos subsistemas se integran simultáneamente, un error que surja durante las pruebas podría estar en cualquiera de dichos subsistemas. Cuando un solo subsistema se

integra con un sistema ya en operación, los errores que ocurran probablemente estarán en el subsistema recientemente integrado, o bien, en las interacciones entre los subsistemas existentes y el nuevo subsistema.

Conforme más sistemas se construyen mediante la integración de componentes COTS de hardware y software, es cada vez más difusa la distinción entre implementación e integración. En algunos casos, no hay necesidad de desarrollar un nuevo hardware o software, y la integración es, en esencia, la fase de implementación del sistema.

Durante y después del proceso de integración, el sistema se pone a prueba. Al hacerlo, hay que enfocarse en poner a prueba las interfaces entre componentes y el comportamiento del sistema como un todo. Inevitablemente, esto también revelará problemas con subsistemas individuales que deban repararse.

Las fallas de subsistema son una consecuencia de suposiciones inválidas sobre otros subsistemas que, con frecuencia, se revelarán durante la integración del sistema. Esto conduciría a disputas entre los contratistas responsables de la implementación de diferentes subsistemas. Cuando se descubren problemas en la interacción de subsistemas, los contratistas pueden estar de acuerdo sobre cuál subsistema es el defectuoso. Tal vez las negociaciones referentes a cómo resolver los problemas duren semanas o meses.

La etapa final del proceso de desarrollo del sistema es la entrega y la implementación del mismo. El software se instala en el hardware y está listo para su operación. Esto podría requerir más configuración del sistema para reflejar el entorno local donde se usa, transferencia de datos de sistemas existentes y preparación de la documentación y capacitación del usuario. En esta etapa quizá también se deban reconfigurar otros sistemas en el entorno, con la finalidad de garantizar que el nuevo sistema interopere con ellos.

Aunque directa en principio, en ocasiones durante la implementación surgen varias dificultades. El entorno del usuario puede ser diferente del anticipado por los desarrolladores del sistema y tal vez sea difícil adaptar el sistema para enfrentar diversos entornos de usuario. Los datos existentes quizá requieran de una limpieza a fondo y las partes de los mismos pueden perderse. Las interfaces hacia otros sistemas tal vez no estén documentadas de forma adecuada.

Es clara la influencia de los procesos de desarrollo del sistema sobre la confiabilidad y la seguridad. Durante tales procesos, es cuando se toman decisiones sobre los requerimientos de confiabilidad y seguridad, así como sobre las negociaciones entre costos, fecha, rendimiento y confiabilidad. Los errores humanos en todas las etapas del proceso de desarrollo suelen conducir a la introducción de fallas en el sistema que, en la operación, llevarían a una falla del sistema. Los procesos de prueba y validación están inevitablemente restringidos por los costos y el tiempo disponible. Como resultado, el sistema quizá no se pruebe de manera adecuada. Se deja a los usuarios probar el sistema conforme lo utilicen. Finalmente, los problemas en la implementación del sistema podrían significar que hay una falta de concordancia entre el sistema y su entorno operacional. Esto puede conducir a errores humanos al usar el sistema.

10.5 Operación del sistema

Los procesos operacionales son aquellos que están relacionados con el uso del sistema para su propósito definido. Por ejemplo, los operadores de un sistema de control de tráfico aéreo siguen procesos específicos cuando una aeronave entra y sale del espacio aéreo,

cuando tienen que cambiar altura o velocidad, cuando ocurre una emergencia, etcétera. Para los nuevos sistemas, dichos procesos operacionales deben definirse y documentarse durante el proceso de desarrollo del sistema. Tal vez se deba capacitar a los operadores y adaptar otros procesos laborales para usar el nuevo sistema de manera efectiva. En esta etapa podrían surgir problemas no descubiertos, porque la especificación del sistema tenga errores u omisiones. Así, aunque el sistema se desempeñe de acuerdo con las especificaciones, probablemente sus funciones no cubran las necesidades operacionales reales. En consecuencia, los operadores quizá no usen el sistema como pretendían los diseñadores.

El beneficio clave de contar con operadores de sistemas es que las personas tienen una capacidad única de responder efectivamente ante situaciones inesperadas, aunque nunca hayan tenido una experiencia directa con tales situaciones. Por lo tanto, cuando las cosas salen mal, los operadores con frecuencia rescatan la situación, aunque en ocasiones ello signifique que se infrinja el proceso definido. Los operadores también emplean su conocimiento local para adaptar y mejorar los procesos. Normalmente, los procesos operacionales reales son diferentes de los anticipados por los diseñadores del sistema.

Por consiguiente, se requiere diseñar procesos operacionales que sean flexibles y adaptables. Los procesos operacionales no tienen que ser demasiado restrictivos, ni requerir que las operaciones se realicen en un orden específico, ni tampoco el software del sistema debe apoyarse en el seguimiento de un proceso específico. Los operadores por lo regular mejoran el proceso, ya que saben qué funciona y qué no funciona en una situación real.

Un problema que surge sólo después de que el sistema entra en operación es la operación del nuevo sistema junto con los sistemas existentes. Pueden existir problemas físicos de incompatibilidad o quizá resulte difícil transferir datos de un sistema a otro. También surgirían problemas más sutiles debido a que diferentes sistemas tienen distintas interfaces de usuario. Introducir un nuevo sistema puede aumentar la tasa de error del operador, ya que los operadores usan comandos de interfaz de usuario para el sistema equivocado.

10.5.1 Error humano

En las primeras páginas del capítulo se sugirió que el no determinismo era un conflicto importante en los sistemas sociotécnicos, y que una razón para ello obedecía a que las personas en el sistema no se comportan siempre de la misma forma. Algunas veces cometen errores al usar el sistema y esto tiene el potencial de provocar fallas del sistema. Por ejemplo, si un operador olvida registrar que se realizó cierta acción, otro operador (erróneamente) repetiría dicha acción. Si la acción, por ejemplo, es la deducción o acreditación de una cuenta bancaria, entonces ocurre una falla del sistema, pues es incorrecta la cantidad en la cuenta.

Como afirma Reason (2000), los errores humanos ocurrirán siempre y existen dos formas de considerar el problema del error humano:

1. *El enfoque personal*. Los errores se consideran responsabilidad del individuo y los "actos inseguros" (como un operador que falla al implementar una barrera de seguridad) son consecuencia de un descuido individual o un comportamiento imprudente. Las personas que adoptan este enfoque creen que los errores humanos suelen reducirse con amenazas disciplinarias, procedimientos más rigurosos, capacitación adicional, etcétera. Su visión es que el error es culpa del individuo responsable por cometer la falla.

2. *El enfoque de sistemas*. El supuesto básico es que las personas son falibles y se equivocarán. Los errores que la gente comete por lo general son consecuencia de decisiones de diseño del sistema, que llevan a formas erróneas de trabajar, o bien, de factores de la organización, que afectan a los operadores del sistema. Los sistemas eficaces tienen que reconocer la posibilidad del error humano, e incluir barreras y protecciones que los detecten, y permitir al sistema recuperarse antes de que ocurra la falla. Cuando ésta sucede, la prioridad no es encontrar al individuo para echarle la culpa, sino entender cómo y por qué las protecciones del sistema pasaron por alto el error.

Se considera que el enfoque de sistemas es el correcto, y que los ingenieros de sistemas deben admitir que ocurrirán errores humanos durante la operación del sistema. Por ello, para mejorar la seguridad y la confiabilidad de un sistema, los diseñadores deben pensar en las protecciones y barreras al error humano que tienen que incluirse en un sistema. También es necesario considerar si dichas barreras tienen que edificarse en los procedimientos técnicos del sistema. Si no es así, serían parte de los procesos y procedimientos para usar el sistema, o serían los lineamientos del operador los que dependen de la verificación y reflexión humanas.

Los siguientes son ejemplos de protecciones que pueden incluirse en un sistema:

1. Un sistema de control de tráfico aéreo podría incluir un sistema automatizado de alerta de conflicto. Cuando un controlador instruya a una aeronave a cambiar su velocidad o su altitud, el sistema extrapola su trayectoria para saber si podría colisionar contra alguna otra aeronave. Si es así, se activa una alarma.
2. El mismo sistema podría tener un procedimiento claramente definido para registrar las instrucciones de control emitidas. Dichas acciones ayudan al controlador a comprobar si emitieron correctamente la instrucción y a disponer de información para que la verifiquen otros.
3. El control de tráfico aéreo comúnmente implica un equipo de controladores que, de manera continua, monitorizan el trabajo de los demás. Por consiguiente, cuando se comete un error, es probable que se detecte y corrija antes de que ocurra un incidente.

Inevitablemente, todas las barreras presentan debilidades de algún tipo. Reason las llama “condiciones latentes”, pues usualmente sólo contribuyen a la falla del sistema cuando ocurre algún otro problema. Por ejemplo, sobre las defensas, un punto débil de un sistema de alerta de conflicto es que puede conducir a muchas falsas alarmas. Por lo tanto, los controladores ignorarían las advertencias del sistema. Una inconsistencia de un sistema procedimental sería que la información inusual, pero esencial, no se registre con facilidad. La revisión humana suele fallar cuando todas las personas que intervienen enfrentan mucho estrés y cometen el mismo error.

Las condiciones latentes conducen a falla del sistema cuando las defensas construidas en éste no detectan una falla activa por parte de un operador del sistema. El error humano es un activador de la falla, pero no debe considerarse como la razón exclusiva de ella. Reason explica lo anterior utilizando su muy conocido modelo de “queso suizo” de la falla del sistema (figura 10.9).

En este modelo, las defensas construidas en un sistema se comparan con rebanadas de queso suizo. Algunos tipos de queso suizo, como el Emmental, tienen orificios y, por

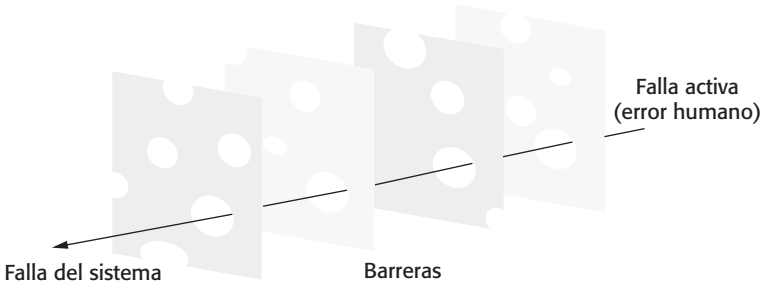


Figura 10.9 Modelo de queso suizo de Reason de la falla del sistema

consiguiente, la analogía es que las condiciones latentes son comparables con los orificios en las rebanadas de queso. La posición de dichas aberturas no es estática, sino que cambia según el estado del sistema sociotécnico global. Si cada rebanada representa una barrera, las fallas podrían ocurrir cuando los orificios se alinean al mismo tiempo que un error operativo humano. Una falla activa de la operación del sistema pasa a través de los agujeros, y conduce a una falla global del sistema.

Por lo general, claro, los agujeros no deben alinearse de modo que el sistema detecte las fallas operacionales. Para reducir la probabilidad de que las fallas del sistema sean producto del error humano, los diseñadores tienen que:

1. Diseñar un sistema que incluya diferentes tipos de barreras. Esto significa que los "orificios" probablemente estarán en diferentes lugares y, por lo tanto, hay una posibilidad menor de que los orificios se alineen y fracasen para detectar un error.
2. Minimizar el número de condiciones latentes en un sistema. Efectivamente, esto significa reducir el número y el tamaño de los "orificios" del sistema.

Desde luego, el diseño del sistema como totalidad también debe tratar de evitar que las fallas activas originen una falla del sistema. Esto implicaría el diseño de los procesos operacionales y el sistema para garantizar que los operadores no trabajen en exceso, se distraigan o se les presenten cantidades excesivas de información.

10.5.2 Evolución del sistema

Los sistemas grandes y complejos tienen una vida muy larga. Durante ella, cambian para corregir errores en los requerimientos originales del sistema e implementar los nuevos requerimientos que hayan surgido. Es probable que las computadoras del sistema se sustituyan por máquinas nuevas y más rápidas. La organización que usa el sistema puede reorganizarse a sí misma y, por consiguiente, usar el sistema en una forma diferente. El entorno externo del sistema llega a variar, lo cual forzaría los cambios al sistema. En consecuencia, la evolución, donde el sistema se modifica para acomodar el cambio ambiental, es un proceso que funciona junto con los procesos operacionales normales del sistema. La evolución del sistema implica reingresar al proceso de desarrollo para realizar cambios y extensiones al hardware, el software y los procesos operacionales del sistema.

Sistemas heredados

Los sistemas heredados son sistemas sociotécnicos basados en computadoras que se desarrollaron en el pasado, usando con frecuencia tecnología más antigua u obsoleta. Dichos sistemas incluyen no sólo el hardware y el software, sino también procesos y procedimientos heredados, añejas formas de hacer las cosas que son difíciles de transformar porque se apoyan en el software heredado. Los cambios a una parte del sistema requieren inevitablemente variaciones a otros componentes. Los sistemas heredados usualmente son sistemas empresariales críticos. Se mantienen porque es muy arriesgado sustituirlos.

http://www.SoftwareEngineering-9.com/LegacySys/

La evolución del sistema, como la evolución del software (estudiado en el capítulo 9), es inherentemente costosa por varias razones:

1. Los cambios propuestos tienen que analizarse con sumo cuidado, desde una perspectiva empresarial y técnica. Los cambios deben contribuir a las metas del sistema y no simplemente tener motivaciones técnicas.
2. Puesto que los subsistemas nunca son totalmente independientes, los cambios a un subsistema podrían afectar de modo adverso el rendimiento o el comportamiento de otros subsistemas. Por ende, quizá se requieran cambios consecuentes a dichos subsistemas.
3. Las razones para las decisiones de diseño original no se registran con frecuencia. Los responsables de la evolución del sistema tienen que reflexionar por qué se tomaron decisiones específicas de diseño.
4. Conforme aumenta la antigüedad de los sistemas, su estructura general puede haberse corrompido progresivamente por las modificaciones, de modo que aumentan los costos al realizar más cambios.

Los sistemas que evolucionaron con el tiempo dependen a menudo de tecnología obsoleta de hardware y software. Si tienen un papel crítico en una organización, se conocen como "sistemas heredados". Por lo general, se trata de sistemas que la organización quisiera sustituir, pero no lo hace ya que los riesgos o costos de la sustitución no son justificables.

Desde una perspectiva de confiabilidad y seguridad, los cambios a un sistema con frecuencia son una fuente de problemas y de vulnerabilidades. Si las personas que implementan los cambios no son las mismas que desarrollaron el sistema, quizá no estén al tanto de que una decisión de diseño se tomó por razones de confiabilidad y seguridad. En consecuencia, podrían cambiar el sistema y perder parte de las protecciones que se implementaron deliberadamente cuando se construyó el sistema. Más aún, dado que las pruebas son costosas, volver a probar por completo sería casi imposible después de cada cambio al sistema. En tal caso, probablemente no se descubran los efectos colaterales adversos de los cambios, que introducen o exponen fallas en otros componentes del sistema.

PUNTOS CLAVE

- Los sistemas sociotécnicos incluyen el hardware, el software y al personal de cómputo, que se sitúan dentro de una organización. Éstos se diseñan para apoyar las metas y los objetivos empresariales o de la organización.
- Los factores humanos y de la organización, como la estructura y las políticas de la organización, tienen un efecto significativo sobre la operación de los sistemas sociotécnicos.
- Las propiedades emergentes de un sistema son características del sistema como un todo, y no de sus partes componentes. Incluyen propiedades como rendimiento, fiabilidad, usabilidad, seguridad y protección. El éxito o el fracaso de un sistema dependen con frecuencia de dichas propiedades emergentes.
- Los procesos fundamentales de la ingeniería de sistemas son procuración, desarrollo y operación del sistema.
- La procuración del sistema cubre todas de las actividades que intervienen en decidir qué sistema comprar y quién debe suministrar dicho sistema. Los requerimientos de alto nivel se desarrollan como parte del proceso de procuración.
- El desarrollo del sistema incluye la especificación de requerimientos, diseño, construcción, integración y pruebas. La integración del sistema, donde los subsistemas con más de un proveedor deben reunirse para trabajar juntos, es particularmente crítica.
- Cuando un sistema se pone en funcionamiento, los procesos operacionales y el sistema en sí deben variar para reflejar los cambiantes requerimientos empresariales.
- Los errores humanos son inevitables y los sistemas deben incluir barreras para detectar dichos errores, antes de que deriven en una falla del sistema. El modelo de queso suizo de Reason explica cómo el error humano y los defectos latentes en las barreras pueden conducir a fallas del sistema.

LECTURAS SUGERIDAS

"Airport 95: Automated baggage system". Un estudio de caso excelente y comprensible de lo que puede salir mal con un proyecto de ingeniería de sistemas y cómo el software tiende a culparse por fallas de sistemas más amplias. (*ACM Software Engineering Notes*, 21, marzo de 1996.) http://doi.acm.org/10.1145/227531.227544.

"Software system engineering: A tutorial". Un amplio panorama general de la ingeniería de sistemas, aunque Thayer se enfoca exclusivamente en los sistemas basados en computadora, sin examinar los temas sociotécnicos. (R. H. Thayer. *IEEE Computer*, abril de 2002.) http://dx.doi.org/10.1109/MC.2002.993773.

Trust in Technology: A Socio-technical Perspective. Este libro es un conjunto de ensayos, los cuales tratan, en cierta forma, de la confiabilidad de los sistemas sociotécnicos. (K. Clarke, G. Hardstone, M. Rouncefield y I. Sommerville (eds.), Springer, 2006.)

"Fundamentals of Systems Engineering". Éste es el capítulo introductorio del manual de ingeniería de sistemas de la NASA. Presenta una visión del proceso de ingeniería de sistemas para sistemas

espaciales. Aunque se trata de sistemas técnicos principalmente, hay temas sociotécnicos por considerar. Desde luego, la confiabilidad es primordial. (En *NASA Systems Engineering Handbook*, NASA-SP2007-6105, 2007.) http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA%20SP-2007-6105%20Rev%201%20Final%2031Dec2007.pdf.

EJERCICIOS

10.1. Dé dos ejemplos de funciones gubernamentales que reciban soporte de sistemas sociotécnicos complejos, y explique por qué, en el futuro previsible, dichas funciones no serán completamente automatizadas.

10.2. Exponga por qué el entorno donde se instaló un sistema basado en computadora tendría efectos no anticipados sobre el sistema, que conduzcan a falla del mismo. Ilustre su respuesta con un ejemplo diferente del que se usó en este capítulo.

10.3. ¿Por qué es imposible inferir las propiedades emergentes de un sistema complejo a partir de las propiedades de los componentes del sistema?

10.4. ¿Por qué en ocasiones es difícil decidir si hubo o no una falla en un sistema sociotécnico? Aclare su respuesta con ejemplos del MHC-PMS que se estudió en capítulos anteriores.

10.5. ¿Qué es un "problema malvado"? Explique por qué el desarrollo de un sistema nacional de registros médicos tiene que considerarse como un "problema malvado".

10.6. Un sistema de museo multimedia que ofrece experiencias virtuales de la antigua Grecia se desarrollará para un consorcio de museos europeos. El sistema debe proporcionar a los usuarios la facilidad de ver modelos en 3-D de la antigua Grecia a través de un navegador Web estándar, y también tiene que soportar una experiencia de realidad virtual de inmersión. ¿Qué dificultades políticas y organizacionales surgirían cuando el sistema se instale en los museos que constituyen el consorcio?

10.7. ¿Por qué la integración de un sistema es una parte especialmente crítica del proceso de desarrollo de sistemas? Sugiera tres conflictos sociotécnicos que causen dificultades en el proceso de integración del sistema.

10.8. Exprese por qué en ocasiones los sistemas heredados son críticos para la operación de una empresa.

10.9. ¿Cuáles son los argumentos en favor y en contra de considerar la ingeniería de sistemas como una profesión por derecho propio, como la ingeniería eléctrica o la ingeniería de software?

10.10. Usted es un ingeniero que participa en el desarrollo de un sistema financiero. Durante la instalación, descubre que este sistema hará redundantes a un número significativo de individuos. Las personas en el entorno le niegan el acceso a información esencial para completar la instalación del sistema. ¿En qué medida debe, como ingeniero de sistemas, involucrarse en esta situación? ¿Es su responsabilidad profesional completar la instalación como se contrató? ¿Simplemente debe abandonar el trabajo hasta que la organización compradora resuelva el problema?

REFERENCIAS

Ackroyd, S., Harper, R., Hughes, J. A. y Shapiro, D. (1992). *Information Technology and Practical Police Work*. Milton Keynes: Open University Press.

Anderson, R. J., Hughes, J. A. y Sharrock, W. W. (1989). *Working for Profit: The Social Organization of Calculability in an Entrepreneurial Firm*. Aldershot: Avebury.

Checkland, P. (1981). *Systems Thinking, Systems Practice*. Chichester: John Wiley & Sons.

Checkland, P. y Scholes, J. (1990). *Soft Systems Methodology in Action*. Chichester: John Wiley & Sons.

Mumford, E. (1989). "User Participation in a Changing Environment—Why we need it". En *Participation in Systems Development*. Knight, K. (ed.). Londres: Kogan Page.

Reason, J. (2000). "Human error: Models and management". *British Medical J*., **320** 768–70.

Rittel, H. y Webber, M. (1973). "Dilemmas in a General Theory of Planning". *Policy Sciences*, **4**, 155–69.

Stevens, R., Brook, P., Jackson, K. y Arnold, S. (1998). *Systems Engineering: Coping with Complexity*. Londres: Prentice Hall.

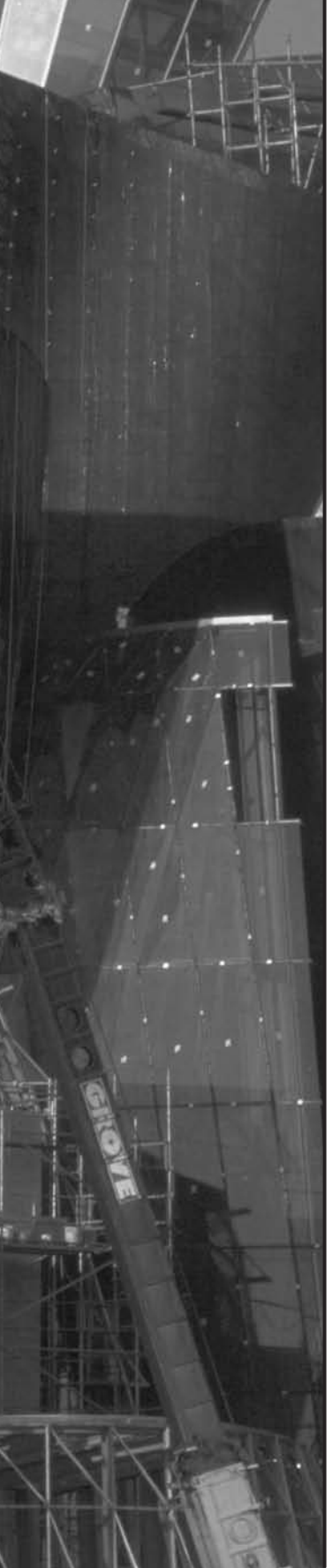
Suchman, L. (1987). *Plans and situated actions: the problem of human-machine communication*. Nueva York: Cambridge University Press.

Swartz, A. J. (1996). "Airport 95: Automated Baggage System?" *ACM Software Engineering Notes*, **21** (2), 79–83.

Thayer, R. H. (2002). "Software System Engineering: A Tutorial." *IEEE Computer*, **35** (4), 68–73.

Thomé, B. (1993). "Systems Engineering: Principles and Practice of Computer-based Systems Engineering". Chichester: John Wiley & Sons.

White, S., Alford, M., Holtzman, J., Kuehl, S., McCay, B., Oliver, D., Owens, D., Tully, C. y Willey, A. (1993). "Systems Engineering of Computer-Based Systems". *IEEE Computer*, **26** (11), 54–65.



11

Confiabilidad y seguridad

Objetivos

El objetivo de este capítulo es introducirlo a la confiabilidad y la seguridad del software. Al estudiar este capítulo:

- comprenderá por qué la confiabilidad y la seguridad son, por lo general, más importantes que las características funcionales de un sistema de software;
- entenderá las cuatro principales dimensiones de la confiabilidad, es decir: disponibilidad, fiabilidad, protección y seguridad;
- conocerá la terminología especializada que se usa cuando se analiza la seguridad y confiabilidad;
- sabrá que, para lograr un software confiable y seguro, necesita evitar errores durante el desarrollo de un sistema, detectarlos y eliminarlos cuando el sistema esté en uso, así como reducir el daño causado por fallas operativas.

Contenido

11.1 Propiedades de confiabilidad
11.2 Disponibilidad y fiabilidad
11.3 Protección
11.4 Seguridad

Conforme los sistemas de cómputo se insertan profundamente en las vidas empresariales y personales, se incrementan los problemas que derivan de las fallas del sistema y del software. Una falla del software del servidor en una empresa de comercio electrónico podría conducir a dicha compañía hacia una gran pérdida de ingresos y, posiblemente, también de clientes. Un error de software en un sistema de control embebido en un automóvil provocaría costosas devoluciones de ese modelo por reparación y, en los peores casos, sería un factor que contribuya a los accidentes. La infección de la compañía de las PC con malware requiere costosas operaciones de limpieza para solventar el problema y quizá dé como resultado la pérdida o el daño de información sensible.

Puesto que los sistemas intensivos de software son tan importantes para los gobiernos, las compañías y los individuos, es esencial que este tipo de software sea digno de confianza. El software debe estar disponible cuando se requiera y ejecutarse correctamente y sin efectos colaterales indeseables, como la circulación de información no autorizada. El término "confiabilidad" fue propuesto por Laprie (1995) para cubrir los atributos relacionados con sistemas de disponibilidad, fiabilidad, protección y seguridad. Como se estudia en la sección 11.1, dichas propiedades están inextricablemente vinculadas, así que tiene sentido disponer de un solo término para tratarlas a todas.

La confiabilidad de los sistemas es ahora usualmente más importante que su funcionalidad detallada por las siguientes razones:

1. *Las fallas del sistema afectan a un gran número de individuos* Muchos sistemas incluyen funcionalidad que se usa rara vez. Si esta funcionalidad se retirara del sistema, tan sólo un número menor de usuarios resultarían afectados. Las fallas del sistema, que afectan la disponibilidad de un sistema, gravitarían potencialmente a todos los usuarios del sistema. La falla significaría que es imposible continuar con la normalidad del negocio.
2. *Los usuarios rechazan a menudo los sistemas que son poco fiables, carecen de protecciones o son inseguros* Si los usuarios descubren que un sistema es poco fiable o inseguro, lo rechazarán. Más aún, ellos también pueden negarse a adquirir o usar otros productos de la compañía que desarrolló el sistema que no es fiable, porque considerarán que dichos productos tienen la misma probabilidad de no ser fiables o seguros.
3. *Los costos por las fallas del sistema suelen ser enormes* Para ciertas aplicaciones, como un sistema de control de reactor o un sistema de navegación de aeronaves, el costo por la falla del sistema es una orden de magnitud mayor que el costo del sistema de control.
4. *Los sistemas no confiables pueden causar pérdida de información* Los datos son muy costosos de recolectar y mantener; por lo general, valen mucho más que el sistema de cómputo donde se procesan. El costo por recuperar datos perdidos o contaminados generalmente es muy alto.

Como se observó en el capítulo 10, el software es siempre parte de un sistema más amplio. Se ejecuta en un entorno operacional que incluye el hardware donde se produce el software, en los usuarios de dicho software y los procesos de la organización o empresa en que se utiliza el software. Por lo tanto, al diseñar un sistema confiable se debe considerar:

1. *Falla del hardware* El hardware del sistema puede fallar por errores en su diseño, componentes que se averían por errores de fabricación o porque los componentes llegaron al final de su vida operativa.

Sistemas críticos

Algunas clases de sistemas son "sistemas críticos" en los que la falla del sistema podría derivar en una lesión a individuos, daño al ambiente o pérdidas económicas mayores. Los ejemplos de sistemas críticos incluyen sistemas embebidos en dispositivos médicos, como una bomba de insulina (crítica para la protección), sistemas de navegación de aeronaves (críticos para la misión) y sistemas de transferencia de dinero en línea (críticos para la empresa).

Los sistemas críticos son muy costosos de desarrollar. No sólo tienen que diseñarse de modo que sean muy raras las fallas, sino también deben incluir mecanismos de recuperación que se usen al ocurrir las fallas.

http://www.SoftwareEngineering-9.com/Web/Dependability/CritSys.html

2. *Falla en el desarrollo de software* El software del sistema puede fallar debido a errores en su especificación, diseño o implementación.
3. *Falla de operación* Los usuarios fallan al usar o ejecutar el sistema correctamente. Conforme el hardware y el software se vuelven más confiables, las fallas en la operación son ahora, quizá, la principal causa individual de fallas del sistema.

Dichas fallas suelen estar interrelacionadas. Un componente de hardware que falla significaría que los operadores del sistema tengan que lidiar con una situación inesperada y una carga de trabajo adicional, que los pone bajo estrés; así, las personas cometen errores con frecuencia. Esto llega a provocar que falle el software, lo que genera más trabajo para los operadores, incluso más estrés, entre otras contrariedades.

Como resultado, es importante sobre todo que los diseñadores de sistemas confiables con software intensivo adquieran una perspectiva holística de los sistemas, en vez de enfocarse en un solo aspecto del sistema, como su software o hardware. Si estos últimos, además de los procesos operacionales se diseñan por separado, sin considerar las debilidades potenciales de otras partes del sistema, entonces es muy probable que ocurran errores en las interfaces entre las diferentes partes del sistema.

11.1 Propiedades de confiabilidad

Todos estamos familiarizados con el problema de la falla de los sistemas de cómputo. Por razones que no son evidentes, en ocasiones las computadoras fallan o se descomponen en alguna forma. Los programas que operan en estas computadoras quizá no operen como se espera y, en ocasiones, contaminan los datos que administra el sistema. Uno aprende a vivir con tales fallas, aunque pocos confían por completo en las computadoras que usan normalmente.

La confiabilidad de un sistema de cómputo es una propiedad del sistema que refleja su fiabilidad. Aquí, esta última significa en esencia el grado de confianza que un usuario tiene que el sistema ejecutará como se espera, y que el sistema no "fallará" en su uso normal. No es significativo expresar numéricamente la confiabilidad. En vez de ello, se usan

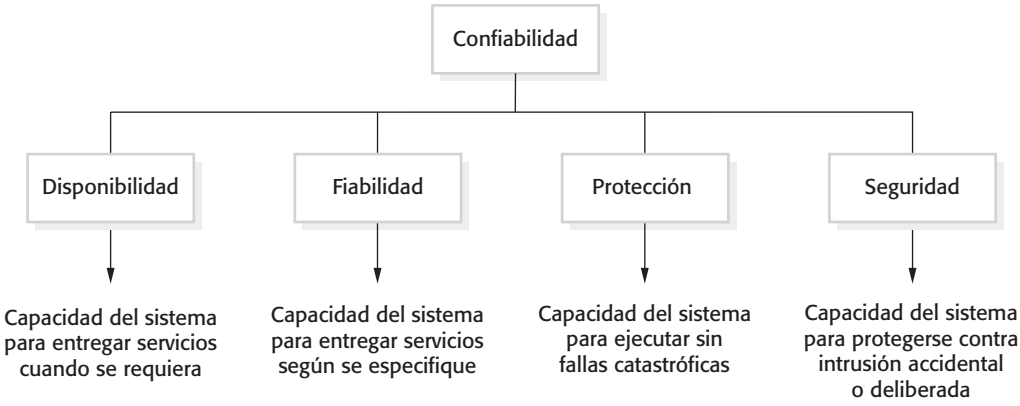


Figura 11.1 Principales propiedades de confiabilidad

términos relativos como "no confiable", "muy confiable" y "ultraconfiable", para reflejar los grados de confianza que uno espera de un sistema.

Desde luego, fiabilidad y utilidad no son lo mismo. El autor del libro no considera que el procesador de texto que usó para escribirlo sea un sistema muy confiable. Algunas veces se congela y tiene que reiniciarse. No obstante, puesto que es muy útil, el autor se preparó para tolerar fallas esporádicas. Sin embargo, para reflejar la falta de confianza en el sistema, frecuentemente guarda su trabajo y mantiene múltiples copias de respaldo. Para compensar la falta de confiabilidad del sistema, toma acciones que limitan el daño que resultaría de la falla del sistema.

Hay cuatro dimensiones principales de la confiabilidad, como se indica en la figura 11.1.

1. *Disponibilidad* De manera informal, la disponibilidad de un sistema es la probabilidad de que en un momento dado éste funcionará, ejecutará y ofrecerá servicios útiles a los usuarios.
2. *Fiabilidad* De manera informal, la fiabilidad de un sistema es la probabilidad, durante un tiempo determinado, de que el sistema brindará correctamente servicios como espera el usuario.
3. *Protección* De modo no convencional, la protección de un sistema es un juicio de cuán probable es que el sistema causará daños a las personas o su ambiente.
4. *Seguridad* Informalmente, la seguridad de un sistema es un juicio de cuán probable es que el sistema pueda resistir intrusiones accidentales o deliberadas.

Las propiedades de confiabilidad que se muestran en la figura 11.1 son propiedades complejas que podrían fragmentarse en algunas otras propiedades más simples. Por ejemplo, la seguridad incluye "integridad" (garantiza que los programas y datos del sistema no se dañen) y "confidencialidad" (garantiza que sólo personas autorizadas accedan a la información). La fiabilidad incluye "exactitud" (garantiza que los servicios del sistema sean los especificados), "precisión" (garantiza que la información se entregue en un nivel de detalle adecuado) y "oportunidad" (garantiza que la información se entregue cuando se requiera).

Naturalmente, no todas esas propiedades de confiabilidad son aplicables a todos los sistemas. Para el sistema de bomba de insulina, que se trató en el capítulo 1, las propiedades más importantes son disponibilidad (debe funcionar cuando se requiere), fiabilidad (tiene que suministrar la dosis correcta de insulina) y protección (nunca debe entregar una dosis peligrosa de insulina). La seguridad no es un problema, pues la bomba no mantiene información confidencial. No está en red y, por consiguiente, no puede ser atacada de manera maliciosa. Para el sistema meteorológico a campo abierto, disponibilidad y fiabilidad son las propiedades más importantes, porque los costos de reparación suelen ser muy altos. En tanto, para el sistema de información de pacientes, la seguridad es esencialmente importante, debido a que mantiene los datos privados y sensibles.

Además de estas cuatro propiedades básicas de confiabilidad, también se pueden considerar las siguientes:

1. *Reparabilidad* Las fallas del sistema son inevitables; no obstante, el desajuste causado por la falla podría minimizarse siempre que el sistema logre repararse rápidamente. Para que ello ocurra, se puede diagnosticar el problema, acceder al componente que falló y hacer cambios para corregir dicho componente. La reparabilidad en software se mejora cuando la organización que usa el sistema tiene acceso al código fuente y cuenta con las habilidades para cambiarlo. El software de fuente abierta facilita esta labor, aunque la reutilización de componentes suele dificultarlo más.

2. *Mantenibilidad* Mientras se usan los sistemas, surgen nuevos requerimientos y es importante mantener la utilidad de un sistema al cambiarlo para acomodar estos nuevos requerimientos. El software mantenible es aquel que económicamente se adapta para lidiar con los nuevos requerimientos, y donde existe una baja probabilidad de que los cambios insertarán nuevos errores en el sistema.

3. *Supervivencia* Un atributo muy importante para sistemas basados en Internet es la supervivencia (Ellison *et al*., 1999b). La supervivencia es la habilidad de un sistema para continuar entregando servicio en tanto está bajo ataque y mientras que, potencialmente, parte del sistema se deshabilita. El trabajo sobre supervivencia se enfoca en la identificación de los componentes clave del sistema y en la garantía de que ellos puedan entregar un servicio mínimo. Para mejorar la supervivencia se usan tres estrategias: resistencia al ataque, reconocimiento del ataque y recuperación del daño causado por un ataque (Ellison *et al*., 1999a; Ellison *et al*., 2002). En el capítulo 14 se examina esto con más detalle.

4. *Tolerancia para el error* Esta propiedad se considera como parte de la usabilidad y refleja la medida en que el sistema se diseñó de modo que se eviten y toleren los errores de entrada de usuario. Cuando ocurren errores de usuario, el sistema debe, en la medida de lo posible, detectar dichos errores y corregirlos automáticamente o solicitar al usuario que reintroduzca sus datos.

La noción de confiabilidad del sistema como propiedad que abarca la disponibilidad, seguridad, fiabilidad y protección, se introdujo porque estas propiedades están estrechamente relacionadas. La operación segura del sistema depende, por lo general, de que éste se halle disponible y opere de manera fiable. Un sistema se volvería no fiable cuando un curioso corrompa sus datos. Los ataques de negación de servicio sobre un sistema tienen

la intención de comprometer la disponibilidad de un sistema. Si un sistema se infecta con un virus, entonces no se estaría seguro de su fiabilidad o su protección, dado que el virus podría cambiar su comportamiento.

Por lo tanto, para desarrollar software confiable, es necesario garantizar que:

1. Se evite la entrada de errores accidentales en el sistema durante la especificación y el desarrollo del software.
2. Se diseñen procesos de verificación y validación que sean efectivos en el descubrimiento de errores residuales que afecten la confiabilidad del sistema.
3. Se desarrollen mecanismos de protección contra ataques externos que comprometan la disponibilidad o la seguridad del sistema.
4. Se configuren correctamente el sistema utilizado y el software de apoyo para su entorno operacional.

Además, conviene suponer que el software por lo general no es perfecto y que pueden ocurrirle fallas. Por consiguiente, el software debe incluir mecanismos de recuperación que hagan posible la restauración del servicio normal del sistema tan rápido como sea posible.

La necesidad para tolerar fallas significa que los sistemas confiables deben incluir un código redundante que los ayude a monitorizarse a sí mismos, detectar estados erróneos y recuperarse de las fallas en el desarrollo antes de que ocurran fallas en la operación. Ello afecta el rendimiento de los sistemas, pues se requiere verificación adicional cada vez que se ejecuta el sistema. Por consiguiente, los diseñadores comúnmente deben negociar entre rendimiento y confiabilidad. Tal vez se necesite dejar las comprobaciones del sistema porque éstas lo ralentizan. Sin embargo, aquí la consecuencia del riesgo es que ocurran algunas fallas en la operación a causa de que no se detectó el error.

Debido a costos de diseño, implementación y validación adicionales, se incrementa la confiabilidad de un sistema al aumentar significativamente los costos de desarrollo. En particular, los costos de validación son altos para sistemas que deben ser ultraconfiables, como los sistemas de control críticos para la protección. Además de validar que el sistema cumpla con sus requerimientos, el proceso de validación debería ser probado por un regulador externo para que el sistema sea seguro. Por ejemplo, los sistemas en las aeronaves tienen que probar a los reguladores, como la Federal Aviation Authority, que la probabilidad de una falla catastrófica del sistema, que afecte la seguridad de la aeronave, es extremadamente baja.

La figura 11.2 muestra la relación entre los costos y las mejoras incrementales en la confiabilidad. Si su software no es muy confiable, quizás obtenga mejoras considerables a costos relativamente bajos usando mejor ingeniería de software. Sin embargo, si ya usa buenas prácticas, los costos de mejorar son mucho mayores aun cuando los beneficios de dicha mejora sean modestos. También está el problema de probar el software para demostrar que es confiable. Esto se apoya en ejecutar muchas pruebas y observar el número de fallas que ocurren. Conforme su software se vuelva más confiable, verá cada vez menos fallas. En consecuencia, se necesitan cada vez más pruebas para tratar y evaluar cuántos problemas permanecen en el software. Como las pruebas son muy costosas, esto aumenta drásticamente el costo de los sistemas de alta confiabilidad.

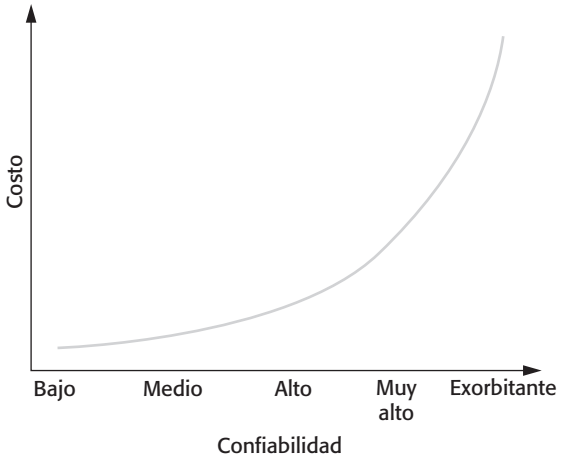


Figura 11.2 Curva de costo/confiabilidad

11.2 Disponibilidad y fiabilidad

La disponibilidad y fiabilidad del sistema son propiedades estrechamente relacionadas que se expresan también como probabilidades numéricas. La disponibilidad de un sistema es la probabilidad de que el sistema funcionará y ejecutará para entregar a los usuarios sus servicios bajo pedido. La fiabilidad de un sistema es la probabilidad de que los servicios del sistema se entregarán como se definió en la especificación del sistema. Si, en promedio, 2 entradas de cada 1,000 causan fallas, entonces la fiabilidad, que se expresa como una tasa de ocurrencia de falla es 0.002. Si la disponibilidad es 0.999, esto significa que, durante cierto tiempo, el sistema está disponible para el 99.9% de ese tiempo.

La fiabilidad y disponibilidad están estrechamente relacionadas, pero en ocasiones una es más importante que la otra. Si los usuarios esperan un sistema en servicio continuo, el sistema en tal caso tiene un alto requerimiento de disponibilidad. Debe estar disponible siempre que se realice una demanda. Sin embargo, si son bajas las pérdidas que resultan de una falla de sistema y éste se recupera rápidamente, entonces las fallas no afectan seriamente a los usuarios del sistema. En tales sistemas, los requerimientos de fiabilidad suelen ser relativamente bajos.

Un conmutador telefónico que enruta llamadas telefónicas es un ejemplo de un sistema donde la disponibilidad es más importante que la fiabilidad. Los usuarios esperan un tono de marcado cuando levantan el auricular, así que el sistema tiene altos requerimientos de disponibilidad. Si ocurre una falla del sistema mientras se establece una conexión, esto con frecuencia es rápidamente recuperable. Los conmutadores por lo general restablecen el sistema y reintentan la conexión. Ello puede hacerse tan rápidamente que los usuarios del teléfono incluso quizá no se enteren de que ocurrió una falla. Más aún, si una llamada incluso se interrumpe, las consecuencias regularmente no son serias. Por consiguiente, la disponibilidad más que fiabilidad es el principal requerimiento de confiabilidad para este tipo de sistema.

La fiabilidad y disponibilidad del sistema pueden definirse de manera más precisa como sigue:

1. *Fiabilidad* La probabilidad de operación libre de falla durante cierto tiempo, en un entorno dado, para un propósito específico.

2. *Disponibilidad* La probabilidad de que un sistema, en un momento en el tiempo, sea operativo y brinde los servicios solicitados.

Uno de los problemas prácticos en el desarrollo de sistemas fiables es que las nociones intuitivas de fiabilidad y disponibilidad en ocasiones son más amplias que estas definiciones limitadas. La definición de fiabilidad establece que deben tomarse en cuenta tanto el entorno donde se usa el sistema, como el propósito para el que se utiliza. Si se mide la fiabilidad de un sistema en un entorno, no se puede suponer que ésta será la misma si el sistema se usa en una forma diferente.

Por ejemplo, suponga que se mide la fiabilidad de un procesador de texto en un entorno de oficina, donde la mayoría de los usuarios no se interesan en la operación del software. Ellos seguirán las instrucciones para su uso, pero no tratarán de experimentar con el sistema. Si luego se mide la fiabilidad del mismo sistema en un entorno universitario, entonces la fiabilidad sería bastante diferente. Aquí, quizá los estudiantes exploren las fronteras del sistema y lo usen de formas inesperadas, lo cual podría resultar en fallas del sistema que no ocurrirían en el ambiente más restringido de la oficina.

Estas definiciones estándar de disponibilidad y fiabilidad no consideran la severidad de la falla ni las consecuencias de la indisponibilidad. Aun cuando las personas aceptan con frecuencia fallas menores del sistema, se preocupan más por fallas serias que deriven en altos costos. Por ejemplo, son menos aceptables las fallas de cómputo que corrompen datos almacenados, que las fallas que congelan la máquina y que se resuelven al reiniciar la computadora.

Una definición estricta de fiabilidad relaciona la implementación del sistema con su especificación. Esto es, el sistema se conduce de manera fiable, si su comportamiento es consistente con el definido en la especificación. Sin embargo, una causa común de falibilidad percibida es que la especificación del sistema no coincide con las expectativas de los usuarios del sistema. Por desgracia, muchas especificaciones están incompletas o son incorrectas, y se deja a los ingenieros de software interpretar cómo debería comportarse el sistema. Como no son expertos de dominio, pueden no implementar, por consiguiente, el comportamiento que esperan los usuarios. Desde luego, también es cierto que los usuarios no leen las especificaciones del sistema, de manera que quizá tengan expectativas irreales del sistema.

Evidentemente, la disponibilidad y la fiabilidad están vinculadas, pues las fallas del sistema pueden colapsar al sistema. No obstante, la disponibilidad no sólo depende del número de caídas del sistema, sino también del tiempo necesario para reparar los componentes que causaron la falla. Por ello, si el sistema A falla una vez al año y el sistema B falla una vez al mes, entonces a todas luces A es más fiable que B. Sin embargo, suponga que el sistema A tarda tres días en reiniciarse después de una falla, mientras que el sistema B tarda sólo 10 minutos en reiniciar. La disponibilidad del sistema B durante el año (120 minutos de tiempo muerto) es mucho mejor que la del sistema A (4,320 minutos de tiempo muerto).

La alteración provocada por sistemas indisponibles no se refleja en la simple métrica de disponibilidad que especifica el porcentaje de tiempo en que el sistema está disponible. El momento en que falla el sistema es también significativo. Si un sistema no está disponible durante una hora cada día, entre las 3 y 4 de la mañana, quizá no afecte a muchos usuarios. Pero, si el mismo sistema está indisponible durante 10 minutos a lo largo del día laboral, la indisponibilidad del sistema probablemente tendrá un efecto mucho mayor.

Término	Descripción
Error o equivocación humano	El comportamiento humano que resulta en la introducción de fallas en el desarrollo en un sistema. Por ejemplo, en la estación meteorológica a campo abierto, un programador podría decidir que la forma de calcular la hora para la siguiente transmisión es agregar una hora a la hora actual. Esto funciona salvo cuando la hora de transmisión es entre 23:00 y medianoche (medianoche es 00:00 en el reloj de 24 horas).
Falla en el desarrollo del sistema	Una característica de un sistema de software que puede conducir a un error del sistema. La falla en el desarrollo es la inclusión del código para agregar una hora a la hora de la última transmisión, sin una verificación para saber si la hora es mayor o igual a 23:00.
Error del sistema	Un estado erróneo del sistema que puede conducir a un comportamiento de sistema que es inesperado para los usuarios del mismo. El valor de la hora de transmisión se establece de manera incorrecta (a 24.XX en vez de 00.XX), cuando se ejecuta el código defectuoso.
Caída del sistema	Un evento que ocurre en algún punto del tiempo, cuando el sistema no entrega un servicio como espera su usuario. No se transmiten datos meteorológicos porque la hora es inválida.

Figura 11.3 Terminología de fiabilidad

Los problemas de fiabilidad y disponibilidad del sistema se originan principalmente por caídas de éste. Algunas de estas caídas de sistema son una consecuencia de errores de especificación o fallas en otros sistemas relacionados, como un sistema de comunicación. A pesar de ello, muchas fallas en la operación son resultado del comportamiento erróneo del sistema que se derivan de fallas en el desarrollo del sistema. Cuando se discute la fiabilidad, es útil usar terminología precisa y distinguir entre los términos "falla en el desarrollo", "error del sistema" y "caída del sistema". En la figura 11.3 se definen dichos términos, y cada definición se ilustra con un ejemplo de la estación meteorológica a campo abierto.

Cuando una entrada o una secuencia de entradas provocan la ejecución de código defectuoso en un sistema, se crea un estado erróneo que podría conducir a una falla en la operación del software. La figura 11.4, derivada de Littlewood (1990), muestra un sistema de software como un mapeo de un conjunto de entradas a un conjunto de salidas. Dada una entrada o secuencia de entradas, el programa responde por la producción de una salida correspondiente. Por ejemplo, dada una entrada de una URL, un navegador Web produce una salida que es el despliegue de la página Web solicitada.

La mayoría de las entradas no conducen a caídas del sistema. Sin embargo, algunas entradas o combinaciones de entradas, que se muestran en la elipse sombreada I_e en la figura 11.4, causan la generación de caídas del sistema o salidas erróneas. La fiabilidad del programa depende del número de entradas del sistema que son miembros del conjunto de entradas que conducen a una salida errónea. Si las entradas en el conjunto I_e se ejecutan para usar regularmente partes del sistema, entonces las caídas serán frecuentes. No obstante, si las entradas en I_e se efectúan por un código que rara vez se usa, entonces los usuarios difícilmente verán alguna vez las caídas.

Puesto que cada usuario de un sistema lo utiliza de manera diferente, tienen diversas percepciones de su fiabilidad. Las fallas en el desarrollo que afectan la fiabilidad del sistema para un usuario no pueden nunca revelarse bajo el modo de trabajo de alguien más (figura 11.5). En esta misma figura, el conjunto de entradas erróneas corresponde a la elipse marcada I_e en la figura 11.4. El conjunto de entradas producido por el usuario 2 se interseca con este conjunto de entradas erróneas. Por lo tanto, el usuario 2 experimenta

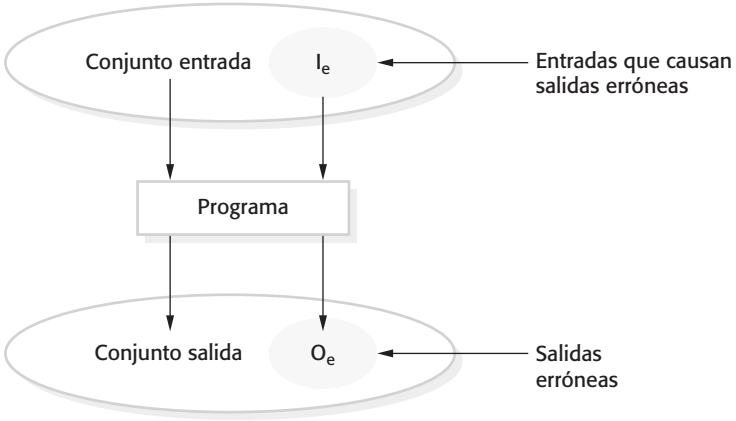


Figura 11.4 Un sistema como un mapeo entrada/salida

algunas caídas del sistema. Sin embargo, el usuario 1 y el usuario 3 nunca usan entradas del conjunto erróneo. Para ellos, el software siempre será fiable.

La fiabilidad práctica de un programa depende del número de entradas que causan salidas erróneas (caídas), durante el uso normal del sistema por la mayoría de los usuarios. Las fallas en el desarrollo de software que sólo ocurren en situaciones excepcionales tienen poco efecto práctico sobre la fiabilidad del sistema. En consecuencia, eliminar las fallas del software quizá no mejore significativamente la fiabilidad total del sistema. Mills y sus colaboradores (1987) descubrieron que eliminar el 60% de los errores conocidos en su software llevó a una mejoría del 3% en la fiabilidad. Adams (1984), en un estudio de productos de software IBM, notó que muchos defectos en los productos sólo era probable que causaran caídas después de cientos o miles de meses de uso del producto.

Las fallas en el desarrollo no siempre derivan en errores del sistema, y los errores del sistema no necesariamente originan caídas del sistema. Las razones para esto son las siguientes:

1. No todo el código en un programa se ejecuta. El código que incluye una falla en el desarrollo (por ejemplo, equivocarse al inicializar una variable) tal vez nunca se ejecute debido a la forma en que se usa el programa.

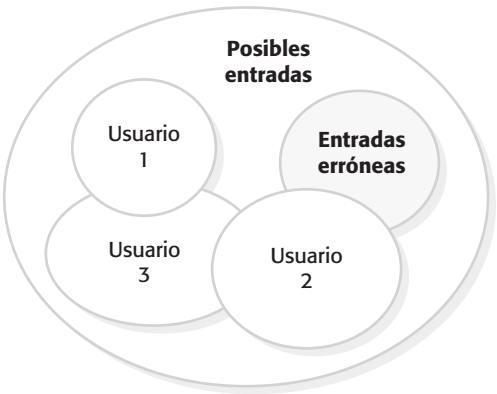


Figura 11.5 Patrones de uso de software

2. Los errores son transitorios. Una variable de estado puede tener un valor incorrecto causado por la ejecución de un código defectuoso. Sin embargo, antes de que se acceda a ésta y se origine una caída del sistema, es factible procesar alguna otra entrada del sistema que restablezca el estado a un valor válido.
3. El sistema puede incluir mecanismos de detección de fallas y de protección, que aseguran que el comportamiento erróneo se descubra y corrija antes de que resulten afectados los servicios del sistema.

Otra razón por la que las fallas en un sistema pueden no conducir a caídas del sistema es que, en la práctica, los usuarios adaptan su comportamiento para evitar el uso de entradas que saben que causan caídas del programa. Los usuarios experimentados "soslayan" las características de software que descubren que son falibles. Por ejemplo, el autor evita ciertas características, tales como la numeración automática en el sistema de procesamiento de texto que usó para escribir este libro. Cuando usaba la autonumeración, ésta normalmente salía mal. La reparación de las fallas de desarrollo en las características sin usar no muestra una diferencia práctica con la fiabilidad del sistema. A medida que los usuarios comparten información sobre problemas y soluciones alternativas, se reducen los efectos de los problemas del software.

La distinción entre fallas en el desarrollo, errores y caídas, que se explica en la figura 11.3, ayuda a identificar tres enfoques complementarios usados para mejorar la fiabilidad de un sistema:

1. *Prevención de fallas de desarrollo* Se usan técnicas de desarrollo que minimizan la posibilidad de los errores humanos y/o captan las equivocaciones antes de que resulten en fallas de desarrollo del sistema. Los ejemplos de estas técnicas incluyen evitar códigos del lenguaje de programación proclives al error, como punteros y el uso de análisis estático para descubrir anomalías del programa.
2. *Detección y eliminación de fallas en el desarrollo* El uso de técnicas de verificación y validación que incrementan las oportunidades de que se detecten y eliminen las fallas en el desarrollo antes de que se use el sistema. Las pruebas y la depuración sistemáticas son un ejemplo de una técnica de detección de este tipo de fallas.
3. *Tolerancia a fallas en el desarrollo* Se refiere a las técnicas que aseguran que las fallas en el desarrollo de un sistema no deriven en errores del sistema o que los errores del sistema no deriven en caídas del sistema. La incorporación de mecanismos de autocomprobación en un sistema y el uso de módulos de sistema redundantes son ejemplos de técnicas de tolerancia a la fallas en el desarrollo.

La aplicación práctica de estas técnicas se trata en el capítulo 13, que examina las técnicas para la ingeniería de software confiable.

11.3 Protección

Los sistemas críticos para la protección son aquellos en los que resulta esencial que la operación del sistema sea segura en todo momento; esto es, el sistema nunca debe dañar a las personas o a su entorno, incluso cuando falle. Los ejemplos de sistemas críticos para la protección incluyen los sistemas de control y monitorización en las aeronaves, los

sistemas de control de procesos en plantas químicas y farmacéuticas, y los sistemas de control de automotores.

El control de hardware de los sistemas críticos para la protección es más fácil de implementar y analizar que el control del software. A pesar de ello, ahora se construyen sistemas de tal complejidad que no pueden controlarse tan sólo con el hardware. El control del software es esencial debido a la necesidad de manejar gran cantidad de sensores y actuadores con leyes de control complejas. Por ejemplo, una aeronave militar avanzada, aerodinámicamente inestable, requiere ajuste continuo, controlado por software, de sus superficies de vuelo para garantizar que no se desplome.

El software crítico para la protección se divide en dos clases:

1. *Software primario crítico para la protección* Éste es software embebido que sirve como controlador en un sistema. El mal funcionamiento de tal software puede repetirse en el hardware, lo cual derivaría en una lesión humana o daño ambiental. El software de bomba de insulina, que se trató en el capítulo 1, es un ejemplo de un sistema primario crítico para la protección. La falla del sistema puede conducir a una lesión en el usuario.
2. *Software secundario crítico para la protección* Es un software que podría repercutir indirectamente en una lesión. Un ejemplo de dicho software consiste en un sistema de diseño de ingeniería auxiliado por computadora, cuyo mal funcionamiento ocasionaría un error de diseño en el objeto por desarrollar. Esta equivocación quizá cauce una lesión a los individuos, si el sistema diseñado funciona mal. Otro ejemplo de un sistema secundario crítico para la protección es el sistema de administración de atención a la salud mental, MHC-PMS. La falla de este sistema cuando un paciente inestable no se trata de manera adecuada podría llevar a que éste se lesione a sí mismo o a otros.

La fiabilidad y protección del sistema se relacionan, pero un sistema fiable puede ser inseguro y viceversa. El software todavía suele comportarse de tal forma que el comportamiento resultante del sistema conduzca a un accidente. Hay cuatro razones por las que los sistemas de software que son fiables no necesariamente son seguros:

1. Nunca se puede tener una total certeza de que un sistema de software esté libre de fallas en el desarrollo y sea tolerante a los mismos. Las fallas en el desarrollo no detectadas pueden estar inactivas durante mucho tiempo y las fallas en la operación del software pueden ocurrir después de muchos años de operación infalible.
2. La especificación podría estar incompleta en cuanto a que no describe el comportamiento requerido del sistema en algunas situaciones críticas. Un alto porcentaje de mal funcionamiento del sistema (Boehm *et al*., 1975; Endres, 1975; Lutz, 1993; Nakajo y Kume, 1991) es resultado de errores de especificación más que de diseño. En un estudio de errores en sistemas embebidos, Lutz concluye:

   > . . . las dificultades con los requerimientos son la clave principal de los errores de software que se relacionan con la seguridad, los cuales persisten hasta la integración y las pruebas del sistema.

3. El mal funcionamiento del hardware origina que el sistema se comporte de forma impredecible, y presente al software con un entorno no anticipado. Cuando los componentes se hallan cerca de la falla física, éstos pueden comportarse de manera errática y generar señales fuera de los rangos para el manejo del software.

Término	Definición
Accidente (o contratiempo)	Evento no planeado o secuencia de eventos que derivan en muerte o lesión de un individuo, o daño a la propiedad o al ambiente. Una sobredosis de insulina es un ejemplo de accidente.
Peligro	Condición con el potencial para causar o contribuir a un accidente. Un ejemplo de riesgo es una falla del sensor que mide la glucosa sanguínea.
Daño	Una medida de la pérdida que resulta de un contratiempo. El daño puede variar desde el hecho de que muchas personas mueran como resultado de un accidente, hasta una lesión o daño menor a la propiedad. El daño, producto de una sobredosis de insulina, sería una lesión grave o la muerte del usuario que utiliza la bomba de insulina.
Severidad del peligro	Una valoración del peor daño posible que resultaría de un peligro en particular. La severidad del peligro puede ser desde catastrófico, cuando muchas personas mueren, hasta menor, cuando sólo ocurre un daño mínimo. Cuando la muerte de un individuo es una posibilidad, la valoración razonable de la severidad del peligro es "muy alta".
Probabilidad del peligro	La probabilidad de que ocurran eventos que causen peligro. Los valores de probabilidad tienden a ser arbitrarios, pero varían de "probable" (por ejemplo, 1/100 de posibilidad de que ocurra un peligro) a "improbable" (no son probables situaciones concebibles en que pudiera ocurrir el peligro). Es baja la probabilidad de que la falla en un sensor de la bomba de insulina dé como resultado una sobredosis.
Riesgo	Ésta es una medida de probabilidad de que el sistema causará un accidente. El riesgo se valora al considerar la posibilidad, severidad y verosimilitud de que el peligro conduzca a un accidente. El riesgo de una sobredosis de insulina es quizá de medio a bajo.

Figura 11.6 Terminología de seguridad

4. Los operadores del sistema pueden generar entradas que no son individualmente incorrectas, pero que, en ciertas situaciones, conducirían a un mal funcionamiento del sistema. Un ejemplo anecdótico de esto sucedió cuando el tren de aterrizaje de una aeronave colapsó mientras la nave estaba en tierra. Al parecer, un técnico presionó un botón que ordenó al software de gestión de utilidades elevar el tren de aterrizaje. El software realizó la instrucción del mecánico a la perfección. Sin embargo, el sistema debía desactivar el comando a menos que el avión estuviera en el aire.

Un vocabulario especializado evolucionó con la finalidad de discutir los sistemas críticos para la seguridad, así como para comprender la importancia de los términos específicos utilizados. La figura 11.6 resume algunas definiciones de vocablos importantes, con ejemplos tomados del sistema de la bomba de insulina.

La clave para garantizar la seguridad consiste en cerciorarse de que no ocurrirán accidentes o de que serán mínimas las consecuencias de un accidente. Esto se logrará mediante tres formas complementarias:

1. *Evitar el peligro* El sistema está diseñado de modo que se eviten los riesgos. Por ejemplo, un sistema de corte, donde se requiera a un operador usar las dos manos

para presionar simultáneamente dos botones separados, evita el peligro de que las manos del operador estén en la ruta de las cuchillas.

2. *Detectar y eliminar el peligro* El sistema se diseña de modo que los peligros se detecten y eliminen antes de que ocasionen un accidente. Por ejemplo, cuando un sistema de una planta química detecta presión excesiva y abre una válvula de descarga para reducir la presión antes de que ocurra un estallido.

3. *Limitar el daño* El sistema puede incluir características de protección que minimicen el daño que resulte de un accidente. Por ejemplo, un motor de avión, por lo general, incluye extintores automáticos. Si ocurre un incendio, generalmente se suele controlar antes de que represente una amenaza para la aeronave.

Los accidentes ocurren con mayor frecuencia cuando muchas cosas salen mal al mismo tiempo. Un análisis de accidentes serios (Perrow, 1984) indica que casi todos se debieron a una combinación de fallas en diferentes partes de un sistema. Las combinaciones no anticipadas de fallas de subsistemas conducen a interacciones que derivaron en una falla global del sistema. Por ejemplo, la falla de un sistema de aire acondicionado podría conducir a sobrecalentamiento, que luego haría que el hardware del sistema generara señales incorrectas. Perrow también refiere que es imposible anticipar todas las combinaciones posibles de fallas. Por lo tanto, los accidentes son parte inevitable del uso de sistemas complejos.

Algunas personas usan esta explicación como un argumento contra el control del software. A causa de la complejidad del software, existen más interacciones entre las diferentes partes de un sistema, lo cual significa que probablemente existirán más combinaciones de fallas en el desarrollo que podrían conducir a una falla en la operación del sistema.

Sin embargo, los sistemas controlados por software pueden monitorizar una variedad más amplia de condiciones que los sistemas electromecánicos. Pueden adaptarse de manera relativamente sencilla. Usan hardware que tiene fiabilidad inherente muy alta y además es físicamente pequeño y ligero. Los sistemas controlados por software ofrecen seguridad sofisticada entrelazada. Soportan estrategias de control que disminuyan la cantidad de tiempo que necesitan emplear las personas en ambientes peligrosos. Aunque el control por software puede introducir más formas en las que un sistema podría salir mal, también permite mejor monitorización y protección y, en consecuencia, contribuirá a mejoras en la seguridad del sistema.

En todos los casos, es importante mantener un sentido de proporción sobre la seguridad del sistema. Es imposible hacer un sistema cien por ciento seguro, así que la sociedad debe decidir si las consecuencias de un accidente ocasional valen o no los beneficios que provienen del uso de tecnologías avanzadas. Asimismo, es una decisión social y política sobre cómo implementar recursos nacionales limitados para reducir el riesgo a la población.

11.4 Seguridad

La seguridad es un atributo del sistema que refleja la habilidad de éste para protegerse a sí mismo de ataques externos, que podrían ser accidentales o deliberados. Estos ataques externos son posibles puesto que la mayoría de las computadoras de propósito general

Término	Definición
Activo	Algo de valor que debe protegerse. El activo sería el sistema de software en sí o los datos usados por dicho sistema.
Exposición	Posible pérdida o daño a un sistema de cómputo. Esto sería la pérdida o el daño a los datos, o bien, una pérdida de tiempo y esfuerzo si es necesaria la recuperación después de una violación a la seguridad.
Vulnerabilidad	Una debilidad en un sistema basado en computadora que puede aprovecharse para causar pérdida o daño.
Ataque	Aprovechamiento de una vulnerabilidad del sistema. Por lo general, es desde afuera del sistema y es un intento deliberado por causar cierto daño.
Amenaza	Circunstancias que tienen potencial para causar pérdida o daño. Se puede pensar en ellas como una vulnerabilidad de sistema que está sujeta a un ataque.
Control	Medida de protección que reduce la vulnerabilidad de un sistema. La encriptación es un ejemplo de control que reduce la vulnerabilidad de un sistema de control de acceso débil.

Figura 11.7 Terminología de seguridad

ahora están en red y, en consecuencia, son accesibles a personas externas. Ejemplos de ataques pueden ser la instalación de virus y caballos de Troya, el uso sin autorización de servicios del sistema o la modificación no aprobada de un sistema o de sus datos. Si realmente se quiere un sistema seguro, es mejor no conectarlo a Internet. Siendo así, sus problemas de seguridad estarán limitados a garantizar que usuarios autorizados no abusen del sistema. Sin embargo, en la práctica, existen enormes beneficios del acceso en red para los sistemas más grandes, de modo que desconectarlos de Internet no es efectivo en costo.

Para algunos sistemas, la seguridad es la dimensión más importante de confiabilidad del sistema. Los sistemas militares, los de comercio electrónico y los que requieren procesamiento e intercambio de información confidencial deben diseñarse de modo que logren un alto nivel de seguridad. Por ejemplo, si un sistema de reservaciones de una aerolínea no está disponible, causará inconvenientes y ciertas demoras en la emisión de boletos. Sin embargo, si el sistema es inseguro, entonces un atacante podría borrar todos los libros y sería prácticamente imposible que continuaran las operaciones normales de la aerolínea.

Al igual que con otros aspectos de la confiabilidad, existe una terminología especializada asociada con la seguridad. Algunos términos importantes, como los analiza Pfleeger (Pfleeger y Pfleeger, 2007), se definen en la figura 11.7. La figura 11.8 incluye los conceptos de seguridad descritos en la figura 11.7 y muestra cómo se relacionan con el siguiente escenario extraído del MHC-PMS:

> *El personal clínico ingresa en el MHC-PMS con un nombre de usuario y contraseña. El sistema requiere que las contraseñas sean de al menos ocho caracteres; sin embargo, permite sin mayor verificación el establecimiento de cualquier contraseña. Un delincuente descubre que una figura deportiva bien pagada recibe tratamiento por problemas de salud mental. Le gustaría conseguir acceso ilegal a la información de este sistema, de modo que pueda extorsionar a la estrella.*

Término	Definición
Activo	Los registros de cada paciente que recibe o recibió tratamiento.
Exposición	Potencial pérdida financiera de futuros pacientes, quienes no buscan tratamiento porque no confían en que la clínica conserve sus datos. Pérdida financiera por acción legal de la estrella deportiva. Pérdida de reputación.
Vulnerabilidad	Sistema de contraseña débil que facilita a los usuarios establecer contraseñas sencillas de adivinar. Identificaciones de usuario que son iguales a los nombres.
Ataque	Imitación de un usuario autorizado.
Amenaza	Un usuario no autorizado conseguirá acceso al sistema al descifrar las credenciales (nombre y contraseña de ingreso) de un usuario autorizado.
Control	Un sistema de comprobación de contraseña que deshabilita las contraseñas de usuario que sean nombres propios o palabras que normalmente se incluyan en el diccionario.

Figura 11.8 Ejemplos de terminología de seguridad

Al pasar como pariente preocupado y hablar con las enfermeras en la clínica de salud mental, descubre cómo ingresar al sistema y a información personal sobre las enfermeras. Al revisar listas, descubre los nombres de algunas de las personas a quienes se permite el acceso. Entonces trata de ingresar en el sistema con dichos nombres y descifrar sistemáticamente posibles contraseñas (como nombres de los hijos).

En cualquier sistema en red, existen tres principales tipos de amenazas a la seguridad:

1. *Amenazas a la confidencialidad del sistema y sus datos* Pueden difundir información a individuos o programas que no están autorizados a tener acceso a dicha información.
2. *Amenazas a la integridad del sistema y sus datos* Tales amenazas pueden dañar o corromper el software o sus datos.
3. *Amenazas a la disponibilidad del sistema y sus datos* Dichas amenazas pueden restringir el acceso al software o sus datos a usuarios autorizados.

Desde luego, dichas amenazas son interdependientes. Si un ataque provoca que el sistema no esté disponible, entonces no podrá actualizar la información que cambia con el tiempo. Esto significa que la integridad del sistema puede estar comprometida; entonces, tal vez deba desmantelarse para reparar el problema. Por ello, se reduce la disponibilidad del sistema.

En la práctica, la mayoría de las vulnerabilidades en los sistemas sociotécnicos obedecen a fallas humanas más que a problemas técnicos. Las personas eligen contraseñas

fáciles de descifrar o escriben sus contraseñas en lugares sencillos de encontrar. Los administradores de sistemas cometen errores al establecer control de acceso o archivos de configuración, en tanto que los usuarios no instalan o usan software de protección. Sin embargo, como se estudió en la sección 10.5, se debe tener mucho cuidado cuando se clasifique un problema como un error de usuario. Los problemas humanos reflejan con frecuencia malas decisiones de diseño de sistemas que requieren, por ejemplo, cambios de contraseñas a menudo (de modo que los usuarios escriben sus contraseñas) o complejos mecanismos de configuración.

Los controles que se deben implementar para mejorar la seguridad del sistema son comparables con los de la fiabilidad y protección:

1. *Evitar la vulnerabilidad* Controles cuya intención sea garantizar que los ataques no tengan éxito. Aquí, la estrategia es diseñar el sistema de modo que se eviten los problemas de seguridad. Por ejemplo, los sistemas militares sensibles no están conectados a redes públicas, de forma que es imposible el acceso externo. También hay que pensar en la encriptación como un control basado en la evitación. Cualquier acceso no autorizado a datos encriptados significa que el atacante no puede leerlos. En la práctica, es muy costoso y consume mucho tiempo romper una encriptación fuerte.
2. *Detectar y neutralizar ataques* Controles cuya intención sea detectar y repeler ataques. Dichos controles implican la inclusión de funcionalidad en un sistema que monitoriza su operación y verifica patrones de actividad inusuales. Si se detectan, entonces se toman acciones, como desactivar partes del sistema, restringir el acceso a ciertos usuarios, etcétera.
3. *Limitar la exposición y recuperación* Controles que soportan la recuperación de los problemas. Éstos varían desde estrategias de respaldo automatizadas y "réplica" de la información, hasta pólizas de seguros que cubran los costos asociados con un ataque exitoso al sistema.

Sin un nivel razonable de seguridad, no se puede confiar en la disponibilidad, fiabilidad y protección de un sistema. Los métodos para certificar la disponibilidad, fiabilidad y seguridad suponen que el software en operación es el mismo que el software que se instaló originalmente. Si el sistema es atacado y el software se compromete de alguna forma (por ejemplo, si el software se modifica al incluir un gusano), entonces ya son insostenibles los argumentos de fiabilidad y protección.

Los errores en el desarrollo de un sistema pueden conducir a lagunas de seguridad. Si un sistema no responde a entradas inesperadas o si no se verifican los límites vectoriales, entonces los atacantes aprovecharían tales debilidades para conseguir acceso al sistema. Los principales incidentes de seguridad, como el gusano Internet original (Spafford, 1989) y el gusano Code Red más de 10 años después (Berghel, 2001) sacaron ventaja de la misma vulnerabilidad. Los programas en C# no incluyen comprobación de límites vectoriales, de manera que es posible sobrescribir parte de la memoria con código que permita el acceso no autorizado al sistema.

PUNTOS CLAVE

- Las fallas de los sistemas de cómputo críticos pueden conducir a grandes pérdidas económicas, pérdidas serias de información, daño físico o amenazas a la vida humana.
- La confiabilidad de un sistema de cómputo es una propiedad del sistema que refleja el grado de confianza del usuario en el sistema. Las dimensiones más importantes de confiabilidad son disponibilidad, fiabilidad, protección y seguridad.
- La disponibilidad de un sistema es la probabilidad de que el sistema entregará los servicios a sus usuarios cuando lo soliciten. La fiabilidad es la probabilidad de que los servicios del sistema se entregarán como se especificó.
- La fiabilidad percibida se relaciona con la probabilidad de que en el uso operacional ocurra un error. Un programa puede contener fallas en el desarrollo conocidas, pero aun así percibirse como fiable por parte de sus usuarios. Tal vez nunca usen características del sistema que sean afectadas por las fallas en el desarrollo.
- La protección de un sistema es un atributo que refleja la habilidad de mismo para ejecutar, normal o anormalmente, sin lesionar a los individuos o dañar el ambiente.
- La seguridad refleja la habilidad de un sistema para protegerse a sí mismo contra ataques externos. Las fallas en la seguridad pueden conducir a pérdidas de disponibilidad, daño al sistema o a sus datos, o bien, la fuga de información a personas no autorizadas.
- Sin un nivel razonable de seguridad, la disponibilidad, fiabilidad y protección del sistema estarían comprometidas si ataques externos dañan el sistema. Si un sistema no es fiable, es difícil garantizar la protección o la seguridad del sistema, ya que se comprometería por fallas del sistema.

LECTURAS SUGERIDAS

"The evolution of information assurance". Un excelente artículo que analiza la necesidad de proteger, contra accidentes y ataques, la información crítica en una organización. (R. Cummings, *IEEE Computer*, **35** (12), diciembre de 2002.) http://dx.doi.org/10.1109/MC.2002.1106181.

"Designing Safety Critical Computer Systems". Es una buena introducción al campo de los sistemas críticos para la protección, que analiza los conceptos fundamentales de peligros y riesgos. Más accesible que el libro de Dunn sobre sistemas críticos para la protección. (W. R. Dunn, *IEEE Computer*, **36** (11), noviembre de 2003.) http://dx.doi.org/10.1109/MC.2003.1244533.

Secrets and Lies: Digital Security in a Networked World. Un excelente libro, muy comprensible, acerca de la seguridad de computadoras, que examina el tema desde un enfoque sociotécnico. También son muy atractivas las columnas de Schneier en relación con los conflictos de seguridad en general (URL siguiente). (B. Schneier, John Wiley & Sons, 2004.) http://www.schneier.com/essays.html.

EJERCICIOS

11.1. Sugiera seis razones por las que la confiabilidad del software es importante en la mayoría de los sistemas sociotécnicos.

11.2. ¿Cuáles son las dimensiones más significativas de la confiabilidad de un sistema?

11.3. ¿Por qué los costos para garantizar la confiabilidad aumentan exponencialmente conforme se incrementan los requerimientos de fiabilidad?

11.4. Ofrezca razones para su respuesta y sugiera cuáles atributos de confiabilidad es probable que sean más críticos para los siguientes sistemas:

Un servidor de Internet proporcionado por un ISP con miles de clientes

Un bisturí controlado por computadora usado en cirugía laparoscópica

Un sistema de control direccional usado en un vehículo de lanzamiento de satélites

Un sistema de administración financiera personal basado en Internet

11.5. Identifique seis productos de consumo que sea probable que estén controlados mediante sistemas de software críticos para la protección.

11.6. Fiabilidad y protección son atributos de confiabilidad relacionados, pero distintos. Describa la diferencia más importante entre dichos atributos y explique por qué es posible que un sistema fiable sea inseguro, y viceversa.

11.7. En un sistema médico que se diseña para emitir radiación en el tratamiento de tumores, sugiera un riesgo potencial y proponga una característica de software que sirva para garantizar que el peligro identificado no origine un accidente.

11.8. En términos de seguridad de computadoras, explique las diferencias entre un ataque y una amenaza.

11.9. Con el MHC-PMS como ejemplo, identifique tres amenazas a este sistema (además de las mostradas en la figura 11.8). Sugiera controles que puedan aplicarse para reducir las posibilidades de un ataque exitoso con base en dichas amenazas.

11.10. Una organización que aboga por los derechos de las víctimas de tortura se pone en contacto con usted, como experto en seguridad computacional, y le solicita ayuda para conseguir acceso no autorizado a los sistemas de cómputo de una compañía estadounidense. Esto los ayudará a confirmar o negar que esta compañía vende equipo que se usa directamente en la tortura de prisioneros políticos. Discuta los dilemas éticos que se plantean en esta petición y cómo reaccionaría ante la misma.

REFERENCIAS

Adams, E. N. (1984). "Optimizing preventative service of software products". *IBM J. Res & Dev.*, **28** (1), 2–14.

Berghel, H. (2001). "The Code Red Worm". *Comm. ACM*, **44** (12), 15–19.

Boehm, B. W., McClean, R. L. y Urfig, D. B. (1975). "Some experience with automated aids to the design of large-scale reliable software". *IEEE Trans. on Software Engineering.*, **SE-1** (1), 125–33.

Ellison, R., Linger, R., Lipson, H., Mead, N. y Moore, A. (2002). "Foundations of Survivable Systems Engineering". *Crosstalk: The Journal of Defense Software Engineering*, **12**, 10–15.

Ellison, R. J., Fisher, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. A. y Mead, N. R. (1999a). "Survivability: Protecting Your Critical Systems". *IEEE Internet Computing*, **3** (6), 55–63.

Ellison, R. J., Linger, R. C., Longstaff, T. y Mead, N. R. (1999b). "Survivable Network System Analysis: A Case Study". *IEEE Software*, **16** (4), 70–7.

Endres, A. (1975). "An analysis of errors and their causes in system programs". *IEEE Trans. on Software Engineering.*, **SE-1** (2), 140–9.

Laprie, J.-C. (1995). "Dependable Computing: Concepts, Limits, Challenges". FTCS- 25: 25th IEEE Symposium on Fault-Tolerant Computing, Pasadena, Calif.: IEEE Press.

Littlewood, B. (1990). "Software Reliability Growth Models". In *Software Reliability Handbook*. Rook, P. (ed.). Amsterdam: Elsevier. 401–412.

Lutz, R. R. (1993). "Analysing Software Requirements Errors in Safety-Critical Embedded Systems". RE'93, San Diego, Calif: IEEE.

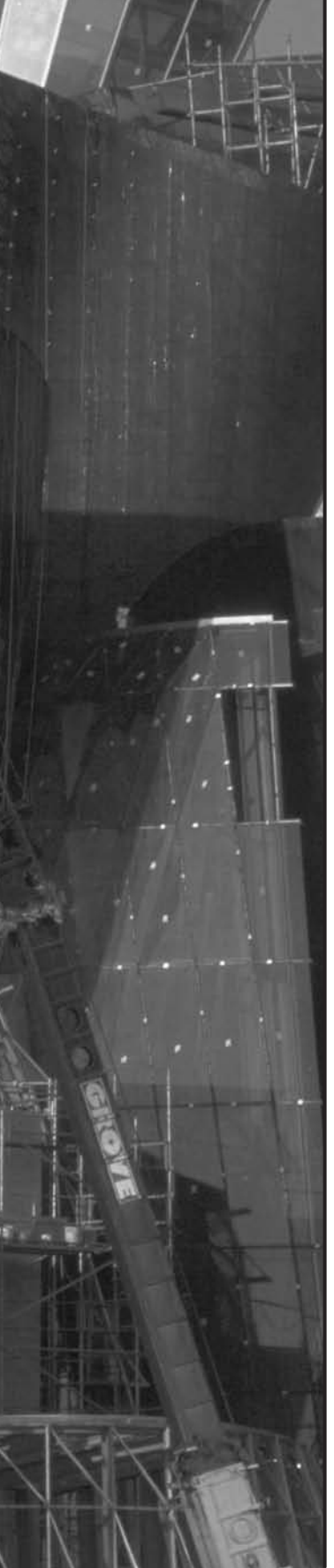
Mills, H. D., Dyer, M. y Linger, R. (1987). "Cleanroom Software Engineering". *IEEE Software*, **4** (5), 19–25.

Nakajo, T. y Kume, H. (1991). "A Case History Analysis of Software Error-Cause Relationships". *IEEE Trans. on Software Eng.*, **18** (8), 830–8.

Perrow, C. (1984). *Normal Accidents: Living with High-Risk Technology*. Nueva York: Basic Books.

Pfleeger, C. P. y Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.

Spafford, E. (1989). "The Internet Worm: Crisis and Aftermath". *Comm. ACM*, **32** (6), 678–87.



12

Especificación de confiabilidad y seguridad

Objetivos

El objetivo de este capítulo es conocer cómo se especifican los requerimientos de confiabilidad y seguridad funcionales y no funcionales. Al estudiar este capítulo:

- comprenderá cómo usar el enfoque dirigido por riesgos para identificar y analizar los requerimientos de protección, fiabilidad y seguridad;
- entenderá cómo utilizar los árboles de fallas para ayudar a analizar riesgos y derivar requerimientos de seguridad;
- se introducirá en las métricas para la especificación de fiabilidad y cómo se usan éstas para especificar los requerimientos de fiabilidad mensurables;
- identificará los diferentes tipos de requerimientos de seguridad que se requieren en un sistema complejo;
- conocerá las ventajas y las desventajas de usar las especificaciones matemáticas formales de un sistema.

Contenido

12.1 Especificación de requerimientos dirigida por riesgos
12.2 Especificación de protección
12.3 Especificación de fiabilidad
12.4 Especificación de seguridad
12.5 Especificación formal

En septiembre de 1993, un avión aterrizó en el aeropuerto de Varsovia, Polonia, en medio de una tormenta. Durante nueve segundos, después del aterrizaje, no funcionaron los frenos en el sistema de frenado controlado por computadora. El sistema de frenado no reconoció que el avión había aterrizado y operó como si la aeronave aún estuviera en el aire. Una característica de seguridad de la aeronave detuvo el despliegue del sistema de empuje inverso, que desacelera la aeronave, lo cual es muy peligroso cuando el avión se encuentra en el aire. El avión salió por el extremo de la pista, golpeó un banco de tierra y se incendió.

La investigación del accidente demostró que el software del sistema de frenado operó según su especificación. Por lo tanto, no había errores en el programa. Sin embargo, la especificación del software estaba incompleta y no consideró una situación extraña, que surgió en este caso. El software funcionó, pero el sistema falló.

Esto demuestra que la confiabilidad del sistema no sólo depende de buena ingeniería, sino también requiere de la atención a los detalles cuando se derivan los requerimientos del sistema y la inclusión de requerimientos especiales de software que se ajustan para garantizar la confiabilidad y seguridad de un sistema. Estos requerimientos de confiabilidad y seguridad son de dos tipos:

1. Requerimientos funcionales, definen mecanismos de comprobación y recuperación que deben incluirse en el sistema y en las características que ofrecen protección contra fallas de sistema y ataques externos.
2. Requerimientos no funcionales, definen la confiabilidad y disponibilidad requeridas del sistema.

El punto de partida para generar requerimientos funcionales de confiabilidad y seguridad con frecuencia es un conjunto de reglas, políticas o regulaciones empresariales o de dominio de alto nivel. Se trata de requerimientos de alto nivel que tal vez se describen mejor como requerimientos “no debe”. En contraste con los requerimientos funcionales normales que delimitan la conducta del sistema, los requerimientos “no debe” definen el comportamiento del sistema que es inaceptable. Ejemplos de requerimientos “no debe” son:

> “El sistema no debe permitir que los usuarios modifiquen los permisos de acceso sobre algún archivo que no hayan creado” (seguridad).
>
> “El sistema no debe permitir la selección del modo de empuje inverso cuando la aeronave está en vuelo” (protección).
>
> “El sistema no debe permitir la activación simultánea de más de tres señales de alarma” (protección).

Estos requerimientos “no debe” no pueden implementarse directamente, sino que tienen que descomponerse en requerimientos funcionales de software más específicos, o bien, aplicarse a través de decisiones de diseño de sistema, como una decisión para usar tipos particulares de equipo en el sistema.

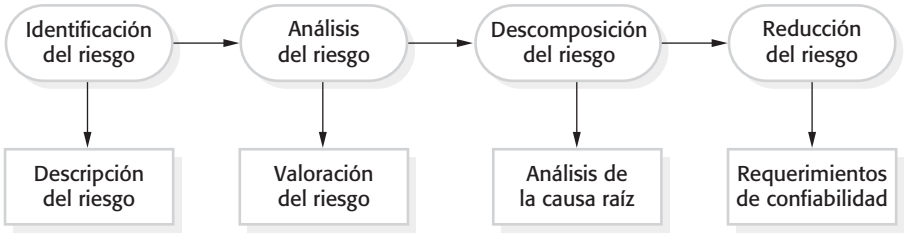


Figura 12.1 Especificación dirigida por riesgos

12.1 Especificación de requerimientos dirigida por riesgos

Los requerimientos de confiabilidad y seguridad pueden considerarse como requerimientos de protección. Especifican cómo un sistema debe protegerse a sí mismo de fallas internas, detener fallas de sistema que causen daño a su entorno, contener los accidentes o ataques del entorno que dañen el sistema, así como facilitar la recuperación en el caso de falla. Para descubrir tales requerimientos de protección, es necesario entender los riesgos al sistema y su entorno. Un enfoque dirigido por riesgos para la especificación de requerimientos toma en cuenta los eventos peligrosos que pudieran ocurrir, la probabilidad de que realmente sucedan, la posibilidad de que el daño derivará de tal evento y el grado del daño causado. En tal caso, pueden establecerse requerimientos de seguridad y confiabilidad, con base en un análisis de posibles causas de eventos peligrosos.

La especificación dirigida por riesgos es un enfoque que usan ampliamente los desarrolladores de sistemas de protección y los sistemas críticos de seguridad. Se dirigen a aquellos eventos que podrían causar más daño o que quizás ocurran con frecuencia. Es posible ignorar aquellos eventos que tienen sólo consecuencias menores o que son considerablemente inusuales. En los sistemas de protección críticos, los riesgos se asocian con peligros que podrían derivar en accidentes; en los sistemas críticos para la seguridad, los riesgos provienen de ataques internos y externos sobre un sistema cuya intención es aprovechar las posibles vulnerabilidades.

Un proceso general de especificación dirigida por riesgos (figura 12.1) incluye la comprensión de riesgos que enfrenta el sistema, el descubrimiento de sus causas raíz y la generación de requerimientos para gestionar dichos riesgos. Las etapas de este proceso son:

1. *Identificación del riesgo* Se identifican los riesgos potenciales al sistema. Éstos dependen del entorno en que se usa el sistema. Pueden surgir riesgos de interacciones entre el sistema y las condiciones extrañas de su entorno operacional. El accidente de Varsovia descrito anteriormente ocurrió cuando vientos cruzados generados durante una tormenta produjeron (inusualmente) que el avión se inclinara de modo que aterrizó sobre una rueda en lugar de dos.

2. *Análisis y clasificación del riesgo* Cada riesgo se considera por separado. Aquéllos potencialmente serios y no improbables se seleccionan para un mayor análisis. En

esta etapa, los riesgos pueden eliminarse porque es improbable que surjan o porque no se pueden detectar con el software (por ejemplo, una reacción alérgica al sensor en el sistema de bomba de insulina).

3. *Descomposición del riesgo* Cada riesgo se analiza para descubrir las causas raíz potenciales de dicho riesgo. Dichas causas son las razones por las que es posible que falle un sistema. Pueden ser errores de software, hardware o vulnerabilidades inherentes que son consecuencia de decisiones de diseño del sistema.

4. *Reducción del riesgo* Se hacen proposiciones de formas para reducir o eliminar los riesgos identificados. Ello contribuye con los requerimientos de confiabilidad del sistema que definen las defensas contra el riesgo y cómo se manejará éste.

Para sistemas grandes, el análisis de riesgo puede estructurarse en fases (Leveson, 1995), donde cada fase considera diferentes tipos de riesgos:

1. Análisis preliminar del riesgo, en el que se identifican los principales riesgos del entorno del sistema. Tales análisis son independientes de la tecnología utilizada para el desarrollo del sistema. La meta del análisis preliminar del riesgo es el desarrollo de un conjunto inicial de requerimientos de seguridad y confiabilidad para el sistema.

2. Análisis de riesgo de ciclo de vida, que tiene lugar durante el desarrollo del sistema y se dirige principalmente a los riesgos surgidos por decisiones de diseño del sistema. Diferentes tecnologías y arquitecturas de sistema contienen sus propios riesgos asociados. En esta etapa, se deben extender los requerimientos para protegerse contra estos riesgos.

3. Análisis de riesgo operativo, el cual se preocupa por la interfaz de usuario del sistema y los riesgos de error del operador. De nuevo, una vez que se toman las decisiones sobre el diseño de interfaz del usuario, es posible que deban agregarse más requerimientos de protección.

Estas fases son necesarias porque es imposible tomar todas las decisiones de confiabilidad y seguridad sin información completa de la implementación del sistema. Los requerimientos de seguridad y confiabilidad en particular se ven afectados por la elección de la tecnología y las decisiones de diseño. Tienen que incluirse comprobaciones del sistema para garantizar que los componentes de terceras partes operen de manera correcta. Los requerimientos de seguridad tal vez deban modificarse al entrar en conflicto con las características de seguridad que brinda un sistema comercial.

Por ejemplo, un requerimiento de seguridad sería que los usuarios deban identificarse frente a un sistema mediante una frase de paso (*passphrase*) en vez de una contraseña o palabra de paso (*password*). Las passphrases se consideran más seguras que las passwords. Son más difíciles de descifrar por un atacante o de descubrir a través de un sistema automatizado de rompimiento de contraseñas. Sin embargo, si se toma una decisión de usar un sistema existente que sólo soporte autenticación basada en password, en tal caso no puede sostenerse este requerimiento de seguridad. Entonces sería necesario incluir funcionalidad adicional en el sistema, con la finalidad de compensar los riesgos crecientes de usar passwords en lugar de passphrases.

El estándar IEC para el manejo de la protección

La IEC (Comisión Electrotécnica Internacional) definió un estándar para manejo de la seguridad de los sistemas de protección (es decir, sistemas que tienen la intención de activar salvaguardias cuando surge alguna situación peligrosa). Un ejemplo de un sistema de protección es aquel que detiene automáticamente un tren si pasa por una señal roja. Este estándar incluye guías extensas acerca del proceso de especificación de seguridad.

http://www.SoftwareEngineering-9.com/Web/SafetyLifeCycle/

12.2 Especificación de protección

En los sistemas críticos de protección, las fallas llegan a afectar el entorno del sistema y a causar lesiones o muerte a los individuos en este ambiente. La preocupación principal de la especificación de protección es identificar los requerimientos que reducirán la probabilidad de que ocurran tales fallas de sistema. Los requerimientos de protección son básicamente requerimientos de seguridad y no se interesan por la operación normal del sistema. Podrían especificar que el sistema debe desactivarse de modo que se conserve la protección. Por lo tanto, al derivar requerimientos de protección, se necesita encontrar un equilibrio aceptable entre seguridad y funcionalidad para evitar la sobreprotección. No hay razón para construir un sistema altamente seguro si no resulta efectivo en cuanto a costo.

Según la discusión en el capítulo 10, los sistemas críticos de protección usan una terminología especializada, en la que un peligro es algo que podría (aunque no necesariamente) derivar en muerte o lesión a una persona, y un riesgo es la probabilidad de que el sistema entre en un estado peligroso. Por consiguiente, la especificación de protección se enfoca usualmente en los riesgos que surgen en una situación dada, y los eventos que conducen a dichos peligros.

Las actividades en el proceso de especificación general basado en riesgos, mostrado en la figura 12.1, se mapean en el proceso de especificación de protección del siguiente modo:

1. *Identificación del riesgo* En la especificación de protección, éste es el proceso de identificación del peligro que identifica los riesgos que amenazarían al sistema.
2. *Análisis de riesgo* Es un proceso de valoración del peligro que permite determinar qué riesgos son los más peligrosos y/o los que tienen mayor probabilidad de ocurrir. Éstos deben priorizarse al derivar los requerimientos de protección.
3. *Descomposición del riesgo* Este proceso se enfoca en el descubrimiento de los eventos que quizá conduzcan a que ocurra un peligro. En la especificación de protección, el proceso se conoce como análisis de peligro.
4. *Reducción del riesgo* Este proceso se basa en el resultado del análisis del peligro y lleva a la identificación de requerimientos de protección. Se puede inclinar por el aseguramiento de que no surja un peligro o conduzca a un accidente o que, si ocurre, el daño asociado sea mínimo.

12.2.1 Identificación de peligro

En los sistemas críticos de protección, los riesgos principales provienen de los peligros que conducirían a un accidente. Es posible tratar el problema de identificación del riesgo al considerar los diferentes tipos, como los físicos, eléctricos, biológicos, de radiación, de falla de servicio, etcétera. Entonces, cada una de estas clases puede analizarse para descubrir peligros específicos que podrían ocurrir. También deben identificarse aquellas posibles combinaciones de riesgos que sean potencialmente peligrosas.

El sistema de bomba de insulina usado como ejemplo en capítulos anteriores es un sistema crítico de protección, porque su falla ocasionaría al usuario del sistema lesiones o incluso la muerte. Los accidentes que posiblemente ocurran cuando se usa esta máquina incluyen que el usuario sufra las consecuencias a largo plazo de un inadecuado control de azúcar en la sangre (problemas oculares, cardiacos y renales); disfunción cognitiva como resultado de bajos niveles de azúcar en la sangre; o la aparición de algunas otras condiciones médicas, como una reacción alérgica.

Algunos de los peligros en el sistema de bomba de insulina son:

- cálculo de sobredosis de insulina (falla de servicio);
- cálculo de subdosis de insulina (falla de servicio);
- falla del sistema de monitorización del hardware (falla de servicio);
- falla de energía debido a batería baja (eléctrico);
- interferencia eléctrica con otro equipo médico como un marcapasos cardiaco (eléctrico);
- mal contacto entre el sensor y el accionador causado por un ajuste incorrecto (físico);
- partes de máquina que se rompen en el cuerpo del paciente (físico);
- infección provocada por introducción de máquina (biológico);
- reacción alérgica a los materiales o la insulina usada en la máquina (biológico).

Los ingenieros experimentados, que trabajan con expertos de dominio y consejeros de seguridad profesional, identifican los peligros a partir de la experiencia y desde un análisis del dominio de aplicación. Pueden utilizarse técnicas de trabajo grupal, como lluvia de ideas, donde un grupo de individuos intercambian ideas. Para el sistema de bomba de insulina, las personas que intervendrían incluyen médicos, físicos médicos e ingenieros y diseñadores de software.

Los peligros relacionados con el software se relacionan con frecuencia con las fallas para entregar un servicio de sistema, o con las fallas de los sistemas de monitorización y protección. Los sistemas de monitorización y protección se incluyen en un dispositivo para detectar condiciones, como niveles bajos de batería, que conducirían a una falla del dispositivo.

12.2.2 Valoración del peligro

El proceso de valoración del peligro se enfoca en comprender la probabilidad de que sobrevenga un peligro y las consecuencias de que ocurriera un accidente o incidente asociado con dicho peligro. Es necesario hacer este análisis para comprender si un peligro

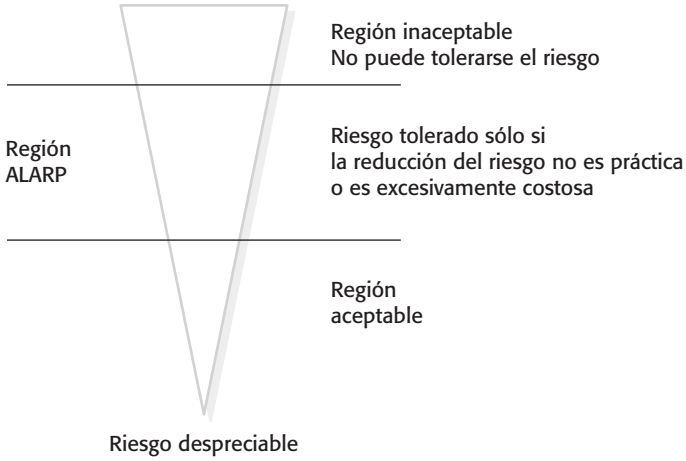


Figura 12.2 El triángulo de riesgo

constituye una seria amenaza al sistema o el entorno. El análisis proporciona también una base para decidir sobre cómo manejar el riesgo asociado.

Para cada peligro, el resultado del proceso de análisis y clasificación es un enunciado de aceptabilidad. Esto se expresa en términos de riesgo, donde el riesgo considera la probabilidad de un accidente y sus consecuencias. Existen tres categorías que se utilizan en la valoración del riesgo:

1. Los riesgos intolerables en los sistemas críticos de protección son aquellos que amenazan la vida humana. El sistema debe diseñarse de forma que tales peligros no puedan surgir o, si es el caso, las características del sistema garantizarán que éstos se detectan antes de que se produzca un accidente. En cuanto a la bomba de insulina, un riesgo intolerable es que se suministre una sobredosis.

2. Los riesgos "tan bajos como sea razonablemente práctico" (ALARP, por las siglas de *As Low As Reasonably Practical*) son aquellos que tienen consecuencias menos serias o que, aun cuando son graves, tienen una muy baja probabilidad de ocurrencia. El sistema debe diseñarse de modo que se minimice la posibilidad de que se suscite un accidente debido a un peligro, sujeto a otras consideraciones tales como costo y entrega. Un riesgo ALARP para una bomba de insulina sería la falla del sistema de monitorización del hardware. Las consecuencias de esto son, en el peor contexto, una subdosis de insulina a corto plazo. Aunque ésta es una situación que no conduciría a un grave accidente.

3. Los riesgos aceptables son aquellos en que los accidentes asociados derivan por lo general en un daño menor. Los diseñadores del sistema deben dar todos los pasos necesarios para reducir los riesgos "aceptables", en tanto esto no aumente los costos, el tiempo de entrega ni otros atributos no funcionales del sistema. Un riesgo aceptable en el caso de la bomba de insulina puede ser el de una reacción alérgica que se presente en el usuario. Con frecuencia, esto sólo causa irritación menor de la piel. No valdría la pena usar materiales especiales más costosos en el dispositivo para reducir tal riesgo.

La figura 12.2 (Brazendale y Bell, 1994), desarrollada para sistemas críticos de seguridad, muestra estas tres regiones. La forma del diagrama refleja los costos para garantizar

Peligro identificado	Probabilidad del riesgo	Severidad del accidente	Riesgo estimado	Aceptabilidad
1. Cálculo de sobredosis insulina	Media	Alta	Alto	Intolerable
2. Cálculo de subdosis insulina	Media	Baja	Bajo	Aceptable
3. Falla del sistema de monitorización del hardware	Media	Media	Bajo	ALARP
4. Falla de energía	Alta	Baja	Bajo	Aceptable
5. Máquina ajustada incorrectamente	Alta	Alta	Alto	Intolerable
6. Máquina que se descompone en el paciente	Baja	Alta	Medio	ALARP
7. Máquina que causa infección	Media	Media	Medio	ALARP
8. Interferencia eléctrica	Baja	Alta	Medio	ALARP
9. Reacción alérgica	Baja	Baja	Bajo	Aceptable

Figura 12.3 Clasificación del riesgo para la bomba de insulina

que los riesgos no deriven en incidentes o accidentes. El costo del diseño del sistema para lidiar con el riesgo se indica mediante el ancho del triángulo. Los costos más altos se generan por los riesgos en la parte superior del diagrama, los costos más bajos, por los riesgos en el vértice del triángulo.

Las fronteras entre las regiones en la figura 12.2 no son técnicas, sino más bien dependen de factores sociales y políticos. Con el tiempo, la sociedad se ha vuelto más reacia al riesgo, de modo que las fronteras se han recorrido hacia abajo. Aunque los costos financieros de aceptar riesgos y pagar por cualquier accidente que resulte podrían ser menores que los de la prevención de accidentes, la opinión pública preferiría que se gaste dinero para reducir la probabilidad de un accidente de sistema y, por lo tanto, se incurre en costos adicionales.

Por ejemplo, sería más barato que una compañía limpie la contaminación que ocurra en alguna muy rara ocasión, en vez de instalar sistemas para prevenir la contaminación. Sin embargo, en virtud de que el público y la prensa no tolerarán tales accidentes, ya no es aceptable reparar el daño en vez de prevenir el accidente. Tales eventos conducirían también a una reclasificación del riesgo. Por ejemplo, los riesgos que se consideraban improbables (y por ello en la región ALARP) se reclasificarían como intolerables debido a eventos, como los ataques terroristas o los accidentes que han ocurrido.

La valoración del peligro implica estimar la probabilidad del peligro y la severidad del riesgo. Por lo general esto es difícil, pues los peligros y los accidentes son inusuales, de modo que los ingenieros implicados quizá no tengan experiencia directa de incidentes o accidentes previos. Las probabilidades y severidades se asignan mediante términos relativos como “probable”, “improbable”, “raro” y “alto”, “medio” y “bajo”. Sólo es posible cuantificar estos términos si para un análisis estadístico está disponible suficiente información de los accidentes e incidentes.

La figura 12.3 muestra una clasificación de riesgo sobre los peligros identificados en la sección previa para el sistema de entrega de insulina. Los peligros que se relacionan con el cálculo incorrecto de insulina se separaron en sobredosis y subdosis de insulina. Una sobredosis de insulina es potencialmente más grave que una subdosis de insulina a corto plazo. La sobredosis de insulina podría derivar en disfunción cognitiva, coma y, a final de cuentas, en la muerte. La subdosis de insulina conduce a altos niveles de azúcar en la sangre. A corto plazo, causa fatiga, pero sus consecuencias no son muy graves; sin embargo, a largo plazo, conduciría a severos problemas cardiacos, renales y oculares.

Los peligros 4 a 9 en la figura 12.3 no se relacionan con software; sin embargo, éste desempeña un papel en la detección de peligros. El software que monitoriza el hardware debe monitorizar el estado del sistema y advertir acerca de problemas potenciales. Con frecuencia, las advertencias permitirán la detección del peligro antes de que ocurra un accidente. Los ejemplos de peligros que pueden detectarse son la falla de energía, que se descubre al monitorizar la batería, y la ubicación incorrecta de la máquina, que suele detectarse al monitorizar señales del sensor de azúcar en la sangre.

Desde luego, el software de monitorización en el sistema se relaciona con la protección. La falla para detectar un peligro podría derivar en un accidente. Si el sistema de monitorización falla, pero el hardware funciona de manera correcta, entonces ésta no es una falla grave. No obstante, si el sistema de monitorización falla y, en consecuencia, la falla del hardware no se detecta, esto tendría consecuencias más drásticas.

12.2.3 Análisis del peligro

El análisis del peligro es el proceso que descubre las causas raíz de los peligros en un sistema de protección crítico. Su meta es detectar qué eventos o combinaciones de eventos causarían una falla de sistema que derive en un peligro. Para hacerlo, se puede usar un enfoque descendente o uno ascendente. Las técnicas deductivas descendentes, que tienden a ser más fáciles de usar, comienzan con el peligro y trabajan con éste hasta la posible falla de sistema. Las técnicas inductivas ascendentes comienzan con una falla de sistema propuesta e identifican qué peligros resultarían por dicha falla.

Se han planteado varias técnicas como posibles enfoques para la descomposición o el análisis del peligro, las cuales resume Storey (1996). Incluyen revisiones y listas de verificación, técnicas formales como el análisis de red de Petri (Peterson, 1981), lógica formal (Jahanian y Mok, 1986) y análisis de árbol de fallas (Leveson y Stolzy, 1987; Storey, 1996). Como no se tiene espacio para estudiar aquí todas esas técnicas, se examinará un enfoque ampliamente usado para el análisis de peligro basado en árboles de fallas. Esta técnica es bastante sencilla de entender sin un conocimiento especializado del dominio.

Para hacer un análisis de árbol de fallas, comience con los peligros identificados. Para cada peligro se puede trabajar entonces en retroceso, con la finalidad de descubrir las posibles causas de dicho peligro. Coloque el peligro en la raíz del árbol e identifique los estados del sistema que podrían conducir a tal peligro. Para cada uno de dichos estados, es posible identificar entonces más estados de sistema que conduzcan hacia ellos. Continúe con esta descomposición hasta que llegue a las causas raíz del riesgo. Los peligros que sólo pueden surgir a partir de una combinación de causas raíz son, por lo general, menos probables de conducir a un accidente, que aquellos peligros con una sola causa raíz.

La figura 12.4 es un árbol de fallas sobre los peligros relacionados con software en el sistema de entrega de insulina que podrían llevar a administrar una dosis incorrecta de insulina. En este caso, se fusionaron la subdosis de insulina y la sobredosis de insulina

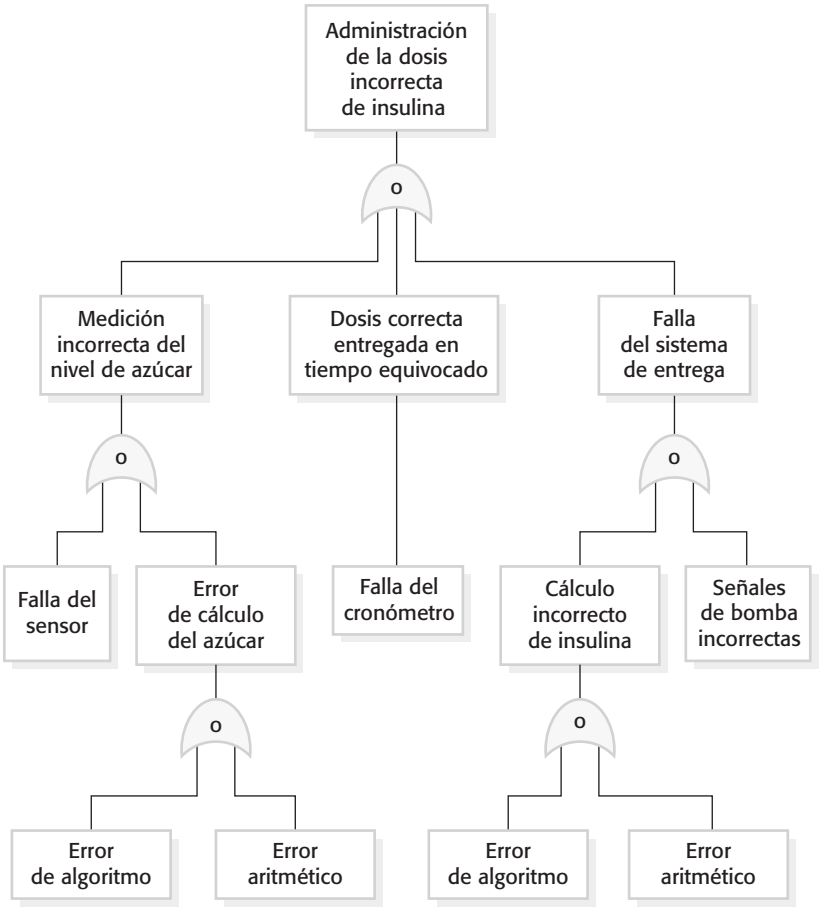


Figura 12.4 Ejemplo de árbol de fallas

en un solo peligro: "administración de la dosis incorrecta de insulina". Esto reduce el número de árboles de fallas requeridos. Desde luego, cuando se especifica cómo debe reaccionar el software ante este peligro, se debe distinguir entre una subdosis y una sobredosis de insulina. Como se comentó, no son igualmente graves: a corto plazo, una sobredosis es el peligro más severo.

En la figura 12.4 se observa que:

1. Hay tres condiciones que podrían conducir a la administración de una dosis incorrecta de insulina. El nivel de azúcar en la sangre pudo medirse de manera incorrecta, de modo que el requerimiento de insulina se calculó con una entrada errónea. El sistema de entrega quizá no responda acertadamente a los comandos que especifican la cantidad de insulina inyectada. O bien, la dosis puede calcularse de manera correcta, pero ésta se entrega muy pronto o demasiado tarde.
2. La rama izquierda del árbol de fallas, que se ocupa de la medición incorrecta del nivel de azúcar en la sangre, busca cómo podría ocurrir esto. Ello sucedería porque

falló el sensor que proporciona una entrada para calcular el nivel de azúcar, o porque se realizó de manera incorrecta el cálculo del nivel de azúcar en la sangre. El nivel de azúcar se calcula a partir de la medición de cierto parámetro, como la conductividad de la piel. El cálculo incorrecto podría derivar en un algoritmo incorrecto o en un error aritmético producto del uso de números con punto flotante.

3. La rama central del árbol se interesa por los problemas de temporización y concluye que éstos sólo pueden resultar de la falla del cronómetro del sistema.

4. La rama derecha del árbol, que se inclina por la falla del sistema de entrega, examina posibles causas de esta falla. Esto surgiría de un cálculo incorrecto del requerimiento de insulina, o bien, de una falla al enviar las señales correctas a la bomba que entrega la insulina. De nuevo, un cálculo inexacto puede provocar una falla de algoritmo o errores aritméticos.

Los árboles de fallas se usan también para identificar problemas potenciales de hardware. Los árboles de fallas de hardware pueden brindar comprensión de los requerimientos para que el software detecte y, quizá, corrija dichos problemas. Por ejemplo, las dosis de insulina no se administran a una frecuencia muy alta, ni más de dos o tres veces por hora y, en ocasiones, con menos periodicidad que esto. Por consiguiente, la capacidad del procesador está disponible para operar programas de diagnóstico y autoverificación. Quizá se descubran errores de hardware, como los de sensor, bomba o cronómetro, y emitirse advertencias antes de que tengan un grave efecto sobre el paciente.

12.2.4 Reducción del riesgo

Una vez identificados los riesgos potenciales y sus causas raíz, entonces se podrán derivar requerimientos de seguridad que administren los riesgos y garanticen que no ocurran los incidentes o accidentes. Existen tres posibles estrategias por utilizar:

1. *Evitar el peligro* El sistema se diseña de modo que no pueda ocurrir el peligro.

2. *Detectar y eliminar el peligro* El sistema se diseña de forma que los peligros se detecten y neutralicen antes de que suceda un accidente.

3. *Limitar el daño* El sistema se diseña de manera que se minimicen las consecuencias de un accidente.

Normalmente, los diseñadores de sistemas críticos combinan dichos enfoques. En un sistema crítico de seguridad, los peligros intolerables pueden manejarse al disminuir su probabilidad y agregar un sistema de protección que brinde un respaldo de seguridad. Por ejemplo, en un sistema de control de planta química, el sistema tratará de detectar y evitar una presión excesiva en el reactor. Sin embargo, también podría haber un sistema de protección independiente que monitorice la presión y abra una válvula de desfogue o alivio al detectarse presión alta.

En el sistema de entrega de insulina, un “estado seguro” es uno desactivado, donde no se inyecta insulina. Durante un breve periodo, no amenaza la salud del diabético. En el

RS1: El sistema no debe entregar una dosis individual de insulina mayor que una dosis máxima especificada para un usuario del sistema.
RS2: El sistema no debe entregar una dosis diaria almacenada de insulina mayor que una dosis diaria máxima especificada para un usuario del sistema.
RS3: El sistema debe incluir una instalación de diagnóstico de hardware que tiene que ejecutarse al menos cuatro veces por hora.
RS4: El sistema debe incluir un manipulador de excepción para todas las excepciones que se identifiquen en la tabla 3.
RS5: La alarma audible tiene que activarse cuando se descubra cualquier anomalía de hardware o software, y hay que desplegar un mensaje diagnóstico, como se define en la tabla 4.
RS6: En el caso de una alarma, la entrega de insulina debe suspenderse hasta que el usuario reestablezca el sistema y desactive las alarmas.

Figura 12.5 Ejemplos de requerimientos de seguridad

caso de las fallas del software que pudieran conducir a una dosis incorrecta de insulina, se consideran las siguientes posibles "soluciones":

1. *Error aritmético* Éste puede ocurrir cuando un cálculo aritmético causa una falla de representación. La especificación tiene que identificar todos los posibles errores aritméticos que ocurrirían, e informar que debe incluirse un manipulador de excepción para cada posible error. La especificación tiene que establecer la acción a tomar en cada uno de dichos errores. La acción de seguridad predeterminada es desactivar el sistema de entrega y activar una alarma de advertencia.
2. *Error algorítmico* Ésta es una situación más difícil, pues no hay una clara excepción del programa que deba manejarse. Podría detectarse este tipo de error al comparar la dosis de insulina requerida calculada con la dosis entregada anteriormente. Si es mucho mayor, esto significaría que la cantidad se calculó de forma incorrecta. El sistema también puede hacer un seguimiento de la secuencia de dosis. Después de la entrega de algunas dosis por arriba del promedio, tiene que emitirse una advertencia y limitarse las dosis posteriores.

En la figura 12.5 se muestran algunos de los requerimientos de seguridad que resultan para el software de bomba de insulina. Se trata de requerimientos de usuarios y, por supuesto, se expresarían con más detalle en la especificación de requerimientos del sistema. En la misma figura, las referencias a las tablas 3 y 4, que no se muestran aquí, se relacionan con las tablas incluidas en el documento de requerimientos.

12.3 Especificación de fiabilidad

Como se estudió en el capítulo 10, la fiabilidad global de un sistema depende de la fiabilidad del hardware, la fiabilidad del software y la fiabilidad de los operadores del sistema. Se debe tomar en cuenta el software del sistema y, además, de incluir requeri-

mientos que compensen la falla de software. También puede haber requerimientos relacionados de fiabilidad para ayudar a detectar y recuperar fallas de hardware y errores del operador.

La fiabilidad es diferente de la seguridad y la protección, pues se considera un atributo mensurable del sistema. Esto es, se puede especificar y comprobar si se logró la fiabilidad requerida. Por ejemplo, un requerimiento de fiabilidad sería que las fallas de sistema que requieran un reinicio (reboot) no deben ocurrir más de una vez por semana. Cuando tal falla se presenta, conviene anotarlo en una bitácora y comprobar si se logró el nivel de fiabilidad requerido. Es caso contrario, se modifica el requerimiento de fiabilidad, o bien, se envía una petición de cambio para enfrentar los problemas subyacentes del sistema. Se puede aceptar un nivel más bajo de fiabilidad debido a los costos de cambiar el sistema para mejorar la fiabilidad, o porque al corregir el problema haya efectos colaterales adversos, como un rendimiento o una producción totales más bajos.

En contraste, seguridad y protección tratan de evitar situaciones indeseables, en vez de especificar un "nivel" deseado de seguridad o protección. Incluso sería inaceptable una de estas situaciones durante la vida de un sistema y, si ocurre, tienen que realizarse cambios al sistema. No tiene sentido hacer enunciados como: "Las fallas del sistema podrían dar como resultado menos de 10 lesiones por año". Tan pronto como suceda una lesión, debe corregirse el problema del sistema.

Por consiguiente, los requerimientos de fiabilidad son de dos tipos:

1. Requerimientos no funcionales, que definen el número de fallas aceptables durante el uso normal del sistema, o el tiempo en que el sistema no está disponible para su uso. Se trata de requerimientos de fiabilidad cuantitativos.
2. Requerimientos funcionales, que definen las funciones del sistema y el software que evitan, detectan o toleran fallas del software y, de ese modo, aseguran que esto no conduzca a fallas de sistema.

Los requerimientos de fiabilidad cuantitativa conducen a requerimientos de sistema funcionales relacionados. Para lograr cierto nivel de fiabilidad requerido, los requerimientos funcionales y de diseño del sistema deben especificar las fallas a detectar y las acciones que deben tomarse para garantizar que éstas no conduzcan a fallas de sistema.

En la figura 12.1 se muestra que el proceso de especificación de fiabilidad puede basarse en el proceso general de especificación dirigido por riesgo:

1. *Identificación del riesgo* En esta etapa se examinan los tipos de fallas de sistema que originarían pérdidas económicas de cierto tipo. Por ejemplo, cuando un sistema de comercio electrónico no está disponible de tal modo que los clientes no pueden realizar pedidos, o una falla que corrompa datos y requiera tiempo para restaurar la base de datos del sistema desde un respaldo y volver a operar transacciones que se hayan procesado. La lista de posibles tipos de fallas, que se muestra en la figura 12.6, sirve como punto de partida en la identificación del riesgo.
2. *Análisis del riesgo* Implica la estimación de los costos y las consecuencias de diferentes tipos de fallas de software y selecciona para un análisis ulterior las fallas de graves consecuencias.

Tipo de falla	Descripción
Pérdida de servicio	El sistema no está disponible y no puede entregar sus servicios a los usuarios. Esto se divide en pérdida de servicios críticos y pérdida de servicios no críticos; evidentemente, son menores las consecuencias de una falla en los servicios no críticos, que las consecuencias de una falla de servicios críticos.
Entrega incorrecta de servicio	El sistema no entrega correctamente un servicio a los usuarios. De nuevo, esto podría especificarse en términos de errores menores y mayores, o bien, en errores en la entrega de servicios críticos y no críticos.
Corrupción de sistema y de datos	La falla del sistema causa daño al sistema en sí o a sus datos. Por lo general, aunque no necesariamente, esto estará en conjunción con otros tipos de fallas.

Figura 12.6 Tipos de fallas del sistema

3. *Descomposición de riesgo* En esta etapa, se realiza un análisis de la causa raíz de las graves y probables fallas de sistema. Sin embargo, esto sería casi imposible en la etapa de requerimientos, pues las causas raíz suelen depender de decisiones de diseño del sistema. Tal vez usted tenga que regresar a esta actividad durante el diseño y el desarrollo.

4. *Reducción del riesgo* En esta etapa, deben generarse especificaciones cuantitativas de fiabilidad que establezcan las probabilidades aceptables de los diferentes tipos de fallas. Desde luego, hay que tomar en cuenta los costos de las fallas. Se pueden usar diferentes probabilidades para distintos servicios del sistema, así como generar requerimientos de fiabilidad funcionales. De nuevo, esto tal vez deba esperar hasta que se tomen las decisiones de diseño del sistema. No obstante, como se estudia en la sección 12.3.2, en ocasiones es difícil crear especificaciones cuantitativas. Quizás usted sólo pueda identificar los requerimientos funcionales de fiabilidad.

12.3.1 Métricas de fiabilidad

En términos generales, la fiabilidad puede especificarse como una probabilidad de que una falla del sistema ocurrirá cuando un sistema está en uso dentro de un entorno operacional especificado. Si está dispuesto a aceptar, por ejemplo, que pueda fallar una de las 1,000 transacciones, entonces especificará la probabilidad de falla como 0.001. Claro, esto no significa que verá una falla en cada 1,000 transacciones. Quiere decir que, si observa N mil transacciones, el número de fallas que observe debe estar alrededor de N. Es posible corregir esto para diferentes tipos de fallas o diferentes partes del sistema. Puede decidir que los componentes críticos deben tener una probabilidad más baja de falla que los no críticos.

Existen dos importantes métricas que se usan para especificar la fiabilidad, además de una métrica adicional que se utiliza para especificar el atributo de sistema relacionado con la disponibilidad. La elección de métrica depende del tipo de sistema especificado y de los requerimientos del dominio de la aplicación. Las métricas son:

1. *Probabilidad de falla a pedido (POFOD,* por las siglas de *Probability Of Failure On Demand)* Si usa esta métrica, defina la probabilidad de que la demanda por un servicio de un sistema derive en una falla del sistema. De este modo, POFOD = 0.001

Disponibilidad	Explicación
0.9	El sistema está disponible el 90% del tiempo. Esto significa que, durante un periodo de 24 horas (1,440 minutos), el sistema no estará disponible por 144 minutos.
0.99	Durante un periodo de 24 horas, el sistema no estará disponible por 14.4 minutos.
0.999	El sistema no estará disponible durante 84 segundos, en un periodo de 24 horas.
0.9999	El sistema no estará disponible durante 8.4 segundos en un periodo de 24 horas. Casi un minuto por semana.

Figura 12.7 Especificación de disponibilidad

significa que hay una probabilidad de 1/1,000 de que ocurra una falla al hacer una petición.

2. *Tasa de ocurrencia de fallas (ROCOF,* por las siglas de *Rate Of Occurrence Of Failures)* Esta métrica establece el número probable de fallas de sistema que se observan en relación con cierto tiempo (por ejemplo, una hora), o el número de ejecuciones del sistema. En el ejemplo anterior, la ROCOF es 1/1,000. El recíproco de la ROCOF es el tiempo medio para la falla (MTTF, por las siglas de *Main Time To Failure*), que a veces se usa como una métrica de fiabilidad. El MTTF es el promedio de unidades de tiempo entre las fallas observadas de sistema. Por lo tanto, una ROCOF de dos fallas por hora significa que el tiempo medio de la falla es de 30 minutos.

3. *Disponibilidad (AVAIL)* La disponibilidad de un sistema refleja la capacidad de entregar servicios cuando se le solicitan. AVAIL es la probabilidad de que un sistema esté en operación cuando se haga una demanda por servicio. En consecuencia, una disponibilidad de 0.9999 significa que, en promedio, el sistema estará disponible el 99.99% del tiempo de operación. La figura 12.7 muestra qué significan en la práctica los diferentes niveles de disponibilidad.

La POFOD debe usarse como una métrica de fiabilidad en situaciones donde una falla sobre demanda podría conducir a una falla grave del sistema. Esto se aplica sin importar la frecuencia de las demandas. Por ejemplo, un sistema de protección que monitoriza un reactor químico, y desactiva la reacción cuando se sobrecalienta, debe especificar su fiabilidad mediante POFOD. En general, son raras las demandas a un sistema de protección, ya que el sistema es la última línea de defensa, una vez que fallan todas las demás estrategias de recuperación. Por ello, una POFOD de 0.001 (1 falla en 1,000 demandas) quizá parezca riesgosa, pero si en su vida sólo hay dos o tres demandas del sistema, entonces probablemente nunca verá una falla de sistema.

La ROCOF es la métrica más adecuada para usar en situaciones donde las demandas al sistema se hacen con regularidad en vez de esporádicamente. Por ejemplo, en un sistema que maneja una gran cantidad de transacciones, usted puede especificar una ROCOF de 10 fallas por día. Esto significa que está dispuesto a aceptar que un promedio de 10 transacciones por día no se completen exitosamente, y tendrán que cancelarse. O bien, es posible especificar una ROCOF como el número de fallas por 1,000 transacciones.

Si es importante el tiempo absoluto entre fallas, puede especificar la fiabilidad como el tiempo medio entre fallas. Por ejemplo, si especifica la fiabilidad requerida para un sistema con transacciones grandes (como un sistema de diseño asistido por computadora),

debe especificar la fiabilidad con un tiempo medio largo para la falla. El MTTF debe ser mucho más largo que el tiempo promedio que un usuario trabaja en sus modelos sin guardar sus resultados. Esto podría representar que sería improbable que los usuarios pierdan su trabajo por una falla del sistema en cualquier sesión.

Para valorar la fiabilidad de un sistema, debe capturar datos sobre su operación. Los datos requeridos incluyen:

1. El número de fallas de sistema dado un número de peticiones por servicios del sistema. Esto se usa para medir la POFOD.
2. El tiempo o número de transacciones entre fallas de sistema, más el tiempo total transcurrido o el número total de transacciones. Esto se usa para medir la ROCOF y el MTTF.
3. El tiempo de reparación o reanudación después de una falla de sistema que conduzca a pérdida de servicio. Se utiliza para medir la disponibilidad. La disponibilidad no sólo depende del tiempo entre fallas, sino también del tiempo requerido para hacer que el sistema esté nuevamente en operación.

Las unidades de tiempo que pueden usarse son tiempo de calendario, tiempo de procesador o una unidad discreta como el número de transacciones. En los sistemas que pasan buena parte de su tiempo esperando una respuesta a una petición de servicio, como los sistemas de conmutación telefónica, la unidad de tiempo que debe usarse es tiempo de procesador. Si se usa tiempo calendario, entonces esto incluirá el tiempo cuando el sistema no hacía nada.

Se debe usar tiempo calendario para los sistemas que estén en operación continua. Los sistemas de monitorización, como los sistemas de alarma y otros tipos de sistemas de control de proceso se sitúan en esta categoría. Los sistemas que procesan transacciones como los cajeros automáticos o los sistemas de reservación de las aerolíneas tienen cargas variables sobre ellos, las cuales dependen de la hora del día. En estos casos, la unidad de "tiempo" que se usa podría ser el número de transacciones (es decir, ROCOF sería el número de transacciones fallidas por cada N mil transacciones).

12.3.2 Requerimientos de fiabilidad no funcionales

Los requerimientos de fiabilidad no funcionales son especificaciones cuantitativas de la fiabilidad y disponibilidad requeridas de un sistema, calculadas mediante el uso de una de las métricas descritas en la sección anterior. Las especificaciones cuantitativas de fiabilidad y disponibilidad se han empleado durante muchos años en los sistemas críticos de seguridad, y rara vez se utilizan en sistemas empresariales críticos. Sin embargo, conforme más compañías demandan servicios de sus sistemas las 24 horas al día los siete días de la semana, es factible que tales técnicas se utilicen cada vez más.

Existen varias ventajas para derivar especificaciones de fiabilidad cuantitativas:

1. El proceso de decidir qué nivel de fiabilidad se requiere ayuda a clarificar qué necesitan realmente los participantes. Contribuye a que éstos comprendan que existen diferentes tipos de fallas de sistema, y deja en claro que los niveles superiores de fiabilidad son muy costosos de lograr.

2. Proporciona una base para valorar cuándo dejar de probar un sistema. Se detiene cuando el sistema obtiene su nivel requerido de fiabilidad.
3. Es un medio para valorar diferentes estrategias de diseño, cuya intención sea mejorar la fiabilidad del sistema. Se puede hacer un juicio sobre cómo cada estrategia llega a conducir a niveles requeridos de fiabilidad.
4. Si un regulador tiene que aprobar un sistema antes de entrar en servicio (por ejemplo, están regulados todos los sistemas críticos para la seguridad del vuelo en una aeronave), entonces es importante la evidencia de que se alcanzó un objetivo de fiabilidad requerido para la certificación del sistema.

Para establecer el nivel requerido de fiabilidad del sistema, hay que considerar las pérdidas asociadas que resultarían de una falla del sistema. Éstas no son simplemente pérdidas financieras para una empresa, sino también pérdidas en la reputación. Esto último significa que los clientes se irán a algún otro lado. Aunque las pérdidas a corto plazo por una falla de sistema suelen ser relativamente pequeñas, las pérdidas a plazo más largo serían mucho más significativas. Por ejemplo, si usted trata de acceder a un sitio de comercio electrónico y se da cuenta de que éste no se encuentra disponible, entonces en vez de esperar a que el sistema esté disponible tratará de buscar lo que quiere en algún otro lado. Si esto ocurre más de una vez, seguramente no comprará de nuevo en dicho sitio.

El problema con la especificación de la fiabilidad usando métricas como POFOD, ROCOF y AVAIL es que es posible sobreespecificar la fiabilidad y, por ende, incurrir en altos costos de desarrollo y validación. La razón es que los participantes del sistema descubren que es difícil traducir su experiencia práctica en especificaciones cuantitativas. Pueden considerar que una POFOD de 0.001 (1 falla en 1,000 demandas) representa un sistema relativamente poco fiable. Sin embargo, como se explicó, si las demandas por un servicio son escasas, en realidad representa un nivel de fiabilidad muy alto.

Si la fiabilidad se especifica como una métrica, evidentemente es importante valorar que se logró el nivel de fiabilidad requerido. Esta valoración se hace como parte de las pruebas del sistema. Para valorar estadísticamente la fiabilidad de un sistema, hay que observar algunas fallas. Si, por ejemplo, se tiene una POFOD de 0.0001 (1 falla en 10,000 demandas), entonces tal vez haya que diseñar pruebas que realicen 50 o 60 mil demandas sobre un sistema donde se observen numerosas fallas. Prácticamente sería imposible diseñar e implementar este número de pruebas. Por lo tanto, la sobreespecificación de la fiabilidad conduce a costos de pruebas muy elevados.

Cuando especifique la disponibilidad de un sistema, es probable que tenga problemas similares. Aun cuando parece ser deseable un nivel de disponibilidad muy elevado, la mayoría de los sistemas tienen patrones de demanda muy intermitentes (por ejemplo, un sistema empresarial se usará básicamente durante horas laborales normales) y una sola cifra de disponibilidad no refleja realmente las necesidades del usuario. Se necesita alta disponibilidad sólo cuando el sistema se usa, pero no en otros momentos. Desde luego, dependiendo del tipo de sistema, podría no haber diferencias prácticas reales entre una disponibilidad de 0.999 y una de 0.9999.

Un problema fundamental con la sobreespecificación es que sería casi imposible demostrar que se logró un nivel de fiabilidad o disponibilidad muy alto. Por ejemplo, suponga que un sistema pretendía usarse en una aplicación crítica para la protección y que luego se le requirió no fallar nunca durante su vida total. Imagine que se instalarán 1,000 copias del sistema y que el sistema se ejecutará 1,000 veces por segundo; la vida

proyectada del sistema es 10 años. Por lo tanto, el número total de ejecuciones del sistema es aproximadamente $3*10^{14}$. No hay razón para especificar que la tasa de ocurrencia de fallas deba ser $1/10^{15}$ ejecuciones (esto permite algún factor de seguridad), pues no es factible probar el sistema por largo tiempo para validar este nivel de fiabilidad.

En consecuencia, las organizaciones deben ser realistas sobre si vale la pena especificar y validar un nivel de fiabilidad muy alto. Los niveles de fiabilidad altos se justifican claramente en los sistemas donde es crítica la operación fiable, como en los sistemas de conmutación telefónica, o donde la falla del sistema daría como resultado grandes pérdidas económicas. Quizá no se justifiquen para muchos tipos de empresas o sistemas científicos. Estos sistemas tienen requerimientos de fiabilidad modestos, pues los costos de una falla son simplemente demoras de procesamiento, y el proceso de recuperación es directo y relativamente económico.

Hay varios pasos para evitar la sobreespecificación de la fiabilidad del sistema:

1. Especifique los requerimientos de disponibilidad y fiabilidad para diferentes tipos de fallas. Debe haber una probabilidad de ocurrencia más baja para fallas graves que para fallas menores.
2. Especifique por separado los requerimientos de disponibilidad y fiabilidad para diferentes servicios. Las fallas que afectan los servicios más críticos tienen que especificarse como menos probables que aquellas sólo con efectos locales. Puede tomar la decisión de limitar la especificación de fiabilidad cuantitativa a los servicios del sistema más críticos.
3. Decida si realmente necesita fiabilidad en un sistema de software o si las metas de confiabilidad globales del sistema se logran en otras formas. Por ejemplo, puede usar mecanismos de detección de errores para verificar las salidas de un sistema y disponer procesos para corregir errores. Entonces, tal vez no haya necesidad de un nivel alto de fiabilidad en el sistema que genera las salidas.

Para ilustrar este último caso, considere los requerimientos de fiabilidad para el sistema de un cajero automático que otorgue efectivo y ofrezca otros servicios a los clientes. Si hay problemas del hardware o software del cajero automático, entonces éstos conducirán a entradas incorrectas en la base de datos de la cuenta del cliente. Esto podría evitarse al especificar un nivel muy alto de fiabilidad del hardware y software en dicho cajero.

Sin embargo, los bancos tienen muchos años de experiencia sobre cómo identificar y corregir las transacciones incorrectas en las cuentas. Usan métodos de contabilidad para detectar cuándo funcionan mal las cosas. La mayoría de las transacciones que fallan pueden simplemente cancelarse, lo cual no deriva en alguna pérdida para el banco y sólo causa inconvenientes menores al cliente. En consecuencia, los bancos que operan redes de cajeros automáticos aceptan que las fallas de éstos podrían significar que un número pequeño de transacciones sean incorrectas; sin embargo, consideran más efectivo en cuanto a costos corregir estas últimas que incurrir en costos muy elevados al evitar algunas transacciones fallidas.

Para un banco (y para sus clientes), la disponibilidad de la red de los cajeros automáticos es más importante que si fallan o no transacciones individuales en el cajero. La falta de disponibilidad significa más demanda por servicios del cliente, insatisfacción del cliente, costos de ingeniería para reparar la red, etcétera. Por lo tanto, para sistemas basados en transacciones, como los sistemas bancarios y de comercio electrónico, el enfoque

de la especificación de fiabilidad, por lo general, está en la especificación de la disponibilidad del sistema.

Para especificar la disponibilidad de una red de cajeros automáticos, hay que identificar los servicios del sistema y especificar la disponibilidad requerida para cada uno de ellos. Éstos son:

- el servicio de base de datos de la cuenta del cliente;
- los servicios individuales proporcionados por un cajero automático, como "retiro de efectivo", "proporcionar información de cuenta", etcétera.

Aquí, el servicio de base de datos es más crítico, pues la falla de este servicio significaría que todos los cajeros automáticos en la red estén fuera de operación. Por lo tanto, es necesario especificar esto para tener un alto nivel de disponibilidad. En tal caso, una cifra aceptable para disponibilidad de base de datos (si se ignoran conflictos como mantenimiento y actualizaciones programadas) probablemente sería de alrededor de 0.9999, entre las 7 a.m. y las 11 p.m. Ello significa un tiempo muerto de menos de un minuto por semana. En la práctica, significa que muy pocos clientes resultarían afectados y sólo conduciría a inconvenientes menores para éstos.

Para un cajero automático individual, la disponibilidad global depende de la fiabilidad mecánica y del hecho de que pueda quedarse sin efectivo. Es probable que los conflictos de software tengan menos efecto que factores como éstos. En consecuencia, es aceptable un menor nivel de disponibilidad para el software del cajero automático. Por consiguiente, la disponibilidad global del software del cajero puede especificarse como 0.999, lo cual significa que una máquina quizá no esté disponible diariamente entre uno y dos minutos.

Para ilustrar la especificación de fiabilidad basada en fallas, considere los requerimientos de fiabilidad para el software de control en la bomba de insulina. Este sistema entrega insulina varias veces al día y monitoriza la glucosa en la sangre del usuario muchas veces por hora. Puesto que el uso del sistema es intermitente y las consecuencias de la falla son graves, la métrica de fiabilidad más adecuada es POFOD (probabilidad de fallas en la petición).

Existen dos posibles tipos de fallas en la bomba de insulina:

1. Fallas transitorias de software que pueden repararse mediante acciones del usuario, como restablecimiento o recalibración de la máquina. Para estos tipos de fallas, un valor relativamente bajo de POFOD (por ejemplo, 0.002) sería aceptable. Esto significa que una falla puede ocurrir en cada 500 demandas hechas a la máquina. Ello representa aproximadamente una vez cada 3.5 días, porque el azúcar en la sangre se verifica casi cinco veces por hora.
2. Fallas permanentes del software que requieren la reinstalación del software por parte del fabricante. La probabilidad de este tipo de fallas debe ser mucho menor. Alrededor de una vez al año es la cifra mínima, de modo que la POFOD no debe ser mayor que 0.00002.

Sin embargo, la falla para administrar insulina no tiene implicaciones inmediatas en la seguridad, de modo que factores comerciales, más que los de seguridad, controlan el nivel de fiabilidad requerida. Los costos del servicio son altos porque los usuarios necesitan reparación y sustitución rápida. Es interés del fabricante limitar el número de fallas permanentes que requieran reparación.

RF1: Debe definirse el rango predefinido para todas las entradas de operador; asimismo, el sistema debe verificar que todas las entradas de operador se sitúen dentro de este rango predefinido. (Comprobación)

RF2: Tienen que conservarse copias de la base de datos del paciente en dos servidores separados que no se alojen en el mismo edificio. (Recuperación, redundancia)

RF3: Debe usarse la programación de n-versión para implementar el sistema de control de frenado. (Redundancia)

RF4: El sistema debe implementarse en un subconjunto seguro de Ada y comprobarse mediante un análisis estático. (Proceso)

Figura 12.8 Ejemplos de requerimientos de fiabilidad funcional

12.3.3 Especificación de fiabilidad funcional

La especificación de fiabilidad funcional incluye identificar los requerimientos que definen las restricciones y las características que contribuyen a la fiabilidad del sistema. Para sistemas donde la fiabilidad se especifica cuantitativamente, pueden ser necesarios tales requerimientos funcionales para garantizar el logro de un nivel requerido de fiabilidad.

Existen tres tipos de requerimientos de fiabilidad funcional para un sistema:

1. *Requerimientos de comprobación* Tales requerimientos identifican las comprobaciones de las entradas al sistema, para garantizar que las entradas incorrectas o fuera de rango se detecten antes de que las procese el sistema.
2. *Requerimientos de recuperación* Dichos requerimientos se implementan para ayudar al sistema a recuperarse luego de que ocurre una falla. Por lo general, estos requerimientos se relacionan con el hecho de conservar copias del sistema y sus datos, y especificar la forma en que se restauran los servicios del sistema después de una falla.
3. *Requerimientos de redundancia* Especifican las características redundantes del sistema que aseguran que la falla en un solo componente no conduzca a una pérdida completa del servicio. Esto se estudia con más detalle en el siguiente capítulo.

Además, los requerimientos de fiabilidad pueden incluir requerimientos de proceso para fiabilidad. Se trata de requerimientos que aseveran que la buena práctica, conocida porque reduce el número de fallas de un sistema, se usa en el proceso de desarrollo. En la figura 12.8 se muestran algunos ejemplos de requerimientos de fiabilidad y proceso funcional.

No hay reglas simples para derivar requerimientos de fiabilidad funcional. En las organizaciones que desarrollan sistemas críticos, usualmente existe conocimiento organizacional sobre posibles requerimientos de fiabilidad y cómo repercuten en la fiabilidad real de un sistema. Estas organizaciones pueden especializarse en tipos específicos de sistemas, tales como los sistemas de control ferroviario, de manera que los requerimientos de fiabilidad pueden reutilizarse mediante un rango de sistemas.

12.4 Especificación de seguridad

La especificación de requerimientos de seguridad para sistemas tiene algo en común con los requerimientos de protección. No resulta práctico especificarlos de manera cuantitativa, y los requerimientos de seguridad con frecuencia son requerimientos enunciados como "no debe" que definen el comportamiento inaceptable del sistema, en lugar de la funcionalidad requerida del sistema. Sin embargo, la seguridad es un problema que exige mayor esfuerzo que la protección, por diversas razones:

1. Cuando se considera la protección, se puede suponer que el entorno donde el sistema se instala no es hostil. Nadie trata de causar un incidente relacionado con la seguridad. Cuando se considera la seguridad, uno debe suponer que son deliberados los ataques al sistema y que el atacante puede conocer las debilidades del sistema.
2. Cuando ocurren fallas de sistema que plantean un riesgo a la protección, se buscan los errores o las omisiones que produjeron la falla. Si los ataques deliberados ocasionan fallas de sistema, encontrar la causa raíz suele ser más difícil, ya que el atacante puede tratar de encubrir el motivo de la falla.
3. Por lo general es aceptable desactivar o degradar los servicios del sistema para evitar una falla relacionada con la protección. Sin embargo, los ataques a un sistema pueden ser los llamados ataques de negación de servicio, que tienen la intención de apagar el sistema. Desactivar el sistema significa que el ataque fue exitoso.
4. Los eventos relacionados con la protección no los genera un adversario inteligente. Un atacante puede indagar las defensas de un sistema a través de una serie de ataques, y modificar éstos conforme aprende más sobre el sistema y sus respuestas.

Tales distinciones significan que los requerimientos de seguridad suelen ser más extensos que los requerimientos de protección. Los requerimientos de protección conducen a la generación de requerimientos funcionales del sistema que suministran protección contra eventos y fallas que podrían originar fallas en la operación vinculadas con la protección. Básicamente se enfocan en la comprobación de problemas y en la toma de acciones si ocurren dichos problemas. En contraste, existen muchos tipos de requerimientos de seguridad que protegen contra las diferentes amenazas que enfrenta un sistema. Firesmith (2003) identificó 10 tipos de requerimientos de seguridad que pueden incluirse en una especificación de sistema:

1. Los requerimientos de identificación especifican si un sistema debe o no debe identificar a sus usuarios antes de interactuar con ellos.
2. Los requerimientos de autenticación explican cómo se identifica a los usuarios.
3. Los requerimientos de autorización detallan los privilegios y permisos de acceso de los usuarios identificados.
4. Los requerimientos de inmunidad definen cómo un sistema debe protegerse a sí mismo contra virus, gusanos y amenazas similares.
5. Los requerimientos de integridad describen cómo puede evitarse la corrupción de datos.

Gestión de riesgos a la seguridad

La protección es un tema legal y las empresas no están en condiciones de negarse a participar en la creación de sistemas seguros. Sin embargo, algunos aspectos de la seguridad son asuntos empresariales: una compañía quizá decida no implementar ciertas medidas de seguridad y cubrir las pérdidas que resulten de dicha decisión. La gestión del riesgo es el proceso para decidir qué activos deben protegerse y cuánto puede gastarse en su protección.

http://www.SoftwareEngineering-9.com/Web/Security/RiskMan.html

6. Los requerimientos de detección de intrusiones puntualizan qué mecanismos deben usarse para detectar ataques al sistema.
7. Los requerimientos de no repudio especifican que una parte en una transacción no puede negar su involucramiento en dicha transacción.
8. Los requerimientos de privacidad se refieren a cómo se mantiene la privacidad de los datos.
9. Los requerimientos de auditoría de seguridad plantean cómo puede auditarse y verificarse el uso del sistema.
10. Los requerimientos de seguridad de mantenimiento del sistema especifican cómo una aplicación puede evitar cambios autorizados a partir de la inhabilitación accidental de sus mecanismos de seguridad.

Desde luego, usted no verá en cada sistema todos estos tipos de requerimientos de seguridad. Los requerimientos particulares dependen del tipo de sistema, de la situación de uso y de los usuarios esperados.

El proceso de análisis y valoración del riesgo, que se estudia en la sección 12.1, puede utilizarse para identificar requerimientos de seguridad del sistema. Existen tres etapas en este proceso:

1. *Análisis preliminar del riesgo* En esta etapa todavía no se toman decisiones sobre los requerimientos detallados del sistema, el diseño del sistema o la tecnología de implementación. La meta de este proceso de valoración es derivar requerimientos de seguridad para el sistema en su conjunto.
2. *Análisis del riesgo del ciclo de vida* Esta valoración de riesgo tiene lugar durante el ciclo de vida de desarrollo del sistema, después de tomarse elecciones de diseño. Los requerimientos adicionales de seguridad toman en cuenta las tecnologías usadas en la construcción del sistema, así como las decisiones de diseño e implementación del sistema.
3. *Análisis del riesgo operativo* Esta valoración de riesgo considera los riesgos al sistema operativo impuestos por ataques maliciosos de los usuarios, con o sin conocimiento interno del sistema.

Los procesos de valoración y análisis de riesgo que se usan en la especificación de requerimientos de seguridad son variantes del proceso de especificación genérico dirigido por riesgos que se estudió en la sección 12.1. En la figura 12.9 se muestra un proceso de

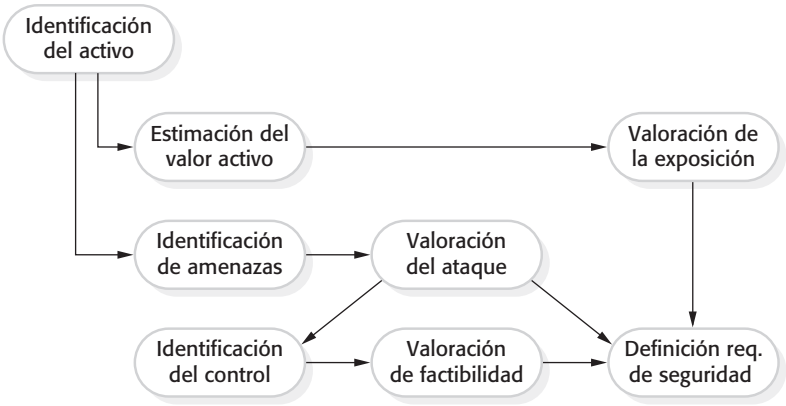


Figura 12.9 Proceso de valoración preliminar del riesgo para requerimientos de seguridad

requerimientos de seguridad dirigido por riesgos. Esto parecería diferente del proceso dirigido por riesgos de la figura 12.1, pero se indica cómo cada etapa corresponde a las etapas en el proceso genérico al incluir entre corchetes la actividad del proceso genérico. Las etapas del proceso son:

1. Identificación del activo, en la que se identifican los activos del sistema que podrían requerir protección. El sistema en sí o funciones particulares del sistema pueden identificarse como activos, al igual que los datos asociados con el sistema (identificación de riesgos).
2. Estimación del valor del activo, donde se evalúa el valor de los activos identificados (análisis de riesgos).
3. Valoración de la exposición, la cual permite valorar las pérdidas potenciales asociadas con cada activo. Deben tomarse en cuenta las pérdidas directas, como el robo de información, los costos de recuperación y la posible pérdida de reputación (análisis de riesgos).
4. Identificación de amenazas, donde se identifican las amenazas a los activos del sistema (análisis de riesgos).
5. Valoración del ataque, en la que cada amenaza se descompone en ataques que pueden hacerse al sistema y las posibles formas en que dichos ataques podrían ocurrir. Es posible usar árboles de ataque (Schneier, 1999) para analizar las posibles embestidas. Son similares a los árboles de fallas, pues comienzan con una amenaza como la raíz del árbol e identifican los posibles ataques causales y cómo éstos podrían realizarse (descomposición del riesgo).
6. Identificación del control, donde se proponen los controles que pueden instalarse para proteger un activo. Los controles son los mecanismos técnicos, como la encriptación, que sirven para proteger los activos (reducción del riesgo).
7. Valoración de factibilidad, la cual permite valorar la factibilidad técnica y los costos de los controles propuestos. No vale la pena tener controles muy caros para proteger los activos que no tienen gran valor (reducción del riesgo).
8. Definición de requerimientos de seguridad, en la que se usa el conocimiento de las valoraciones de exposición, amenazas y control, para derivar requerimientos de

Activo	Valor	Exposición
El sistema de información	Alto. Requerido para apoyar todas las consultas clínicas. Es potencialmente crítico para la seguridad.	Alta. Pérdidas financieras ya que es posible que deban cerrarse clínicas. Costos de sistemas de restauración. Posible daño al paciente si no logran prescribirse tratamientos.
La base de datos de pacientes	Alto. Requerido para apoyar todas las consultas clínicas. Es potencialmente crítico para la seguridad.	Alta. Pérdidas financieras, pues es posible que deban cerrarse clínicas. Costos de sistemas de restauración. Posible daño al paciente si no se prescriben tratamientos.
Un registro de paciente individual	Normalmente bajo, aunque puede ser alto para pacientes específicos de perfil alto.	Pérdidas directas bajas, aunque es posible la pérdida de reputación.

Figura 12.10 Análisis de activos en un reporte de valoración preliminar del riesgo para el MHC-PMS

seguridad del sistema. Éstos pueden ser requerimientos para la infraestructura del sistema o el sistema de aplicación (reducción del riesgo).

Una importante entrada al proceso de valoración y gestión del riesgo es la política de seguridad de la organización. Una política de seguridad organizacional se aplica a todos los sistemas y en ella se establece lo que debe y no debe permitirse. Por ejemplo, un aspecto de una política de seguridad militar establecería lo siguiente: "Los lectores pueden examinar sólo los documentos cuya clasificación sea la misma o esté por abajo del nivel de inspección del lector". Esto significa que, si un lector tiene nivel de inspección "secreto", puede acceder a documentos que estén clasificados como "secretos", "confidenciales" o "abiertos", pero no a documentos clasificados como "ultrasecretos".

La política de seguridad establece condiciones que un sistema de seguridad debe mantener siempre, ya que ayudan a identificar amenazas que se puedan presentar. Las amenazas son cualquier intrusión que vulnere la seguridad de la empresa. En la práctica, las políticas de seguridad son por lo general documentos informales que definen lo que se permite y no se permite. Sin embargo, Bishop (2005) discute la posibilidad de expresar las políticas de seguridad en un lenguaje formal y generar verificaciones automatizadas para garantizar que se sigue la política.

Como una forma de ilustrar este proceso de análisis de riesgo a la seguridad, considere el sistema de información hospitalario para la atención a la salud mental, MHC-PMS. Aquí no se tiene espacio donde analizar una valoración de riesgo completa, pero, en su lugar, se delinearía el sistema como una fuente de ejemplos. Esto se muestra como el fragmento de un reporte (figuras 12.10 y 12.11) que podría generarse a partir del proceso de valoración preliminar del riesgo. El reporte resultante se usa para definir los requerimientos de seguridad.

A partir del análisis de riesgo para el sistema de información hospitalario, se pueden derivar requerimientos de seguridad. Algunos ejemplos de tales requerimientos son:

1. La información del paciente debe descargarse, al inicio de una sesión clínica, desde la base de datos a un área segura en el sistema cliente.

Amenaza	Probabilidad	Control	Factibilidad
Usuario sin autorización tiene acceso como administrador del sistema y hace al sistema no disponible.	Baja	Sólo se permite la administración del sistema desde ubicaciones específicas que sean físicamente seguras.	Bajo costo de implementación, pero debe tenerse cuidado con la distribución de claves y garantizar que éstas se hallen disponibles en el acontecimiento de emergencia.
Usuario sin autorización tiene acceso como usuario del sistema y logra entrar a la información confidencial.	Alta	Exigir a todos los usuarios autenticarse con el uso de mecanismos biométricos. Registrar todos los cambios de la información del paciente para rastrear el uso del sistema.	Técnicamente factible, pero con una solución de alto costo. Posible resistencia del usuario. Simple y transparente para implementar y también soportar la recuperación.

Figura 12.11 Análisis de amenaza y control en un reporte de valoración preliminar del riesgo

2. Debe encriptarse en el sistema cliente toda la información de los pacientes.
3. La información del paciente tiene que subirse a la base de datos al terminar una sesión clínica y borrarla de la computadora cliente.
4. La bitácora de todos los cambios realizados a la base de datos del sistema y el iniciador de dichos cambios deberían mantenerse en una computadora separada del servidor de la base de datos.

Los primeros dos requerimientos están relacionados: la información del paciente se descarga a una máquina local, de modo que las consultas continúen aun cuando el servidor de la base de datos del paciente sea atacado o no se encuentre disponible. Sin embargo, esta información tiene que borrarse, de forma que los usuarios posteriores de la computadora cliente no logren acceder a la información. El cuarto requerimiento es un requerimiento de recuperación y auditoría. Ello significa que los cambios pueden recuperarse al reproducir la bitácora de cambios y que es posible descubrir quién realizó los cambios. Esto desalienta el mal uso del sistema por parte del personal autorizado.

12.5 Especificación formal

Durante más de 30 años, muchos investigadores se han dedicado al uso de métodos formales para el desarrollo de software. Los métodos formales son aproximaciones con base matemática al desarrollo del software, donde se define un modelo formal del software. Entonces es factible analizar formalmente este modelo y usarlo como base para una especificación formal del sistema. En principio, es posible iniciar con un modelo formal para el software y probar que un programa desarrollado es congruente con dicho modelo, lo cual, por consiguiente, elimina las fallas de software que resultan de errores de programación.

Técnicas de especificación formal

Las especificaciones formales del sistema se expresan mediante dos enfoques fundamentales, como modelos de las interfaces del sistema (especificaciones algebraicas) o modelos del estado del sistema. Se sugiere descargar el capítulo Web adicional sobre este tema, donde se muestran ejemplos de ambos enfoques. El capítulo incluye una especificación formal de parte del sistema de la bomba de insulina.

http://www.SoftwareEngineering-9.com/Web/ExtraChaps/FormalSpec.pdf

El punto de partida para todos los procesos de desarrollo formal es un modelo de sistema formal, que sirve como una especificación de sistema. Para crear este modelo, traduzca los requerimientos de usuario del sistema, que se expresan en lenguaje natural, diagramas y tablas, a un lenguaje matemático que tenga semántica definida formalmente. La especificación formal es una descripción sin ambigüedades de qué debe hacer el sistema. Al usar métodos manuales o soportados por herramientas, es posible comprobar que el comportamiento de un programa es congruente con la especificación.

Las especificaciones formales no sólo son importantes para una verificación del diseño e implementación del software, sino también son la forma más precisa de especificar sistemas y, por ende, de reducir el ámbito para las malas interpretaciones. Más aún, construir una especificación formal fuerza un análisis detallado de los requerimientos y ésta es una forma efectiva de descubrir problemas de requerimientos. En una especificación en lenguaje natural, los errores pueden ocultarse mediante la imprecisión del léxico. Éste no es el caso si el sistema se especifica formalmente.

Las especificaciones formales se desarrollan con frecuencia como parte de un proceso de software basado en un plan, en que el sistema se especifica completamente antes de su desarrollo. Los requerimientos del sistema y el diseño se definen a detalle, y se analizan y comprueban cuidadosamente antes de comenzar la implementación. Si se desarrolla una especificación formal del software, esto viene por lo general después de que se especificaron los requerimientos del sistema, pero antes del diseño detallado del sistema. Hay un estrecho ciclo de retroalimentación entre la especificación detallada de requerimientos y la especificación formal.

La figura 12.12 muestra las etapas de especificación de software y su interfaz con el diseño de software en un proceso de software basado en un plan. Como es costoso desarrollar especificaciones formales, es posible decidir limitar el uso de este enfoque a aquellos componentes que son críticos para la operación del sistema. Éstos se identifican en el diseño arquitectónico del sistema.

Hace pocos años se desarrolló el apoyo automatizado para analizar una especificación formal. Los comprobadores de modelos (Clarke *et al*., 2000) son herramientas de software que toman una especificación formal basada en un estado (un modelo de sistema) como una entrada, junto con la especificación de algunas propiedades deseables formalmente expresadas, tales como “no hay estados inalcanzables”. El programa de comprobación de modelo analiza exhaustivamente la especificación, y cualquier reporte de que el modelo satisface la propiedad del sistema, o presenta un ejemplo que muestra que no se satisface. La comprobación del modelo se relaciona estrechamente con la noción de análisis estático, y en el capítulo 15 se estudian estos enfoques generales para la verificación del sistema.

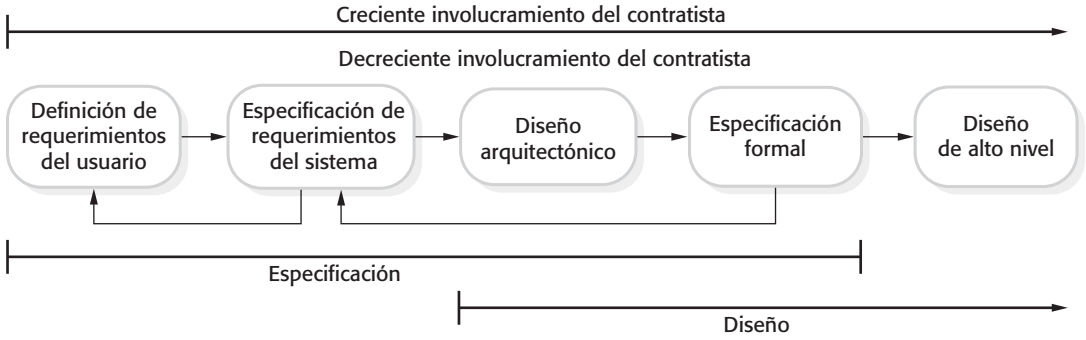


Figura 12.12 Especificación formal en un proceso de software basado en un plan

Las ventajas de desarrollar una especificación formal y de usar ésta en un proceso de desarrollo formal son:

1. Mientras se desarrolla una especificación formal a detalle, se obtiene una comprensión profunda y pormenorizada de los requerimientos del sistema. Incluso si no se usa la especificación en un proceso de desarrollo formal, la detección del error de requerimientos es un potente argumento para elaborar una especificación formal (Hall, 1990). Los problemas de requerimientos que se descubren con antelación son, por lo general, mucho menos costosos de corregir, que si se encuentran en etapas posteriores en el proceso de desarrollo.
2. Conforme la especificación se expresa en un lenguaje con semántica definida formalmente, puede analizarse de manera automática para descubrir inconsistencias y aquello que no se completó.
3. Si se usa un método como el método B, se puede transformar la especificación formal en un programa, a través de una secuencia de transformaciones que preserven la exactitud. En consecuencia, se garantiza que el programa resultante cumpla su especificación.
4. Los costos de las pruebas del programa suelen reducirse porque el programa se verificó contra su especificación.

A pesar de estas ventajas, los métodos formales tienen efecto limitado sobre el desarrollo práctico del software, incluso para sistemas críticos. En consecuencia, existe muy poca experiencia en la comunidad del desarrollo y uso de las especificaciones formales del sistema. Los argumentos que se esgrimen contra el desarrollo de una especificación formal del sistema son:

1. El problema de que los propietarios y expertos de dominio no entiendan una especificación formal, de modo que no pueden comprobar que representa con precisión sus requerimientos. Los ingenieros de software, que entienden la especificación formal, quizá no comprendan el dominio de aplicación, así que tampoco estarían seguros de que la especificación formal es una reflexión precisa de los requerimientos del sistema.
2. Es bastante sencillo cuantificar los costos de crear una especificación formal, pero es más difícil estimar el posible ahorro en los costos que resultará de su uso. Como resultado, los administradores no están dispuestos a asumir el riesgo de adoptar este enfoque.

Costos de especificación formal

El desarrollo de una especificación formal es un proceso costoso, puesto que se necesita mucho tiempo para traducir los requerimientos a un lenguaje formal y comprobar la especificación. La experiencia demuestra que se puede ahorrar en las pruebas y la verificación del sistema y, al parecer, especificar un sistema de manera formal no aumenta significativamente los costos globales de desarrollo. Sin embargo, cambia el equilibrio de costos, pues se incurre en más costos anticipados en el proceso de desarrollo.

http://www.SoftwareEngineering-9.com/Web/FormalSpecCosts/

3. La mayoría de los ingenieros de software no han sido capacitados para usar lenguajes de especificación formal. Por lo tanto, se muestran renuentes a proponer su uso en procesos de desarrollo.
4. Es difícil escalar los enfoques actuales a la especificación formal para sistemas muy grandes. Cuando se usa especificación formal, es básicamente para especificar software núcleo (kernel) crítico en lugar de sistemas completos.
5. La especificación formal no es compatible con los métodos de desarrollo ágiles.

No obstante, al momento de escribir este texto, se han usado métodos formales en el desarrollo de algunas aplicaciones críticas para la protección y la seguridad. También pueden utilizarse efectivamente en términos de costo en el desarrollo y la validación de partes críticas de un sistema de software más grande y complejo (Badeau y Amelot, 2005; Hall, 1996; Hall y Chapman, 2002; Miller *et al*., 2005; Wordworth, 1996). Son la base de las herramientas utilizadas en la verificación estática, tales como el sistema de verificación de controlador que usa Microsoft (Ball *et al*., 2004; Ball *et al*., 2006) y el lenguaje SPARK/Ada (Barnes, 2003) para ingeniería de sistemas críticos.

PUNTOS CLAVE

- El análisis de riesgos es una actividad importante en la especificación de requerimientos de seguridad y confiabilidad. Implica la identificación de riesgos que pueden derivar en accidentes o incidentes. Entonces se generan requerimientos de sistema para garantizar que dichos riesgos no ocurran o que, si ocurren, no conduzcan a un incidente o accidente.
- Puede usarse un enfoque dirigido por peligros para comprender los requerimientos de seguridad de un sistema. Se identifican los peligros potenciales y se desglosan (con métodos tales como el análisis de árbol de fallas) para descubrir sus causas raíz. Luego se especifican los requerimientos para evitar dichos problemas o recuperarse de ellos.
- Los requerimientos de fiabilidad pueden definirse de manera cuantitativa en la especificación de requerimientos del sistema. Las métricas de fiabilidad incluyen probabilidad de fallas en la petición (POFOD), tasa de ocurrencia de fallas (ROCOF) y disponibilidad (AVAIL).

- Es importante no sobreespecificar la fiabilidad requerida del sistema, pues ello conduce a costos adicionales innecesarios en los procesos de desarrollo y validación.
- Es más difícil identificar los requerimientos de seguridad que los requerimientos de protección, puesto que un atacante del sistema podría usar el conocimiento de las vulnerabilidades del sistema para planear un ataque a éste, y aprender sobre las vulnerabilidades a partir de ataques exitosos.
- Para especificar requerimientos de seguridad, se tienen que identificar los activos que deben protegerse y determinar cómo usar las técnicas de seguridad y la tecnología para proteger dichos activos.
- Los métodos formales de desarrollo de software se apoyan en una especificación de sistema que se expresa como modelo matemático. El desarrollo de una especificación formal ofrece el beneficio clave de estimular un examen y un análisis detallado de los requerimientos del sistema.

LECTURAS SUGERIDAS

Safeware: System Safety and Computers. Se trata de una discusión integral de todos los aspectos de sistemas críticos de protección. Es un texto particularmente sólido en su descripción del análisis de peligro y la derivación de requerimientos a partir de esto. (N. Leveson, Addison-Wesley, 1995.)

"Security Use Cases." Buen artículo, disponible en la Web, que se enfoca en cómo pueden usarse casos de uso en la especificación de seguridad. El autor también tiene algunos interesantes artículos acerca de la especificación de seguridad que se citan en este artículo. (D. G. Firesmith, *Journal of Object Technology*, **2** (3), mayo-junio de 2003.) http://www.jot.fm/issues/issue_2003_05/column6/.

"Ten Commandments of Formal Methods . . .Ten Years Later." Conjunto de lineamientos para el uso de métodos formales que se propuso por primera vez en 1996 y que se revisan en este ensayo. Es un atractivo resumen de los conflictos prácticos en torno al uso de los métodos formales. (J. P. Bowen y M. G. Hinchey, *IEEE Computer*, **39** (1), enero de 2006.) http://dx.doi.org/10.1109/MC.2006.35.

"Security Requirements for the Rest of Us: A Survey." Buen punto de partida para leer sobre la especificación de requerimientos de seguridad. Los autores se interesan en los enfoques ligeros, por encima de los formales. (I. A. Tøndel, M. G. Jaatun, y P. H. Meland, *IEEE Software*, **25** (1), enero/febrero de 2008.) http://dx.doi.org/10.1109/MS.2008.19.

EJERCICIOS

12.1. Explique por qué las fronteras en el triángulo de riesgo que se muestra en la figura 12.12 son proclives a modificarse con el tiempo y con las actitudes sociales cambiantes.

12.2. Explique por qué el enfoque basado en riesgos se interpreta en diferentes formas cuando se especifican seguridad y protección.

12.3. En el sistema de bomba de insulina, el usuario debe cambiar la aguja y el suministro de insulina a intervalos regulares, y también puede modificar la dosis individual máxima y la dosis diaria máxima por administrarse. Sugiera tres errores de usuario que puedan ocurrir y proponga requerimientos de protección que evitarían que dichos errores deriven en un accidente.

12.4. Un sistema de software crítico de protección para tratar a pacientes con cáncer tiene dos componentes principales:

- Una máquina de terapia de radiación que entrega dosis controladas de radiación en los sitios del tumor. Esta máquina está controlada por un sistema de software embebido.
- Una base de datos de tratamiento que incluye detalles del tratamiento dado a cada paciente. Los requerimientos de tratamiento se ingresan en esta base de datos y se descargan automáticamente a la máquina de terapia de radiación.

Identifique tres peligros que puedan surgir en este sistema. Para cada peligro, sugiera un requerimiento defensivo que reduzca la probabilidad de que dichos peligros deriven en un accidente. Explique por qué es probable que su defensa sugerida reduzca el riesgo asociado con el peligro.

12.5. Sugiera métricas de fiabilidad adecuadas para las siguientes clases de sistemas de software. Ofrezca razones para su elección de métrica. Prevea el uso de dichos sistemas y sugiera valores adecuados para las métricas de fiabilidad.

- Un sistema que monitorice pacientes en una unidad hospitalaria de cuidado intensivo.
- Un procesador de texto.
- Un sistema de control para una máquina expendedora automatizada.
- Un sistema para controlar el frenado de un automóvil.
- Un sistema para controlar una unidad de refrigeración.
- Un generador de reportes administrativos.

12.6. Un sistema de protección ferroviario aplica automáticamente los frenos de un tren si se supera el límite de rapidez para un segmento de vía, o si el tren entra en un segmento de vía que actualmente está señalado con una luz roja (es decir, no debe entrar al segmento). Ofrezca razones para su respuesta, elija una métrica de fiabilidad que pueda usarse en la especificación de la fiabilidad requerida para tal sistema.

12.7. Existen dos requerimientos esenciales de seguridad para el sistema de protección ferroviario:

- El tren no debe entrar en un segmento de vía que esté señalizado con una luz roja.
- El tren no debe superar el límite de rapidez especificado para una sección de vía.

Suponiendo que el estatus de la señal y el límite de rapidez para el segmento de vía se transmiten a un software a bordo del tren antes de que entre al segmento de vía, proponga cinco posibles requerimientos funcionales del sistema para el software a bordo, que puedan generarse a partir de los requerimientos de protección del sistema.

12.8. Explique por qué hay necesidad tanto de una valoración preliminar de riesgos de seguridad como de una valoración de riesgo de seguridad de ciclo de vida durante el desarrollo de un sistema.

12.9. Extienda la tabla de la figura 12.11 para identificar dos amenazas más al MHC-PMS, junto con controles asociados. Úselos como base para generar más requerimientos de seguridad de software que implementen los controles propuestos.

12.10. ¿Los ingenieros de software que trabajan en la especificación y el desarrollo de sistemas relacionados con la protección deben certificarse profesionalmente en alguna forma? Explique su razonamiento.

REFERENCIAS

Badeau, F. y Amelot, A. (2005). "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL". Proc. ZB 2005: Formal Specification and Development in Z and B, Guildford, UK: Springer.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. y Ustuner, A. (2006). "Thorough Static Analysis of Device Drivers". Proc. EuroSys 2006, Leuven, Belgium.

Ball, T., Cook, B., Levin, V. y Rajamani, S. K. (2004). "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft". Proc. Integrated Formal Methods 2004, Canterbury, UK: Springer.

Barnes, J. P. (2003). *High-integrity Software: The SPARK Approach to Safety and Security*. Harlow, UK: Addison-Wesley.

Bishop, M. (2005). *Introduction to Computer Security*. Boston: Addison-Wesley.

Brazendale, J. y Bell, R. (1994). "Safety-related control and protection systems: standards update". *IEE Computing and Control Engineering J.*, **5** (1), 6–12.

Clarke, E. M., Grumberg, O. y Peled, D. A. (2000). *Model Checking*. Cambridge, Mass.: MIT Press.

Firesmith, D. G. (2003). "Engineering Security Requirements". *Journal of Object Technology*, **2** (1), 53–68.

Hall, A. (1990). "Seven Myths of Formal Methods". *IEEE Software*, **7** (5), 11–20.

Hall, A. (1996). "Using Formal methods to Develop an ATC Information System". *IEEE Software*, **13** (2), 66–76.

Hall, A. y Chapman, R. (2002). "Correctness by Construction: Developing a Commercially Secure System". *IEEE Software*, **19** (1), 18–25.

Jahanian, F. y Mok, A. K. (1986). "Safety analysis of timing properties in real-time systems". *IEEE Trans.on Software Engineering.*, **SE-12** (9), 890–904.

Leveson, N. y Stolzy, J. (1987). "Safety analysis using Petri nets". *IEEE Transactions on Software Engineering*, **13** (3), 386–397.

Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Reading, Mass.: Addison-Wesley.

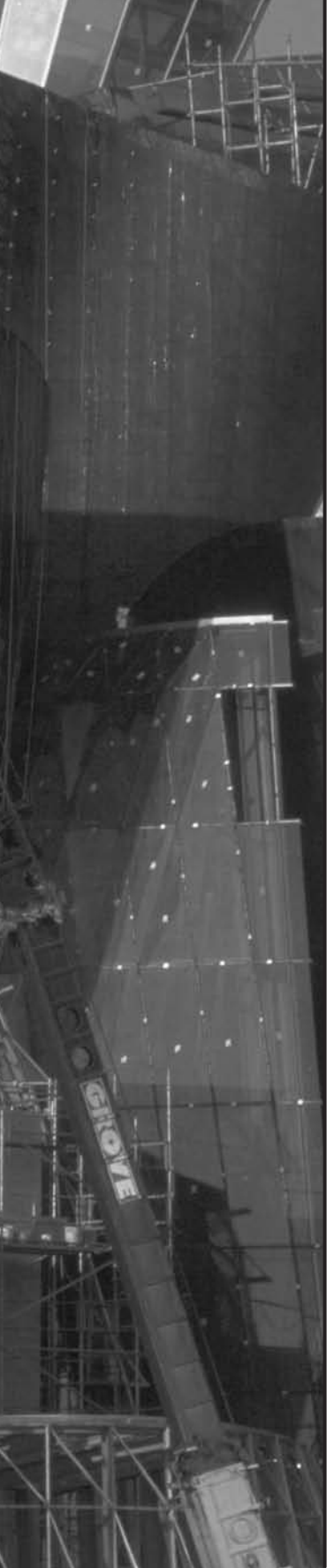
Miller, S. P., Anderson, E. A., Wagner, L. G., Whalen, M. W. y Heimdahl, M. P. E. (2005). "Formal Verification of Flight Control Software". *Proc. AIAA Guidance*, Navigation and Control Conference, San Francisco.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Nueva York: McGraw-Hill.

Schneier, B. (1999). "Attack Trees". *Dr Dobbs Journal*, **24** (12), 1–9.

Storey, N. (1996). *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.



13

Ingeniería de confiabilidad

Objetivos

El objetivo de este capítulo es estudiar los procesos y técnicas para el desarrollo de sistemas ampliamente confiables. Al estudiar este capítulo:

- comprenderá cómo la confiabilidad del sistema puede lograrse mediante componentes redundantes y diversos;
- conocerá cómo los procesos de software confiables contribuyen al desarrollo de software confiable;
- identificará cómo diferentes estilos arquitectónicos favorecen el desarrollo de software redundante y diverso;
- conocerá las buenas prácticas de programación que deben usarse en la ingeniería de sistemas confiables.

Contenido

13.1 Redundancia y diversidad
13.2 Procesos confiables
13.3 Arquitecturas de sistemas confiables
13.4 Programación confiable

El uso de técnicas de ingeniería de software, mejores lenguajes de programación y mejor calidad de gestión conducen a beneficios considerables en la confiabilidad para la mayoría del software. No obstante, aún se generan fallas de sistema que afectan la disponibilidad del sistema o que llevan a la producción de resultados incorrectos. En algunos casos, tales fallas simplemente causan inconvenientes menores. Los proveedores de sistemas pueden decidir sólo coexistir con dichas fallas, sin corregir los errores en sus sistemas. Sin embargo, en algunos sistemas, la falla podría causar pérdidas de vidas, o bien, grandes pérdidas económicas o de reputación. Éstos se conocen como "sistemas críticos", para los cuales resulta esencial un alto nivel de confiabilidad.

Los ejemplos de sistemas críticos comprenden sistemas de control de procesos, sistemas de protección que desactivan otros sistemas en caso de falla, sistemas médicos, conmutadores de telecomunicaciones y sistemas de control de vuelo. Para mejorar la confiabilidad del software en un sistema crítico, se utilizan herramientas y técnicas de desarrollo especial. Aun cuando tales herramientas y técnicas aumentan generalmente los costos de desarrollo del sistema, también reducen el riesgo de fallas de sistema y las pérdidas derivadas de tales fallas.

La ingeniería de confiabilidad se interesa por las técnicas usadas para mejorar la confiabilidad de los sistemas tanto críticos como no críticos. Dichas técnicas apoyan tres enfoques complementarios que se emplean en el desarrollo del software confiable:

1. *Evitación de fallas en el desarrollo* El proceso de diseño e implementación del software debe usar enfoques para el desarrollo de software que ayuden a evitar errores de diseño y de programación y, además, que minimicen el número de fallas que sea factible que surjan al ejecutar el sistema. Menos fallas en el desarrollo significan menos oportunidades de fallas durante el tiempo de ejecución.
2. *Detección y corrección de fallas en el desarrollo* Los procesos de verificación y validación se diseñan para descubrir y eliminar fallas en el desarrollo de un programa, antes de desplegarlo para uso operacional. Los sistemas críticos requieren verificación y validación muy costosa para descubrir el mayor número posible de fallas antes del despliegue y, asimismo, para convencer a los participantes de que el sistema es confiable. Este tema se expone en el capítulo 15.
3. *Tolerancia a fallas en el desarrollo* El sistema se diseñó de modo que se detectan las fallas en el desarrollo o el comportamiento inesperado del sistema en el tiempo de ejecución, y se gestionan para que no ocurra una falla de sistema. En todos los sistemas pueden incluirse enfoques simples a la tolerancia de fallas en el desarrollo basados en la comprobación del tiempo de ejecución interna. Sin embargo, las técnicas más especializadas de tolerancia a fallas (como el uso de arquitecturas de sistema tolerantes a fallas) sólo se usan por lo general cuando se requiere un nivel muy alto de disponibilidad y fiabilidad del sistema.

Por desgracia, aplicar técnicas para evitar, detectar y tolerar fallas en el desarrollo conduce a una situación de baja de rendimientos. El costo para encontrar y eliminar las fallas en el desarrollo restantes en un sistema de software se eleva exponencialmente a medida que se descubren y eliminan las fallas del programa (figura 13.1). Conforme el software se vuelve más confiable, es necesario emplear mayor tiempo y esfuerzo para encontrar cada vez menos fallas. En cierta etapa, incluso para los sistemas críticos, se vuelven injustificables los costos del esfuerzo adicional.

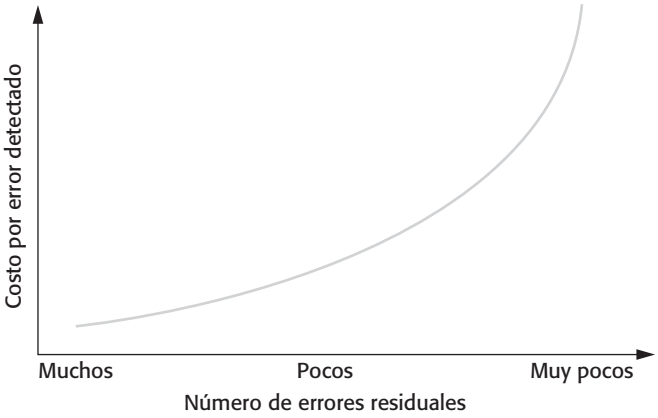


Figura 13.1 El aumento de costos por la eliminación de fallas residuales en el desarrollo

Como resultado, las compañías de desarrollo de software aceptan que su software siempre presentará algunas fallas residuales. El nivel de fallas depende del tipo de sistema. Los productos empacados tienen un nivel de fallas relativamente alto, en tanto que los sistemas críticos tienen mucho menor densidad de fallas.

La razón para aceptar fallas en el desarrollo es que, en caso de falla del sistema, es menos costoso pagar por las consecuencias que descubrir y eliminar las fallas en el desarrollo antes de entregar el sistema. Sin embargo, como se estudió en el capítulo 11, la decisión de liberar software defectuoso no es simplemente una medida económica. También debe considerarse la aceptación social y política de la falla de sistema.

Muchos sistemas críticos, como los de aeronaves, médicos y de contabilidad, se usan en dominios sistematizados como son el transporte aéreo, la medicina y las finanzas. Los gobiernos definen las medidas que se aplican en dichos dominios y designan a un cuerpo organizador para garantizar que las compañías sigan dichas regulaciones. En la práctica, esto significa que el regulador con frecuencia tiene que convencerse de que los sistemas de software críticos son confiables, y esto requiere de una clara evidencia que muestre que dichos sistemas son confiables.

Por consiguiente, el proceso de desarrollo para sistemas críticos no sólo se interesa en la elaboración de un sistema confiable, sino también en producir la evidencia que logre convencer a un regulador sobre la confiabilidad del sistema. La producción de tal evidencia consume una proporción significativa de los costos de desarrollo para sistemas críticos y, también, es un factor importante que contribuye a los altos costos de los sistemas críticos. En el capítulo 15 se examinan los conflictos sobre producir casos de seguridad y confiabilidad.

13.1 Redundancia y diversidad

La redundancia y la diversidad son estrategias fundamentales para aumentar la confiabilidad de cualquier tipo de sistema. Redundancia significa que en un espacio se incluye capacidad de repuesto que está disponible si falla parte de dicho sistema. Diversidad

La explosión del Ariane 5

En 1996, el cohete Ariane 5, de la Agencia Espacial Europea, estalló 37 segundos después del despegue de su vuelo inaugural. La falla fue causada por un error de los sistemas de software. Había un sistema de respaldo, pero éste no era diverso y, además, el software de la computadora de respaldo falló exactamente igual. El cohete y su satélite de carga fueron destruidos.

http://www.SoftwareEngineering-9.com/Web/DependabilityEng/Ariane/

quiere decir que los componentes del sistema son de diferentes tipos, lo cual también aumenta las probabilidades de que no fallen exactamente de la misma forma.

La redundancia y la diversidad se usan para mejorar la confiabilidad en la vida cotidiana. Como ejemplo de redundancia, la mayoría de los consumidores tienen bombillas eléctricas de repuesto en sus hogares, para que puedan recuperarse rápidamente de la falla de una bombilla que estaba en uso. Comúnmente, para asegurar las casas se usa más de una cerradura (redundancia) y, con frecuencia, las cerraduras suelen ser de tipos diferentes (diversidad). Esto significa que, si un intruso encuentra una forma de inhabilitar alguna de las cerraduras, antes de conseguir el acceso, debería descubrir una manera diferente de inhabilitar la otra cerradura. Como tema rutinario, todos deben respaldar sus computadoras y además mantener copias redundantes de los datos. Para evitar problemas con la falla de discos, los respaldos deben mantenerse en un dispositivo externo, independiente y diverso.

Los sistemas de software que se diseñan para confiabilidad pueden incluir componentes redundantes con la misma funcionalidad que otros componentes del sistema. Éstos se sustituyen en el sistema al fallar los componentes primarios. Si los componentes redundantes son diversos (esto es, distintos de otros componentes), una falla común de los componentes replicados no derivará en una falla de sistema. También se proporciona redundancia al agregar un código de comprobación adicional, lo cual no es estrictamente necesario para la función del sistema. Este código puede detectar ciertos tipos de fallas antes de que causen daños. Es posible que se requieran mecanismos de recuperación para garantizar que el sistema continúe en ejecución.

En los sistemas para los que la disponibilidad es un requerimiento crítico, por lo general se usan servidores redundantes. Éstos entran en operación automáticamente al fallar un servidor designado. Algunas veces, para garantizar que los ataques al sistema no puedan detonar una vulnerabilidad común, dichos servidores son de diversos tipos y funcionan con sistemas operativos distintos. El uso de diferentes sistemas operativos es un ejemplo de diversidad y redundancia del software, donde la funcionalidad comparable se ofrece en diferentes formas. En la sección 13.3.4 se discute con más detalle la diversidad del software.

La diversidad y redundancia sirven también para lograr procesos confiables al garantizar que las actividades del proceso, como la validación de software, no se apoyen en un proceso o un método únicos. Esto aumenta la confiabilidad del software al reducir las posibilidades de falla del proceso, ya que los errores humanos cometidos durante el proceso de desarrollo de software conducen a errores en éste. Por ejemplo, las actividades de validación pueden incluir pruebas del programa, inspecciones manuales del mismo y análisis estático, como las técnicas para la detección de fallas en el desarrollo. Se trata de técnicas complementarias ya que permiten encontrar fallas en el desarrollo que soslayen los demás métodos. Más aún, diferentes miembros del equipo serían responsables por la

Procesos operacionales confiables

Este capítulo analiza los procesos de desarrollo confiables, aunque un factor contribuyente igualmente importante para la confiabilidad del sistema es un proceso operacional del sistema. Al diseñar tales procesos operacionales, se deben considerar los factores humanos y pensar siempre que los individuos son proclives a cometer errores cuando usan un sistema. Un proceso confiable tiene que diseñarse para evitar errores humanos y, al cometerse éstos, el software debe detectar los errores y permitir su corrección.

http://www.SoftwareEngineering-9.com/Web/DependabilityEng/HumanFactors/

misma actividad del proceso (por ejemplo, una inspección de programa). Las personas realizan las tareas en diferentes formas, dependiendo de su personalidad, experiencia y educación, de modo que este tipo de redundancia ofrece una perspectiva distinta al sistema.

Como se estudia en la sección 13.3.4, lograr la diversidad de software no es un proceso sencillo. La diversidad y redundancia vuelven a los sistemas más complejos y, por lo tanto, más difíciles de entender. No sólo hay un código adicional para escribir y verificar, también debe agregarse al sistema funcionalidad adicional, con la finalidad de detectar la falla del componente y cambiar el control a componentes alternativos. Esta complejidad adicional significa que es más probable que los programadores cometan errores y que sea menos factible que las personas que comprueban el sistema encuentren dichos errores.

Como consecuencia, algunas personas consideran que es mejor evitar la redundancia y la diversidad del software. Su visión es que el enfoque más adecuado es diseñar el software tan sencillo como sea posible, con procedimientos muy rigurosos de verificación y validación (Parnas *et al*., 1990). Es posible gastar más en verificación y validación gracias a los ahorros que resultan de no tener que desarrollar componentes de software redundantes.

Ambos enfoques se adoptan en sistemas comerciales críticos para la seguridad. Por ejemplo, el hardware y software de control de vuelo del Airbus 340 es tan diverso como redundante (Storey, 1996). El software de control de vuelo del Boeing 777 se basa en un hardware redundante, pero cada computadora opera el mismo software, que se validó ampliamente. Los diseñadores del sistema de control de vuelo del Boeing 777 se centraron en la simplicidad y no en la redundancia. Estas dos aeronaves son muy confiables, así que los enfoques diverso y simple para la confiabilidad pueden ser claramente exitosos.

13.2 Procesos confiables

Los procesos de software confiables están diseñados para producir software confiable. Una compañía que emplea un proceso confiable puede estar segura de que el proceso se realizó y documentó adecuadamente, y que se utilizaron técnicas de desarrollo adecuadas para el diseño de sistemas críticos. La razón para invertir en los procesos confiables es que es probable que un buen proceso de software conduzca a entregar un software con menos errores y, además, es menos probable que yerre en su ejecución. La figura 13.2 muestra algunos de los atributos de los procesos de software confiables.

La evidencia de que se usó un proceso confiable con frecuencia es importante para convencer a un regulador de que se aplicó la práctica de ingeniería de software más efectiva en el desarrollo del software. Por lo general, los desarrolladores de software presentarán un modelo del proceso a un regulador, junto con evidencia de que se siguió

Característica del proceso	Descripción
Documentable	El proceso debe tener un modelo de proceso definido que establezca las actividades en el proceso y la documentación que debe producirse durante dichas actividades.
Estandarizado	Tiene que estar disponible un conjunto amplio de estándares de desarrollo de software que cubren la producción y la documentación del software.
Auditable	El proceso debería ser comprensible para los individuos y los participantes del proceso, quienes pueden verificar el seguimiento de dichos estándares de proceso y hacer sugerencias para la mejora del proceso.
Diverso	El proceso debe comprender actividades de verificación y validación redundante y diversa.
Robusto	El proceso tiene que recuperarse de fallas de actividades de proceso individual.

Figura 13.2 Atributos de procesos confiables

el proceso. El regulador debe convencerse también de que los participantes en el proceso lo ponen en práctica de manera consistente y que éste puede usarse en diferentes proyectos de desarrollo. Esto significa que el proceso tiene que definirse explícitamente y ser repetible:

1. Un proceso definido explícitamente es aquel que sigue un modelo de proceso definido usado para orientar el proceso de producción de software. Debe haber recopilación de datos durante el proceso, que demuestre que se siguieron todos los pasos necesarios en el modelo de proceso.
2. Un proceso repetible es aquel que no se apoya en la interpretación y el juicio individuales. En vez de ello, el proceso suele repetirse a través de proyectos y con diferentes miembros de equipo, sin importar quién esté involucrado en el desarrollo. Esto es muy importante para sistemas críticos, que a menudo tienen un largo ciclo de desarrollo durante el cual con frecuencia existen cambios significativos en el equipo de desarrollo.

Los procesos confiables utilizan redundancia y diversidad para lograr fiabilidad. Tales procesos incluyen diferentes actividades que tienen la misma meta; por ejemplo, inspecciones y pruebas de programa dirigidos a descubrir errores en uno de éstos. Los enfoques son complementarios, además de que es probable que juntos descubran una mayor proporción de errores de los que se encontrarían usando una sola técnica.

Evidentemente, las actividades que se usan en los procesos confiables dependen del tipo de software que se va a desarrollar. Sin embargo, en general, dichas actividades deberían constituir un engranaje para evitar la introducción de errores en un sistema, para detectar y eliminar errores, y mantener la información sobre el proceso en sí. Los ejemplos de actividades que se incluyen en un proceso confiable son:

1. Revisiones de requerimientos para comprobar que, en la medida de lo posible, los requerimientos estén completos y sean consistentes.

El ciclo de vida de seguridad

La International Electrotechnical Commission dispuso un estándar de proceso (IEC 61508) para la ingeniería de sistemas de protección. Éste se basa en la noción de un ciclo de vida de seguridad, que hace una clara distinción entre ingeniería de seguridad e ingeniería de sistemas. Las primeras etapas del ciclo de vida de seguridad IEC 61508 definen el alcance del sistema, valoran los peligros potenciales del sistema y estiman los riesgos que plantean. A esto sigue la especificación de los requerimientos de seguridad y su asignación a diferentes subsistemas. La idea es limitar la extensión de la funcionalidad crítica de protección para permitir la aplicación de técnicas específicas en la ingeniería de sistemas críticos al desarrollo del sistema crítico de protección.

http://www.SoftwareEngineering-9.com/Web/SafetyLifeCycle/

2. Gestión de requerimientos para asegurarse de que los cambios a los requerimientos se controlan y que el efecto de los cambios propuestos a los requerimientos sea comprendido por aquellos desarrolladores afectados por el cambio.
3. Especificación formal, donde se crea y analiza un modelo matemático del software. En el capítulo 12 se estudiaron los beneficios de la especificación formal. Quizá su beneficio más importante sea que obliga a un análisis muy detallado de los requerimientos del sistema. Este análisis, por sí mismo, es probable que descubra problemas de requerimientos que pudieron perderse durante las revisiones de los requerimientos.
4. Modelado de sistema, en el que el diseño del software se documenta explícitamente como un conjunto de modelos gráficos; también se documentan dichos modelos y los vínculos entre los requerimientos.
5. Inspecciones de diseño y programa, las cuales consisten en que diferentes personas inspeccionan y comprueban diversas descripciones del sistema. Con frecuencia, las inspecciones se realizan con base en listas de verificación de errores comunes de diseño y programación.
6. Análisis estático, lo que implica realizar comprobaciones automatizadas sobre el código fuente del programa. En él se buscan anomalías que pudieran indicar errores u omisiones de programación. En el capítulo 15 se estudia el análisis estático.
7. Planeación y administración de pruebas, lo que implica diseñar un conjunto global de pruebas del sistema. El proceso de pruebas tiene que administrarse cuidadosamente para demostrar que tales pruebas ofrecen una cobertura de los requerimientos del sistema y que se aplican correctamente.

Así como hay actividades de proceso que se enfocan en el desarrollo y las pruebas del sistema, también debe haber procesos de administración de calidad y del cambio bien definidos. Aunque las actividades específicas en un proceso confiable pueden variar de una compañía a otra, la necesidad de administración de la calidad y del cambio es universal.

Los procesos de administración de la calidad (que se examinan en el capítulo 24) establecen un conjunto de estándares de proceso y producto. También incluyen actividades que captan información del proceso, con la finalidad de demostrar que se siguieron dichos estándares. Por ejemplo, podría haber un estándar definido para realizar inspecciones de programa. El líder del equipo de inspección es responsable de documentar el proceso para demostrar que se siguió el estándar de inspección.

La administración del cambio, que se estudia en el capítulo 25, se interesa por administrar los cambios a un sistema, lo cual garantiza que los cambios aceptados en realidad se implementen, y confirma que las versiones planeadas del software incluyan los cambios programados. Un problema común con el software es que se incluyan componentes equivocados en la construcción de un sistema. Esto llevaría a una situación en que un sistema en ejecución incluya componentes que no se verificaron durante el proceso de desarrollo. Los procedimientos de administración de la configuración tienen que definirse como parte del proceso de administración del cambio, con la finalidad de asegurar que esto no suceda.

Existe una visión muy difundida de que los enfoques ágiles, que se estudiaron en el capítulo 3, en realidad no son adecuados para los procesos de confiabilidad (Boehm, 2002). Los enfoques ágiles se enfocan en el desarrollo de software y no en documentar lo que se hizo. Con frecuencia tienen un enfoque bastante informal para la administración del cambio y la calidad. Por lo general, se prefieren los enfoques basados en un plan al desarrollo de sistemas confiables, que crean documentación fácil de entender por parte de los reguladores y otros participantes externos del sistema. No obstante, los beneficios de los enfoques ágiles son igualmente aplicables a los sistemas críticos. En esta área se han presentado reportes de éxito en la aplicación de métodos ágiles (Lindvall *et al*., 2004) y es probable que se desarrollen variantes de los métodos ágiles que sean adecuadas para la ingeniería de sistemas críticos.

13.3 Arquitecturas de sistemas confiables

Como vimos, el desarrollo de sistemas confiables debe basarse en un proceso confiable. Aunque probablemente se necesite de un proceso confiable para crear sistemas igualmente confiables, esto no es suficiente para garantizar la confiabilidad. También se requiere diseñar una arquitectura de sistema para la confiabilidad, sobre todo cuando se precisa tolerancia a fallas. Esto significa que la arquitectura tiene que diseñarse para incluir componentes y mecanismos redundantes, que permitan cambiar el control de un componente a otro.

Los ejemplos de sistemas que pueden requerir arquitecturas tolerantes a fallas son los sistemas en las aeronaves, que deben estar en operación durante todo el vuelo, los sistemas de telecomunicación, y los sistemas de comando y control críticos. Pullum (2001) describe los diferentes tipos de arquitectura tolerante a fallas que se han propuesto, y Torres-Pomales estudia las técnicas de tolerancia a fallas de software (2000).

La realización más simple de una arquitectura confiable está en los servidores replicados, cuando dos o más servidores realizan la misma tarea. Las peticiones para procesamiento se canalizan a través de un componente de gestión de servidor que enruta cada petición hacia un servidor específico. Este componente también hace un seguimiento de las respuestas del servidor. En caso de falla de éste, lo que por lo general se detecta mediante la falta de respuesta, el servidor defectuoso se desconecta del sistema. Las peticiones no procesadas se envían a otros servidores para su procesamiento.

Este enfoque de servidor replicado se usa ampliamente para los sistemas de procesamiento de transacciones, donde es fácil mantener copias de las transacciones a procesar. Los sistemas de procesamiento de transacción se diseñan para que sólo se actualicen una vez terminada correctamente la transacción, de manera que las demoras en el procesa-

miento no afecten la integridad del sistema. Ésta puede ser una forma efectiva de usar hardware, si el servidor de respaldo es el que se usa normalmente para tareas de baja prioridad. Si ocurre un problema con un servidor primario, su procesamiento se transfiere al servidor de respaldo, que otorga a dicho trabajo la mayor prioridad.

Los servidores replicados proporcionan redundancia, aunque usualmente no aportan diversidad. El hardware, en general, es idéntico y opera la misma versión del software. Por lo tanto, los servidores son capaces de lidiar con fallas de hardware y de software que se localizan en una sola máquina. No pueden contender con problemas de diseño de software que hacen que todas las versiones del software fallen al mismo tiempo. Para manejar fallas de diseño de software, un sistema debe incluir software y hardware diversos, como se expone en la sección 13.1.

La diversidad y redundancia del software pueden implementarse en algunos estilos arquitectónicos diferentes. En el resto de esta sección se describen algunos de ellos.

13.3.1 Sistemas de protección

Un sistema de protección es un sistema especializado que se asocia con algún otro sistema. Por lo común, éste es un sistema de control para algunos procesos, tales como un proceso de manufactura química o un sistema de control de equipo, por ejemplo el sistema en un tren sin conductor. Un ejemplo de un sistema de protección sería un sistema en un tren que detecte si éste pasa por una señal roja. Si es así, y no hay indicios de que el sistema de control del tren desacelera, entonces el sistema de protección aplica automáticamente los frenos del tren para detenerlo. Los sistemas de protección monitorizan de manera independiente el ambiente y, si los sensores indican un problema que el sistema controlado no detecta, entonces se activa el sistema de protección para desactivar el proceso o el equipo.

La figura 13.3 ilustra la relación entre un sistema de protección y un sistema controlado. El sistema de protección monitoriza tanto el equipo controlado como el ambiente. Si se detecta un problema, emite comandos al actuador para desactivar el sistema o solicitar otros mecanismos de protección, como la apertura de una válvula de desfogue. Observe que existen dos conjuntos de sensores. Un conjunto se usa para monitorización normal del sistema y el otro específicamente para el sistema de protección. En el caso de falla del sensor existen respaldos que permitirán al sistema de protección continuar en operación. También hay actuadores redundantes en el sistema.

Un sistema de protección sólo incluye la funcionalidad crítica que se requiere para cambiar el sistema de un estado potencialmente inseguro a un estado seguro (desactivación del sistema). Es un ejemplo más general de una arquitectura tolerante a fallas, donde un sistema principal recibe apoyo de un sistema de respaldo más pequeño y más simple que sólo incluye funcionalidad esencial. Por ejemplo, el software de control del transbordador espacial tiene un sistema de respaldo que incluye la funcionalidad “llevar a casa”; esto es, si falla el sistema de control principal, el sistema de respaldo logrará aterrizar el vehículo.

La ventaja de este tipo de arquitectura es que el software del sistema de protección suele ser mucho más sencillo que el software que controla el proceso a proteger. La única función del sistema de protección es la operación de monitorizar y asegurar que el sistema se lleve a un estado seguro en caso de una emergencia. Por lo tanto, es posible invertir más esfuerzo en evitar y detectar fallas. Se puede comprobar que la especificación

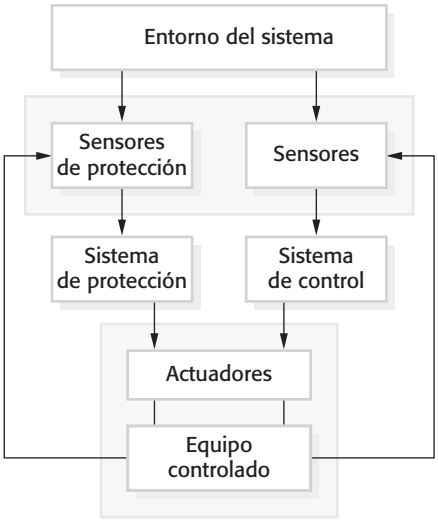


Figura 13.3 Arquitectura de un sistema de protección

del software es acertada y consistente, y que el software es correcto con respecto a su especificación. La meta es garantizar que la fiabilidad del sistema de protección sea tal que tiene una probabilidad muy baja de fallas en la petición (es decir, 0.001). Puesto que las demandas sobre el sistema de protección deben ser escasas, una probabilidad de falla en la petición de 1/1,000 significa que las fallas del sistema de protección, de hecho, deben ser muy extrañas.

13.3.2 Arquitecturas de automonitorización

Una arquitectura de automonitorización es una arquitectura de sistema en que éste se diseña para monitorizar su propia operación y tomar alguna acción al detectar un problema. Esto se logra al realizar cálculos sobre canales separados y comparar las salidas de dichos cálculos. Si las salidas son idénticas y están disponibles al mismo tiempo, entonces se juzga que el sistema opera correctamente. Si las salidas son diferentes, en tal caso se supone una falla. Cuando sucede esto último, el sistema normalmente declarará una excepción de falla en la línea de salida de estatus, que conducirá a la transferencia de control a otro sistema. Esto se ilustra en la figura 13.4.

Para ser efectivos en la detección de fallas de hardware y software, los sistemas de automonitorización tienen que diseñarse de forma que:

1. El hardware utilizado en cada canal sea diverso. En la práctica, esto significaría que cada canal use un tipo de procesador diferente para realizar los cálculos requeridos, o que los chips auxiliares que constituyen el sistema puedan provenir de diferentes fabricantes. Esto reduce la probabilidad de fallas comunes en el diseño del procesador que afecten el cálculo.
2. El software usado en cada canal es diverso. De otro modo, podría surgir el mismo error de software en cada canal al mismo tiempo. En la sección 13.3.4 se examinan las dificultades de lograr software verdaderamente diverso.

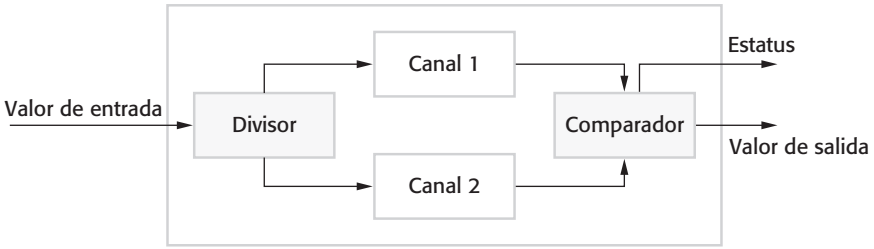


Figura 13.4 Arquitectura de automonitorización

Por sí sola, esta arquitectura puede usarse en situaciones donde es importante que los cálculos sean correctos, pero donde la disponibilidad no sea esencial. Si las respuestas de cada canal difieren, el sistema simplemente se desactiva. Para muchos sistemas de tratamiento y diagnóstico médico, la fiabilidad es más importante que la disponibilidad, pues una respuesta incorrecta del sistema podría conducir a que el paciente reciba un tratamiento equivocado. Sin embargo, si el sistema sencillamente se desactiva en el caso de un error, esto constituye un inconveniente, aunque el paciente por lo general no resulta dañado por el sistema.

En situaciones donde se requiere alta disponibilidad, se tienen que usar muchos sistemas de autoverificación en forma paralela. Se necesita una unidad de conmutación que detecte las fallas y seleccione un resultado de uno de los sistemas, donde ambos canales produzcan una respuesta consistente. Tal enfoque se usa en el sistema de control de vuelo para la serie Airbus 340 de aeronaves, donde se utilizan cinco computadoras de autoverificación. La figura 13.5 es un diagrama simplificado que ilustra esta organización.

En el sistema de control de vuelo del Airbus, cada una de las computadoras de control de vuelo realiza los cálculos en paralelo y usa las mismas entradas. Las salidas se conectan a filtros de hardware que detectan si el estatus indica una falla y, si es así, hacen que la salida de dicha computadora se desactive. Entonces, la salida se toma de un sistema alternativo. Por ello, es posible que cuatro computadoras fallen y que la operación de la aeronave continúe. En más de 15 años de operación, no ha habido reportes de situaciones donde el control de la aeronave se haya perdido debido a una falla total del sistema de control de vuelo.

Los diseñadores del sistema Airbus trataron de lograr diversidad en muchas formas diferentes:

1. Las computadoras primarias de control de vuelo usan un procesador diferente de los sistemas secundarios de control de vuelo.
2. Los chips auxiliares que se usan en cada canal en los sistemas primario y secundario son suministrados por fabricantes diferentes.
3. El software en los sistemas secundarios de control de vuelo sólo ofrece funcionalidad crítica: es menos complejo que el software primario.
4. El software para cada canal, tanto en los sistemas primarios como en los secundarios, se desarrolla mediante diferentes lenguajes de programación y por distintos equipos.
5. Se usan diversos lenguajes de programación en los sistemas secundario y primario.

Como se estudia en la siguiente sección, esto no garantiza la diversidad, pero reduce la probabilidad de fallas comunes en diferentes canales.

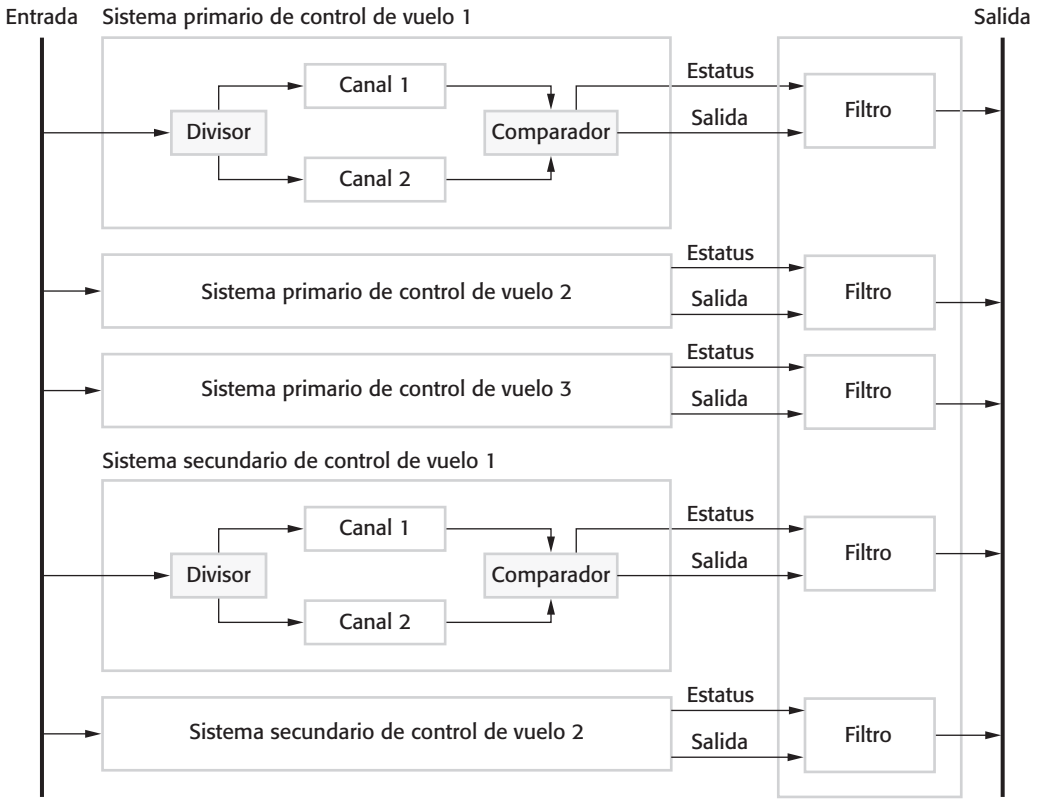


Figura 13.5 Arquitectura del sistema de control de vuelo del Airbus

13.3.3 Programación de n-versión

Las arquitecturas de automonitorización son ejemplos de sistemas en los que se usa la programación multiversión, con la finalidad de ofrecer redundancia y diversidad de software. Esta noción de programación multiversión se derivó de los sistemas de hardware, en los cuales durante muchos años se usó la noción de redundancia modular triple (TMR, por las siglas de *Triple Modular Redundancy*) para construir sistemas que son tolerantes a las fallas de hardware (figura 13.6).

En un sistema TMR, la unidad de hardware se replica tres veces (o en más ocasiones). La salida de cada unidad se pasa a un comparador de salida que se implementa por lo general como un sistema de votación. Este sistema compara todas sus entradas y, si dos o más son iguales, entonces dicho valor es la salida. Si una de las unidades falla y no produce la misma salida que las otras unidades, su salida se ignora. Un administrador de fallas en el desarrollo puede tratar de reparar automáticamente la unidad defectuosa, pero, si esto es imposible, el sistema se reconfigura en forma automática para sacar de servicio la unidad. Entonces el sistema continúa su función con dos unidades trabajando.

Este enfoque a la tolerancia a fallas se apoya en que la mayoría de las fallas de hardware son resultado de fallas en los componentes más que de fallas de diseño. En consecuencia, es probable que los componentes fallen de manera independiente. Se supone que, cuando son completamente operativas, todas las unidades de hardware se desempeñan de acuerdo con las especificaciones. Por consiguiente, hay baja probabilidad de falla simultánea de componentes en todas las unidades de hardware.

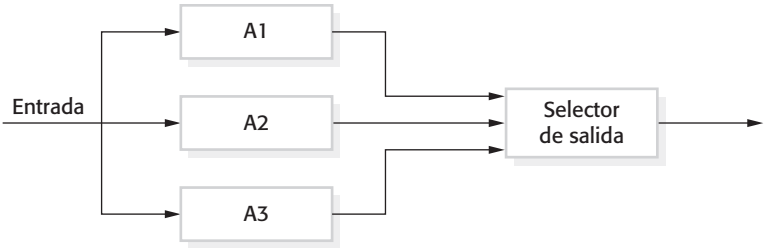


Figura 13.6 Redundancia modular triple

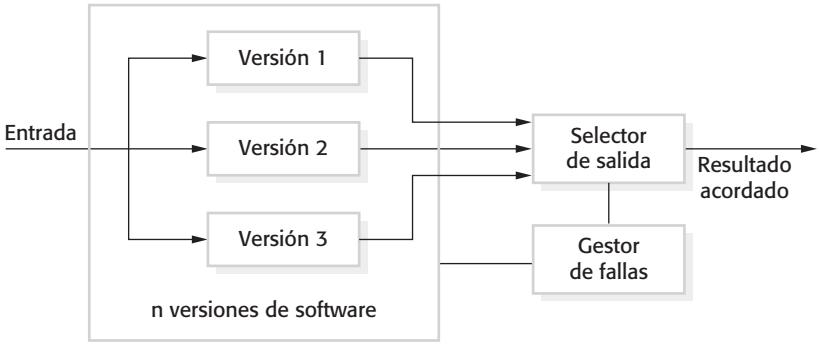


Figura 13.7 Programación de n-versión

Desde luego, todos los componentes podrían tener una falla de diseño común y, además, producir todos la misma respuesta (equivocada). Al usar unidades de hardware que tienen una especificación común, pero que se diseñan y construyen por fabricantes diferentes, se reducen de modo común las posibilidades de tal falla. Se supone que es mínima la probabilidad de que diferentes equipos cometan el mismo error de diseño o de fabricación.

Un enfoque similar puede usarse para software tolerante a fallas, donde n versiones diversas de un sistema de software se ejecutan en paralelo (Avizienis, 1985; Avizienis, 1995). Este enfoque a la tolerancia a fallas de software, que se ilustra en la figura 13.7, se usa en sistemas de señalización ferroviaria, sistemas de aeronaves y sistemas de protección de reactores.

Al usar una especificación común, el mismo sistema de software se implementa por algunos equipos. Dichas versiones se ejecutan en computadoras separadas. Sus salidas se comparan al usar un sistema de votación, y se rechazan las salidas inconsistentes o las que no se producen en tiempo. Al menos tres versiones del sistema deben estar disponibles, de modo que dos versiones tienen que ser consistentes en caso de una falla individual.

La programación de n-versión suele ser menos costosa que las arquitecturas de autoverificación, en sistemas para los cuales se requiere un alto nivel de disponibilidad. Sin embargo, todavía se requieren muchos equipos diferentes para desarrollar diferentes versiones del software. Esto conduce a costos de desarrollo de software muy elevados. Como resultado, este enfoque sólo se usa en sistemas donde no resulta práctico ofrecer un sistema de protección que proteja contra fallas críticas de seguridad.

13.3.4 Diversidad de software

Todas las arquitecturas anteriores tolerantes a fallas se apoyan en la diversidad del software para lograr tolerancia a fallas. Esto se basa en la suposición de que son independientes las implementaciones diversas de la misma especificación (o una parte de la especificación, para sistemas de protección). No deben incluir errores comunes y, además, no fallarán

en la misma forma al mismo tiempo. Esto requiere que el software lo escriban diferentes equipos y que no se comuniquen durante el proceso de desarrollo, lo cual, por consiguiente, reduce las posibilidades de malos entendidos o malas interpretaciones comunes de la especificación.

La compañía que procura el sistema puede incluir políticas explícitas de diversidad que tengan la intención de maximizar las diferencias entre versiones del sistema. Por ejemplo:

1. Al incluir requerimientos de que deben usarse diferentes métodos de diseño. Para ilustrar lo anterior, se puede solicitar a un equipo que produzca un diseño orientado a un objeto y a otro equipo un diseño orientado a una función (estructurado).
2. Al estipular que las implementaciones deben escribirse en diferentes lenguajes de programación. Por ejemplo, en un sistema de tres versiones, podrían usarse Ada, C++ y Java para escribir las versiones del software.
3. Al requerir el uso de diferentes herramientas y entornos de desarrollo para el sistema.
4. Al solicitar explícitamente el uso de diferentes algoritmos en algunas partes de la implementación. Sin embargo, esto limita la libertad del equipo de diseño y puede ser difícil reconciliar con los requerimientos de rendimiento del sistema.

Cada equipo de desarrollo debe trabajar con una especificación detallada del sistema (en ocasiones llamada *V-spec*), que se deriva de la especificación de requerimientos del sistema (Avizienis, 1995). Ésta debe ser suficientemente detallada como para garantizar que no haya ambigüedades en la especificación. Además de especificar la funcionalidad del sistema, la especificación detallada debe definir dónde deben generarse las salidas del sistema para su comparación.

De manera ideal, las diversas versiones del sistema no deben tener dependencias y, además, deberían fallar en formas completamente diferentes. Si éste es el caso, entonces la fiabilidad global de un sistema diverso se obtiene al multiplicar las fiabilidades de cada canal. De este modo, si cada canal tiene una probabilidad de fallas en la petición de 0.001, entonces la POFOD global de un sistema de tres canales (con todos los canales independientes) es un millón de veces mayor que la fiabilidad de un sistema de un solo canal.

Sin embargo, en la práctica, es imposible lograr completa independencia del canal. Se ha demostrado experimentalmente que equipos de diseño independientes cometen con frecuencia los mismos errores o interpretan mal las mismas partes de la especificación (Brilliant *et al*., 1990; Knight y Leveson, 1986; Leveson, 1995). Existen muchas razones para ello:

1. Los miembros de diferentes equipos suelen tener el mismo antecedente cultural y es posible que se hayan educado con el mismo enfoque y con los mismos libros de texto. Esto significa que pueden encontrar difícil de entender las mismas cosas, y tener dificultades comunes para comunicarse con expertos de dominio. Es muy posible que, independientemente, cometan los mismos errores y diseñen los mismos algoritmos para resolver un problema.
2. Si los requerimientos son incorrectos o si se basan en interpretaciones equivocadas acerca del entorno del sistema, entonces dichos errores se reflejarán en cada implementación del sistema.
3. En un sistema crítico, la V-spec es un documento detallado con base en los requerimientos del sistema, que ofrece detalles completos a los equipos sobre cómo debe

comportarse el sistema. No puede haber espacio para la interpretación por parte de los desarrolladores del software. Si existen errores en este documento, entonces se presentarán a todos los equipos de desarrollo y se implementarán en todas las versiones del sistema.

Una forma de reducir la posibilidad de errores comunes de especificación es desarrollar independientemente especificaciones detalladas para el sistema, y definir las especificaciones en diferentes lenguajes. Un equipo de desarrollo puede trabajar desde una especificación formal, otro desde un modelo de sistema basado en estado, y un tercero desde una especificación en lenguaje natural. Esto ayuda a evitar algunos errores de interpretación de la especificación, aunque no evita el problema de los errores de especificación. También introduce la posibilidad de errores en la traducción de los requerimientos, lo cual conduce a especificaciones inconsistentes.

En un análisis de los experimentos, Hatton (1997) concluye que un sistema de tres canales era entre cinco a nueve veces más fiable que un sistema de un solo canal. Concluye que las mejoras en la fiabilidad que podrían obtenerse al dedicar más recursos a una sola versión no coincidirían con esto y, además, es probable que los enfoques de n-versión conduzcan a sistemas más fiables que los enfoques de versión sencilla.

Sin embargo, lo que no está claro es si las mejoras en fiabilidad de un sistema multiversión ameritan los costos de desarrollo adicionales. Para muchos sistemas, los costos adicionales quizá no sean justificables, ya que un sistema de versión sencilla bien realizado sería suficientemente bueno. Sólo en los sistemas críticos para la seguridad y la misión, donde los costos de las fallas son muy elevados, se requeriría software multiversión. Incluso en tales situaciones (por ejemplo, un sistema de nave espacial), sería suficiente al proporcionar un simple respaldo con funcionalidad limitada, hasta que el sistema principal pueda reparase y reiniciarse.

13.4 Programación confiable

Por lo general, en este libro se evitan las discusiones de la programación, porque es casi imposible discutir este tema sin entrar en detalles de un lenguaje de programación específico. Ahora existen tantos enfoques y lenguajes distintos usados en el desarrollo de software, que se evita el uso de un solo lenguaje para los ejemplos de este libro. Sin embargo, cuando se considera la ingeniería de confiabilidad, hay un conjunto de prácticas de programación recomendables y aceptadas, que son bastante universales y ayudan a reducir las fallas de los sistemas entregados.

En la figura 13.8 se presenta una lista de buenos lineamientos prácticos. Pueden aplicarse en cualquier lenguaje de programación que se use para el desarrollo de sistemas, aunque la forma en que se utilizan depende de los lenguajes y las notaciones específicos que se empleen para el desarrollo del sistema.

Lineamiento 1: Controlar la visibilidad de la información en un programa

Un principio de seguridad que adoptan las organizaciones militares es el principio de "necesidad de conocer". Sólo a aquellos individuos que necesiten conocer una parte de información en particular, para realizar sus labores, se otorga dicha información. Se oculta aquella información que no es directamente relevante para su trabajo.

Lineamientos de programación confiable

1. Controlar la visibilidad de la información en un programa.
2. Comprobar la validez de todas las entradas.
3. Proporcionar un manejador para todas las excepciones.
4. Minimizar el uso de códigos proclives a error.
5. Ofrecer capacidades de reinicio.
6. Comprobar los límites.
7. Incluir interrupciones cuando se soliciten componentes externos.
8. Nombrar todas las constantes que representan valores del mundo real.

Figura 13.8 Lineamientos de buena práctica para programación confiable

Cuando uno programa debe adoptar un principio análogo para controlar el acceso a las variables y estructuras de datos que se utilizan. A los componentes del programa sólo se les debe permitir el acceso a los datos que necesitan para su implementación. Otros datos de programa deben ser inaccesibles y ocultarse de ellos. Si se oculta información, ésta no se puede corromper por componentes de programa que se supone que no deben usarlos. Si la interfaz permanece invariable, la representación de datos debería cambiarse sin afectar a otros componentes en el sistema.

Esto se logra al implementar estructuras de datos en el programa como tipos de datos abstractos. Un tipo de datos abstractos es un tipo de datos en que la estructura interna y la representación de una variable de dicho tipo están ocultas. La estructura y los atributos del tipo no son visibles externamente y todo el acceso a los datos es a través de operaciones. Por ejemplo, se puede tener un tipo de datos abstractos que represente una cola de peticiones de servicio. Las operaciones deben incluir *get* (conseguir) y *put* (poner), que agregan y eliminan objetos de la cola, y una operación que regrese el número de objetos en la cola. Inicialmente se implementaría la cola como un arreglo, pero posteriormente cambiaría la implementación a una lista entrelazada. Esto se logra sin cambio alguno al código que usa la cola, porque nunca se accede de manera directa a la representación de la cola.

También se pueden usar tipos de datos abstractos para implementar comprobaciones de que un valor asignado esté dentro del rango. Por ejemplo, suponga que quiere representar la temperatura de un proceso químico, en que las temperaturas permitidas se hallan dentro del rango de 20 a 200 grados Celsius. Al incluir una comprobación del valor a asignar dentro de la operación de tipo de datos abstractos, garantizaría que el valor de la temperatura nunca está fuera del rango requerido.

En algunos lenguajes orientados a objetos es posible implementar tipos de datos abstractos mediante definiciones de interfaz, en las que se declara la interfaz a un objeto sin referencia a su implementación. Por ejemplo, se puede definir una interfaz *Queue* (cola), que soporta métodos para colocar objetos en la cola, eliminarlos de ésta y consultar el tamaño de la misma. En la clase de objetos que implementa esta interfaz, los atributos y los métodos para dicha clase deben ser privados.

Lineamiento 2: Comprobar la validez de todas las entradas

Todos los programas toman entradas de su entorno y las procesan. La especificación hace suposiciones sobre dichas entradas, que reflejan su uso en el mundo real. Por ejemplo, se supone que un número de cuenta bancaria siempre es un entero positivo de ocho dígitos. Sin embargo, en muchos casos, la especificación del sistema no define qué accio-

nes deberían tomarse si la entrada es incorrecta. Inevitablemente, los usuarios cometerán errores y algunas veces ingresarán datos equivocados. En ocasiones, como se estudia en el capítulo 14, ataques maliciosos al sistema se apoyan en el hecho de ingresar deliberadamente la entrada incorrecta. Aun cuando la entrada provenga de sensores u otros sistemas, estos sistemas pueden estar equivocados y proporcionar valores incorrectos.

Por consiguiente, siempre se debe verificar la validez de las entradas tan pronto como éstas se lean del entorno de los programas en operación. Naturalmente, las comprobaciones implicadas dependen de las mismas entradas, pero las posibles comprobaciones que pueden usarse son las siguientes:

1. *Comprobaciones de rango* Se puede esperar que las entradas estén dentro de un rango específico. Por ejemplo, una entrada que represente una probabilidad debe estar dentro del rango 0.0 a 1.0; una entrada que represente la temperatura de agua líquida debería estar entre 0 y 100 grados Celsius, etcétera.
2. *Comprobaciones de tamaño* Se puede esperar que las entradas sean de un número dado de caracteres (por ejemplo, ocho caracteres para representar una cuenta bancaria). En otros casos, el tamaño quizá no sea fijo, pero debe haber un límite superior real. Por ejemplo, es improbable que el nombre de una persona tenga más de 40 caracteres.
3. *Comprobaciones de representación* Cabe esperar que una entrada sea de un tipo particular, que se representa en forma estándar. Por ejemplo, los nombres de las personas no incluyen caracteres numéricos, las direcciones de correo electrónico constan de dos partes, separadas por un signo @, etcétera.
4. *Comprobaciones de racionalidad* Si una entrada es una serie y usted conoce algo sobre las relaciones entre los miembros de ésta, entonces, podrá comprobar que un valor de entrada es razonable. Por ejemplo, si el valor de entrada representa la lectura del medidor de electricidad doméstico, en tal caso esperaría que la cantidad de energía eléctrica consumida sea casi la misma que en el periodo correspondiente del año anterior. Desde luego, habrá variaciones, pero diferencias en el orden de magnitud sugieren que surgió un problema.

Las acciones que se toman si falla una comprobación de validación de entrada dependen del tipo de sistema que se va a implementar. En algunos casos, el usuario reporta el problema y solicita el reingreso del valor. Cuando un valor provenga de un sensor, puede usar el valor válido más reciente. En los sistemas embebidos de tiempo real, quizá deba estimar el valor con base en la historia, de modo que el sistema pueda continuar en operación.

Lineamiento 3: Proporcionar un manejador para todas las excepciones

Durante la ejecución del programa, inevitablemente ocurren errores o eventos inesperados. Éstos surgirían debido a una falla en el desarrollo del programa o podrían ser el resultado de circunstancias externas impredecibles. Un error o un evento inesperado que ocurre durante la ejecución de un programa se llama “excepción”. Ejemplos de excepciones son una falla eléctrica del sistema, un intento por ingresar datos inexistentes, o el desbordamiento o subdesbordamiento numéricos.

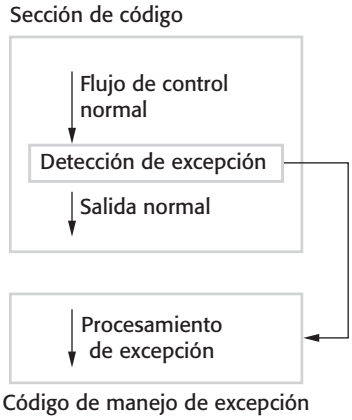


Figura 13.9 Manejo de excepción

Las excepciones suelen originarse por las condiciones del hardware o software. Cuando ocurre una excepción, debe ser gestionada por el sistema. Esto puede hacerse dentro del mismo programa o al realizar la transferencia del control a un mecanismo de manejo de excepción del sistema. Por lo regular, el mecanismo de gestión de excepciones del sistema reporta el error y desactiva la ejecución. Por consiguiente, para garantizar que las excepciones de programa no causen la falla de sistema, hay que definir un manejador de excepciones para todas las posibles excepciones que surjan, y asegurarse de que todas las excepciones se detecten y manejen explícitamente.

En lenguajes de programación como C, deben usarse enunciados *if* para detectar excepciones y transferir el control al código de manejo de excepciones. Esto significa que hay que comprobar explícitamente las excepciones que ocurran en cualquier parte del programa. Sin embargo, este enfoque agrega complejidad significativa a la tarea del manejo de excepciones, lo cual aumenta las posibilidades de que se cometan errores y, por consiguiente, de que se manejen inadecuadamente las excepciones.

Algunos lenguajes de programación, como Java, C++ y Ada, incluyen constructos que soportan manejo de excepciones, además de que no necesitan enunciados condicionales adicionales para comprobar las excepciones. Dichos lenguajes de programación incluyen un tipo interno especial (llamado con frecuencia *Exception*), y diferentes excepciones pueden declararse como de este tipo. Cuando ocurre una situación excepcional, se señala la excepción y el lenguaje del sistema de tiempo de ejecución transfiere el control a un manejador de excepciones. Ésta es una sección de código que enuncia los nombres de excepciones y las acciones adecuadas para manejar cada una (figura 13.9). Observe que el manejador de excepciones está afuera del flujo de control normal, y que este flujo de control normal no se reanuda después de manejar la excepción.

Los manejadores de excepciones, por lo general, efectúan una o más de las siguientes tres acciones:

1. Indicar a un componente de nivel superior que ocurrió una excepción, y brindar información a dicho componente sobre el tipo de excepción. Este enfoque se usa cuando un componente solicita otro, y el componente solicitante necesita saber si el componente solicitado se ejecutó exitosamente. Si no, queda en manos del componente solicitante tomar acciones para recuperarse del problema.

2. Realizar algún procesamiento alternativo al que originalmente se pretendía. Así, el manejador de excepciones realiza algunas acciones para recuperarse del problema. Entonces es capaz de continuar el procesamiento normal, o bien, el manejador de excepciones indica que ocurrió una excepción, de modo que un componente solicitante está al tanto del problema.

3. Pasar el control a un sistema de apoyo en tiempo de ejecución que maneje la excepción. Con frecuencia esto sucede cuando ocurren fallas en un programa (por ejemplo, cuando se desborda un valor numérico). La acción común del sistema en tiempo de ejecución es detener el procesamiento. Sólo debe usarse este enfoque cuando sea posible llevar al sistema a un estado seguro e inactivo, antes de guiar el control al sistema en tiempo de ejecución.

El manejo de excepciones dentro de un programa posibilita la detección y la recuperación de algunos errores de entrada y de eventos externos inesperados. Como tal, ofrece un grado de tolerancia a la falla: el programa detecta las fallas y toma acciones para recuperarse de ellas. Como la mayoría de los errores de entrada y eventos externos inesperados por lo general son transitorios, a menudo es posible continuar la operación normal después de procesar la excepción.

Lineamiento 4: Minimizar el uso de sentencias proclives a error

Las fallas en los programas y, por consiguiente, muchas fallas en la operación del programa, comúnmente son consecuencia de un error humano. Los programadores cometen errores porque pierden la pista de las numerosas relaciones entre las variables de estado. Escriben enunciados de programa que dan como resultado un comportamiento inesperado y cambios de estado del sistema. Las personas siempre cometerán errores, pero a finales de la década de 1970 quedó claro que algunos enfoques a la programación tenían más probabilidad de introducir errores en un programa que otros.

Algunos constructos de lenguaje de programación y técnicas de programación son inherentemente proclives a error y, por lo tanto, deben evitarse o, al menos, usarse lo mínimo que sea posible. Los constructos potencialmente proclives a error incluyen:

1. *Enunciados de ramificación incondicional (go-to)* Los peligros de los enunciados go-to se reconocieron desde 1968 (Dijkstra, 1968) y, en consecuencia, se excluyen de los lenguajes de programación modernos. Sin embargo, todavía se permiten en lenguajes como C. El uso de enunciados go-to conduce a un "código espagueti" que está enmarañado y dificulta la comprensión y la depuración.

2. *Números con punto flotante* La representación de números con punto flotante en una palabra de memoria de longitud fija es inherentemente imprecisa. Éste es un problema específico cuando los números se comparan, porque la imprecisión de la representación conduce a comparaciones inválidas. Por ejemplo, 3.00000000 puede representarse a veces como 2.99999999 y en ocasiones como 3.00000001. Una comparación mostraría que no son iguales. Los números con punto fijo, donde un número se representa a un número dado de lugares decimales, por lo general son más seguros porque permiten comparaciones exactas.

3. *Apuntadores* Los lenguajes de programación como C y C++ soportan sentencias de bajo nivel llamados apuntadores, que retienen direcciones que se refieren directamente

a áreas de la memoria de la máquina (apuntan a una ubicación de memoria). Los errores en el uso de apuntadores suelen ser devastadores si se establecen de manera incorrecta y, en consecuencia, apuntan al área de memoria equivocada. También hacen más difícil la implementación de comprobaciones de límites y otras estructuras.

4. *Asignación de memoria dinámica* La memoria del programa puede asignarse a tiempo de ejecución, en vez de a tiempo de compilación. El peligro con esto es que es posible no desasignar la memoria adecuadamente, de modo que con el tiempo se agota la memoria disponible. Quizás esto sea un error muy difícil de detectar, porque el sistema podría operar exitosamente durante mucho tiempo antes de que ocurra el problema.

5. *Paralelismo* Cuando los procesos se ejecutan concurrentemente, puede haber sutiles dependencias de temporización entre ellos. Los problemas de temporización por lo general no suelen detectarse mediante inspección del programa, en tanto que la combinación peculiar de circunstancias que causan un problema de temporización quizá no ocurra durante las pruebas de un sistema. El paralelismo puede ser inevitable, pero su uso tiene que controlarse cuidadosamente para minimizar las dependencias entre procesos.

6. *Recursión* Cuando un procedimiento o método se solicita a sí mismo o solicita otro procedimiento, que entonces solicita al procedimiento solicitante original, esto se llama "recursión". El uso de recursión puede derivar en programas concisos; sin embargo, sería difícil seguir la lógica de los programas recursivos. Por lo tanto, los errores de programación son más difíciles de detectar. Los errores de recursión podrían dar como resultado la asignación de toda la memoria del sistema conforme se crean variables en pilas temporales.

7. *Interrupciones* Se trata de un medio para forzar la transferencia del control a una sección de código, sin importar si el código se ejecuta en la actualidad. Los peligros de esto son evidentes; la interrupción llega a provocar la terminación de una operación crítica.

8. *Herencia* El problema con la herencia en la programación orientada a objetos es que el código asociado con un objeto no está todo en un lugar. Esto hace más difícil de entender el comportamiento del objeto. De ahí que sea más probable que se pierdan errores de programación. Más aún, cuando se combina con enlaces dinámicos, la herencia puede causar problemas de temporización en tiempo de ejecución. Diferentes instancias de un método pueden ligarse a una petición, dependiendo de los tipos de parámetro. En consecuencia, se emplearán diferentes cantidades de tiempo en busca de la instancia del método correcto.

9. *Alias* Esto ocurre cuando se usa más de un nombre para referirse a la misma entidad en un programa; por ejemplo, cuando dos apuntadores con diferentes nombres señalan la misma ubicación de memoria. Es fácil que los lectores del programa pierdan enunciados que cambian la entidad cuando tienen varios nombres por considerar.

10. *Arreglos sin límites* En lenguajes como C, los arreglos simplemente son formas de acceder a la memoria y usted puede hacer asignaciones más allá del final de un arreglo. El sistema de tiempo de ejecución no comprueba que las asignaciones se refieran realmente a elementos en el arreglo. El desbordamiento de *buffer*, en que un atacante construye deliberadamente un programa para escribir memoria más allá del final de un *buffer* implementado como arreglo, es una conocida vulnerabilidad en la seguridad.

11. *Procesamiento de entrada por defecto* Algunos sistemas ofrecen un procesamiento de entrada por defecto, sin importar la entrada que se presente al sistema. Éste es un hueco de seguridad que un atacante aprovecharía al presentar el programa con entradas inesperadas que no rechaza el sistema.

Algunos estándares para el desarrollo de sistemas de seguridad críticos prohíben por completo el uso de estos constructos. Sin embargo, tal posición extrema generalmente no es práctica. Todos estos constructos y técnicas son útiles, aunque deben usarse con cuidado. Siempre que sea posible, sus efectos potencialmente riesgosos deben controlarse mediante el uso de tipos de datos abstractos u objetos, que actúan como "firewalls" naturales que limitan el daño causado si ocurren errores.

Lineamiento 5: Ofrecer capacidades de reinicio

Muchos sistemas de información organizacional se basan en transacciones cortas, en las que el procesamiento de entradas del usuario tarda un tiempo relativamente breve. Dichos sistemas se diseñan de modo que los cambios a la base de datos del sistema sólo finalizan después de que todos los demás procesamientos se completan con éxito. Si algo sale mal durante el procesamiento, la base de datos no se actualiza y, así, no se vuelve inconsistente. Prácticamente todos los sistemas de comercio electrónico (e-commerce), donde uno sólo se compromete con la compra en la pantalla final, trabajan de esta forma.

Las interacciones del usuario con los sistemas de comercio electrónico duran por lo general algunos minutos y requieren procesamiento mínimo. Las transacciones en la base de datos son breves y con frecuencia se completan en menos de un segundo. Sin embargo, otros tipos de sistemas, como los sistemas de CAD y los de procesamiento de texto, necesitan transacciones amplias. En un sistema de transacción amplia, el tiempo entre comenzar a usar el sistema y terminar de trabajar tomaría varios minutos o incluso horas. Si el sistema falla durante una transacción larga, entonces podría perderse todo el trabajo. De igual modo, en sistemas computacionalmente intensivos, como algunos sistemas de ciencia electrónica (e-science), tal vez se requieran minutos u horas de procesamiento para completar el cálculo. Todo este tiempo se pierde en caso de una falla del sistema.

En todos estos tipos de sistemas se debe proporcionar una capacidad de reinicio, que se basa en la conservación de copias de los datos que se recopilan o generan durante el procesamiento. La instalación de reinicio tiene que permitir al sistema reiniciar mediante dichas copias, en vez de tener que comenzar todo desde el principio. Dichas copias se conocen en ocasiones como *checkpoints* (punto de control o de verificación). Por ejemplo:

1. En un sistema de comercio electrónico se pueden conservar copias de los formularios llenados por un usuario, y permitir a éste el acceso y envío de dichos formularios sin tener que llenarlos nuevamente.

2. En una transacción larga o un sistema computacionalmente intensivo se pueden guardar automáticamente los datos cada determinada cantidad de minutos y, en caso de falla del sistema, reiniciar con los datos guardados más recientemente. También se deben permitir errores de usuario y proporcionar una forma de que los usuarios regresen al *checkpoint* más reciente y comiencen de nuevo desde ahí.

Si ocurre una excepción y es imposible continuar la operación normal, se puede manejar la excepción mediante recuperación de error hacia atrás. Esto significa que se resta-

blece el estado del sistema al estado guardado en el *checkpoint* y se reinicia la ejecución a partir de dicho punto.

Lineamiento 6: Comprobar los límites de los arreglos

Todos los lenguajes de programación permiten la especificación de arreglos: estructuras de datos secuenciales a los que se accede a través de un índice numérico. Dichos arreglos se suelen encontrar en áreas contiguas dentro de la memoria operativa de un programa. Los arreglos se especifican en un tamaño particular, que refleja cómo se usan. Por ejemplo, si se quieren representar las edades de hasta 10,000 personas, entonces se declara un arreglo con 10,000 ubicaciones para contener los datos de la edad.

Algunos lenguajes de programación, como Java, siempre comprueban que, cuando se ingresa un valor en un arreglo, el índice esté dentro de dicho arreglo. De este modo, si un arreglo A está indexado de 0 a 10,000, un intento por ingresar valores en los elementos A [-5] o A [12345] conducirá al surgimiento de una excepción. Sin embargo, lenguajes de programación como C y C++ no incluyen automáticamente comprobaciones de límite de arreglo y sólo calculan un corrimiento (*offset*) desde el comienzo del arreglo. Por consiguiente, A [12345] accedería a la palabra que tuviera la ubicación 12345 desde el comienzo del arreglo, sin importar si ésta era parte del arreglo o no.

La razón por la que estos lenguajes no incluyen comprobación automática de límite de arreglo es que ello introduce una sobrecarga cada vez que se accede al arreglo. La mayoría de los accesos al arreglo son correctos, de modo que la comprobación de límite generalmente es innecesaria y aumenta el tiempo de ejecución del programa. No obstante, la falta de comprobación de límites conduce a vulnerabilidades en la seguridad, como el desbordamiento de *buffer*, que se estudia en el capítulo 14. De manera más general, introduce una vulnerabilidad al sistema que puede derivar en falla de sistema. Si se usa un lenguaje que no incluye comprobación de límite de arreglo, siempre se debe incluir un código adicional que garantice que el índice del arreglo está dentro de los límites. Esto se logra con facilidad al implementar el arreglo como un tipo de datos abstractos, como se analizó en el lineamiento 1.

Lineamiento 7: Incluir interrupciones cuando se soliciten componentes externos

En los sistemas distribuidos, los componentes del sistema se ejecutan en diferentes computadoras y las peticiones se realizan a través de la red de componente a componente. Para recibir algún servicio, el componente A lo pedirá al componente B. A espera la respuesta de B antes de continuar la ejecución. Sin embargo, si el componente B falla en responder por alguna razón, entonces el componente A no continúa. Tan sólo espera indefinidamente una respuesta. Una persona que espere una respuesta del sistema observa una falla de sistema silenciosa, sin respuesta del sistema. No tiene alternativa más que aniquilar el proceso de espera y reiniciar el sistema.

Para evitar lo anterior, siempre hay que incluir interrupciones cuando se soliciten componentes externos. Una interrupción (*timeout*) es una suposición automática de que un componente solicitado falló y no producirá una respuesta. Se define un periodo durante el cual espera recibir una respuesta de un componente solicitado. Si no recibe una respuesta en ese lapso, supone una falla y retira el control del componente solicitado. Entonces puede tratar de recuperarse de la falla o indicar al usuario del sistema lo que sucedió y permitirle decidir qué hacer.

Lineamiento 8: Nombrar todas las constantes que representan valores del mundo real

Todos los programas no triviales incluyen un número de valores constantes que representan los valores de entidades del mundo real. Dichos valores no se modifican conforme se ejecuta el programa. En ocasiones, se trata de constantes absolutas que nunca cambian (por ejemplo, la velocidad de vuelo), sino que con más frecuencia son valores que cambian con relativa lentitud en el tiempo. Por ejemplo, un programa para calcular impuestos personales incluirá constantes que son las tasas impositivas actuales. Esto cambia de un año a otro y, por consiguiente, el programa debe actualizarse con los nuevos valores constantes.

Siempre se debería incluir en el programa una sección donde se mencionen todos los valores constantes que se usan del mundo real. Cuando se utilizan las constantes, hay que referirse a éstas por sus nombres y no por su valor. Esto tiene dos ventajas, en cuanto concierne a la confiabilidad:

1. Existe menos probabilidad de cometer errores y de usar el valor equivocado. Es fácil escribir mal un número y el sistema con frecuencia no podrá detectar un error. Por ejemplo, suponga que una tasa fiscal es del 34 por ciento. Un error de transposición simple llevaría a que esto se escribiera mal como 43 por ciento. Sin embargo, si se escribe mal un nombre (como el de tasa fiscal estándar), esto usualmente lo detecta el compilador como una variable sin declarar.
2. Cuando cambia un valor, no hay que buscar a través de todo el programa para descubrir dónde tiene que usar dicho valor. Todo lo que se necesita es cambiar el valor asociado con la declaración constante. Entonces, el nuevo valor se incluye automáticamente en todas las partes donde se necesite.

PUNTOS CLAVE

- La confiabilidad en un programa puede lograrse al evitar la introducción de fallas, al detectar y eliminar éstas antes de la implementación del sistema, y al incluir mecanismos de tolerancia a fallas que permitan al sistema permanecer operacional después de que una falla en el desarrollo causó una falla en la operación del sistema.
- El empleo de redundancia y diversidad en el hardware, los procesos de software y los sistemas de software es esencial para el desarrollo de sistemas confiables.
- El uso de un proceso repetible bien definido es esencial si las fallas en un sistema tienen que minimizarse. El proceso debe incluir en todas las etapas actividades de verificación y validación, desde la definición de requerimientos hasta la implementación del sistema.
- Las arquitecturas de sistema confiable son aquellas que se diseñan para tolerancia a fallas. Existen algunos estilos arquitectónicos que soportan tolerancia a fallas e incluyen sistemas de protección, arquitecturas de automonitorización y programación de n-versión.
- La diversidad de software es difícil de lograr, ya que es prácticamente imposible garantizar que cada versión del software sea en verdad independiente.

- La programación confiable se apoya en la inclusión de redundancia en un programa, con la finalidad de verificar la validez de las entradas y los valores de las variables del programa.
- Algunas sentencias y técnicas de programación, como enunciados *go-to*, apuntadores, recursión, herencia y números de punto flotante, son inherentemente proclives al error. Hay que tratar de evitar dichos constructos para desarrollar sistemas confiables.

LECTURAS SUGERIDAS

Software Fault Tolerance Techniques and Implementation. Un análisis global de técnicas para lograr tolerancia a fallas de software y arquitecturas tolerantes a las fallas. El libro también cubre temas generales de confiabilidad del software. (L. L. Pullum, Artech House, 2001.)

"Software Reliability Engineering: A Roadmap". Este ensayo por parte de un investigador líder en fiabilidad del software resume la vanguardia en la ingeniería de fiabilidad del software, así como un estudio de futuros retos en la investigación. (M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.24.

EJERCICIOS

13.1. Mencione cuatro razones por las que difícilmente sería efectivo en cuanto a costo el hecho de que las compañías garanticen que su software esté libre de fallas.

13.2. Explique por qué es razonable suponer que el uso de procesos confiables conducirá a la creación de software confiable.

13.3. Ofrezca dos ejemplos de actividades diversas y redundantes que puedan incorporarse en los procesos confiables.

13.4. ¿Cuál es la característica común de todos los estilos arquitectónicos que se conjuntan para soportar tolerancia a fallas de software?

13.5. Imagine que usted implementa un sistema de control basado en software. Sugiera circunstancias en las cuales sería adecuado usar una arquitectura tolerante a fallas y explique por qué se requeriría este enfoque.

13.6. Usted es el responsable del diseño de un conmutador de comunicaciones que tiene que ofrecer disponibilidad 24/7, pero que no es crítico para la protección. Ofrezca razones para su respuesta y sugiera un estilo arquitectónico para dicho sistema.

13.7. Se sugiere que el software de control para una máquina de terapia de radiación, usado para tratar pacientes con cáncer, debe implementarse mediante programación de n-versión. Comente sobre si considera que ésta es una buena sugerencia o no.

13.8. Mencione dos razones por las que diferentes versiones de un sistema, basadas en diversidad de software, podrían fallar en forma similar.

13.9. Explique por qué se deben manejar explícitamente todas las excepciones en un sistema cuya intención es lograr un nivel de disponibilidad alto.

13.10. El uso de técnicas para la producción de software seguro, como se estudió en este capítulo, naturalmente incluye considerables costos adicionales. ¿Qué costos adicionales se justificarían si 100 vidas se salvarían durante los 15 años de vida de un sistema? ¿Los mismos costos se justificarían si se salvaran 10 vidas? ¿Cuánto vale una vida? ¿Las capacidades de ingreso de las personas afectadas hacen una diferencia en este juicio?

REFERENCIAS

Avizienis, A. (1985). "The N-Version Approach to Fault-Tolerant Software". *IEEE Trans. on Software Eng*., **SE-11** (12), 1491–501.

Avizienis, A. A. (1995). "A Methodology of N-Version Programming". In *Software Fault Tolerance*. Lyu, M. R. (ed.). Chichester: John Wiley & Sons. 23–46.

Boehm, B. (2002). "Get Ready for Agile Methods, With Care". *IEEE Computer*, **35** (1), 64–9.

Brilliant, S. S., Knight, J. C. y Leveson, N. G. (1990). "Analysis of Faults in an N-Version Software Experiment". *IEEE Trans. On Software Engineering*, **16** (2), 238–47.

Dijkstra, E. W. (1968). "Goto statement considered harmful". *Comm. ACM*., **11** (3), 147–8.

Hatton, L. (1997). "N-version design versus one good version". *IEEE Software*, **14** (6), 71–6.

Knight, J. C. y Leveson, N. G. (1986). "An experimental evaluation of the assumption of independence in multi-version programming". *IEEE Trans. on Software Engineering*., **SE-12** (1), 96–109.

Leveson, N. G. (1995). Safeware: *System Safety and Computers. Reading*, Mass.: Addison-Wesley.

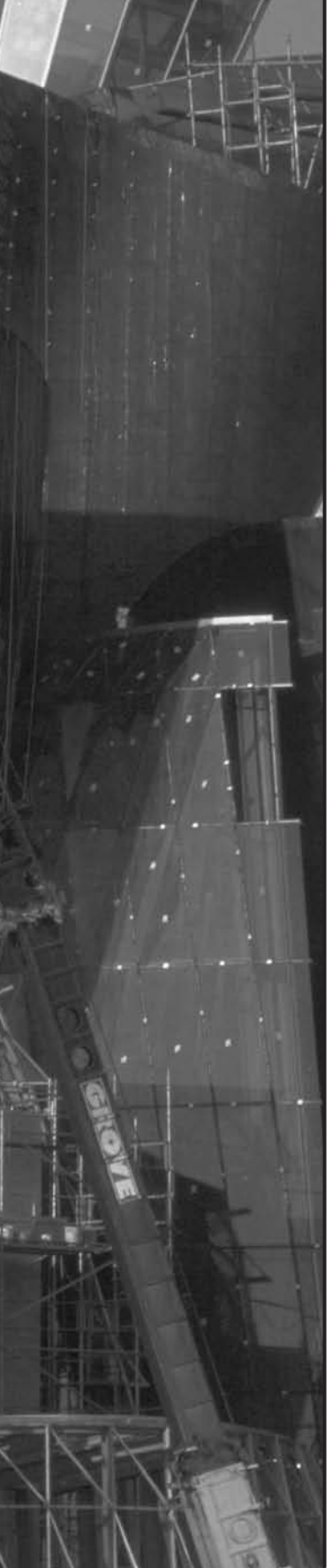
Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. y Kahkonen, T. (2004). "Agile Software Development in Large Organizations". *IEEE Computer*, **37** (12), 26–34.

Parnas, D. L., Van Schouwen, J. y Shu, P. K. (1990). "Evaluation of Safety-Critical Software". *Comm. ACM*, **33** (6), 636–51.

Pullum, L. L. (2001). *Software Fault Tolerance Techniques and Implementation*. Norwood, Mass.: Artech House.

Storey, N. (1996). *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Torres-Pomales, W. (2000). "Software Fault Tolerance: A Tutorial." http://ntrs.nasa.gov/archive/nasa/casi./20000120144_2000175863.pdf.



14

Ingeniería de seguridad

Objetivos

El objetivo de este capítulo es introducirlo a los temas que deben considerarse cuando se diseñan sistemas de aplicación segura. Al estudiar este capítulo:

- comprenderá la diferencia entre seguridad de aplicación y seguridad de infraestructura;
- identificará cómo la valoración del riesgo del ciclo de vida y la valoración del riesgo operativo permiten entender los conflictos de seguridad que afectan el diseño de un sistema;
- tendrá en cuenta las arquitecturas de software y los lineamientos de diseño para el desarrollo de sistemas seguros;
- aprenderá la noción de supervivencia del sistema y por qué es importante el análisis de supervivencia para sistemas de software complejos.

Contenido

14.1 Gestión del riesgo de seguridad
14.2 Diseño para la seguridad
14.3 Supervivencia del sistema

El uso extendido de Internet en la década de 1990 planteó un nuevo reto a los ingenieros de software: diseñar e implementar sistemas que fueran seguros. Conforme más y más sistemas se conectaban a Internet, se emprendió una variedad de ataques externos para amenazar estos sistemas. Los problemas para generar sistemas confiables aumentaron considerablemente. Los ingenieros de sistemas debían tomar en cuenta las amenazas de atacantes maliciosos y técnicamente hábiles, además de los problemas resultantes de errores accidentales en el proceso de desarrollo.

Ahora es esencial diseñar sistemas para soportar ataques externos y recuperarse de ellos. Sin precauciones de seguridad, es casi inevitable que los atacantes comprometerán un sistema en red. Los atacantes pueden hacer mal uso del hardware del sistema, robar datos confidenciales o dificultar los servicios que ofrece el sistema. Por lo tanto, la ingeniería de seguridad de sistemas es un aspecto cada vez más importante del proceso de ingeniería de sistemas.

La ingeniería de seguridad se interesa por el desarrollo y la evolución de los sistemas que pueden hacer frente a ataques maliciosos, cuya intención es perjudicar el sistema o los datos. La ingeniería de seguridad de software es la parte más general del campo de seguridad computacional. Ésta se ha convertido en una prioridad para las compañías y los individuos, pues más y más transgresores tratan de aprovecharse de los sistemas en red con propósitos ilícitos. Los ingenieros de software deben conocer tanto las amenazas a la seguridad que enfrentan los sistemas, como las formas en las que es posible neutralizar tales amenazas.

La intención de este capítulo es presentar la ingeniería de seguridad a los ingenieros de software, enfocándose en los conflictos de diseño que afectan la seguridad de la aplicación. Este apartado no trata la seguridad de la computadora en su conjunto ni cubre temas como encriptación, control de acceso, mecanismos de autorización, virus y caballos troyanos, etcétera, sino que describe a detalle textos generales sobre seguridad computacional (Anderson, 2008; Bishop, 2005; Pfleeger y Pfleeger, 2007).

El capítulo se suma al análisis de la seguridad que se presenta en otras partes del libro. Este material debe leerse junto con:

- Sección 10.1, que explica cómo la seguridad y la confiabilidad están estrechamente relacionadas;
- Sección 10.4, que introduce la terminología de seguridad;
- Sección 12.1, que expone la noción general de especificación dirigida por riesgo;
- Sección 12.4, que analiza los conflictos generales de la especificación de requerimientos de seguridad;
- Sección 15.3, que describe algunos enfoques a las pruebas de seguridad.

Al considerar los conflictos de seguridad, debe contemplarse tanto el software de aplicación (el sistema de control, el sistema de información, etcétera) como la infraestructura sobre la que se construye el sistema (figura 14.1). La infraestructura para aplicaciones complejas incluye:

- una plataforma de sistema operativo, tal como Linux o Windows;
- otras aplicaciones genéricas que operan en dicho sistema, tales como navegadores Web y clientes de correo electrónico;
- un sistema de gestión de base de datos;

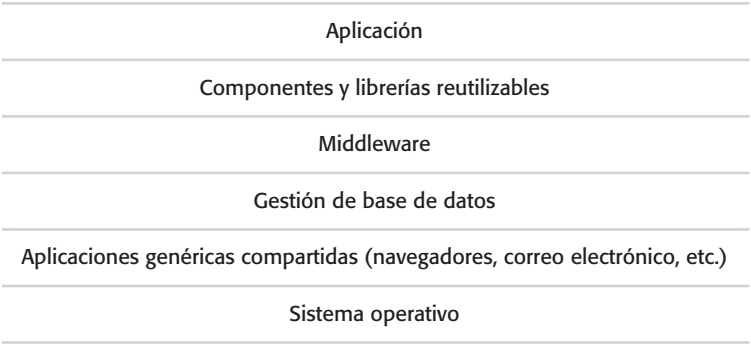


Figura 14.1 Capas de sistema donde puede comprometerse la seguridad

- middleware, que soporta computación distribuida y acceso a la base de datos;
- librerías de componentes reutilizables, que usa el software de aplicación.

La mayoría de los ataques externos se enfocan en infraestructuras de sistema debido a que los componentes de infraestructura (por ejemplo, navegadores Web) son bien conocidos y ampliamente disponibles. Los atacantes pueden probar dichos sistemas para localizar puntos débiles y buscar información compartida sobre las vulnerabilidades descubiertas. Puesto que muchas personas usan el mismo software, los ataques tienen amplia aplicabilidad. Las vulnerabilidades de infraestructura pueden conducir a que los atacantes obtengan sin autorización acceso a un sistema de aplicación y a los datos.

En la práctica, existe una importante diferencia entre seguridad de aplicación y seguridad de infraestructura:

1. La seguridad de aplicación es un problema de ingeniería de software; los ingenieros de software deben asegurarse de que el sistema se diseñó para soportar ataques.
2. La seguridad de infraestructura es un problema de gestión en el que los administradores del sistema configuran la infraestructura para resistir ataques. Los administradores de sistema deben configurar la infraestructura con la finalidad de usar de manera más efectiva cualquier característica de seguridad de infraestructura disponible. También deben corregir las vulnerabilidades de seguridad de infraestructura que salgan a la luz conforme se use el software.

La gestión de la seguridad del sistema no es una tarea sencilla, sino que debe incluir un rango de actividades como gestión de usuarios y permisos, implementación y mantenimiento del software del sistema, y monitorización, detección y recuperación de ataque.

1. La gestión de usuarios y permisos implica agregar y eliminar usuarios del sistema, asegurar que se coloquen mecanismos adecuados de autenticación de usuarios, y configurar los permisos en el sistema de forma que los usuarios sólo tengan acceso a los recursos que necesitan.
2. La implementación y el mantenimiento del software del sistema comprenden la instalación del software y middleware del sistema y configurar éste de manera adecuada, además de evitar las vulnerabilidades de seguridad. Asimismo, incluyen la

Ataques internos e ingeniería social

Los ataques internos son ataques a un sistema realizados por un individuo de confianza (un empleado de la organización) que abusa de la situación. Por ejemplo, una enfermera que trabaja en un hospital puede tener acceso a registros médicos confidenciales de pacientes a quienes no atiende. Los ataques internos son difíciles de contrarrestar porque las técnicas adicionales de seguridad que pudieran usarse afectarían a los usuarios confiables del sistema.

La ingeniería social es una forma de engañar a los usuarios acreditados para que muestren sus credenciales. En consecuencia, un atacante puede comportarse como empleado de la organización cuando ingresa al sistema.

http://www.SoftwareEngineering-9.com/Web/SecurityEng/insiders.html

actualización regular de este software con nuevas versiones o parches, que reparen los problemas de seguridad descubiertos.

3. La monitorización del ataque, detección y recuperación incluyen actividades que monitorizan el sistema en relación con acceso no autorizado, detectan e instauran estrategias para resistir ataques, así como actividades de respaldo de manera que, después de un ataque externo, pueda reanudarse la operación normal.

La gestión de la seguridad tiene una importancia vital, pero no suele considerarse como parte de la ingeniería de seguridad de aplicación. En vez de ello, este tipo de ingeniería se interesa por el diseño de un sistema que sea tan seguro como sea posible, dadas las restricciones presupuestales y de uso. Parte de este proceso consiste en "diseñar para administrar", es decir, diseñar sistemas para minimizar las probabilidades de errores en la administración de la seguridad que permitan ataques al sistema.

En sistemas de control crítico y sistemas embebidos, es práctica normal seleccionar una infraestructura adecuada para apoyar el sistema de aplicación. Por ejemplo, los desarrolladores de sistemas embebidos eligen por lo general un sistema operativo de tiempo real que ofrezca a la aplicación embebida las facilidades que necesita. Deben considerarse las vulnerabilidades conocidas y los requerimientos de seguridad, lo que significa que para la ingeniería de seguridad puede adoptarse un enfoque holístico. Los requerimientos de seguridad de aplicación se implementan mediante la infraestructura o la aplicación en sí.

Sin embargo, los sistemas de aplicación en una organización por lo general se implementan utilizando la infraestructura existente (sistema operativo, base de datos, etcétera). Por lo tanto, deben considerarse los riesgos de utilizar dicha infraestructura y las características de seguridad como parte del proceso de diseño del sistema.

14.1 Gestión del riesgo de seguridad

Para la ingeniería de seguridad efectiva son esenciales tanto la valoración como la gestión del riesgo de seguridad. Esta última se encarga de evaluar las posibles pérdidas que se derivan de ataques a los activos en el sistema, y de equilibrar dichas pérdidas frente

a los costos de los procedimientos de seguridad encaminados a reducir las pérdidas. Las compañías de tarjetas de crédito actúan continuamente de esa forma. Es relativamente sencillo introducir nueva tecnología para reducir la probabilidad de fraude con las tarjetas de crédito. Sin embargo, con frecuencia es más económico que las empresas compensen a los usuarios por sus pérdidas como consecuencia de los fraudes, que comprar e implementar tecnología que permita reducir estos últimos. Conforme los costos disminuyen y aumentan los ataques, este balance puede cambiar. Por ejemplo, las compañías de tarjetas de crédito ahora codifican información en un chip sobre la tarjeta y no sobre la banda magnética. Esto hace que sea mucho más difícil copiar la tarjeta.

La gestión del riesgo es un conflicto empresarial más que un conflicto técnico, así que los ingenieros de software no son quienes deben decidir qué controles incluir en un sistema. Es responsabilidad de los altos ejecutivos de la empresa decidir si aceptan el costo de la seguridad o se exponen a las consecuencias de una falta de procedimientos de seguridad. En cambio, el papel de los ingenieros de software es ofrecer guía y orientación técnica informada acerca de conflictos de seguridad. Por lo tanto, ellos son participantes esenciales dentro del proceso de gestión del riesgo.

Como se explicó en el capítulo 12, una entrada crítica al proceso de valoración y gestión del riesgo es la política de seguridad de la organización. Ésta se aplica a todos los sistemas y establece lo que debe permitirse y lo que no. La política de seguridad establece las condiciones que deben mantenerse siempre mediante un sistema de seguridad, y además ayuda a identificar los riesgos y las amenazas que pudieran surgir. En consecuencia, la política de seguridad define lo que se permite y lo que no se permite. En el proceso de ingeniería de seguridad se diseñan los mecanismos para aplicar esta política.

La valoración del riesgo se inicia antes de decidir la adquisición del sistema y debe continuar a lo largo del proceso de desarrollo de éste, y después de que se pone en uso (Alberts y Dorofee, 2002). En el capítulo 12 se introdujo la idea de que esta valoración del riesgo es un proceso por etapas:

1. *Valoración preliminar del riesgo* En esta etapa aún no se toman decisiones sobre los requerimientos detallados del sistema, del diseño o de la tecnología de implementación. La meta de este proceso de valoración es decidir si puede lograrse un nivel adecuado de seguridad a un costo razonable. Si éste es el caso, entonces es posible establecer requerimientos de seguridad específicos para el sistema. No se tiene información de las vulnerabilidades potenciales en el sistema o los controles que se incluyen en componentes reutilizados del sistema o middleware.

2. *Valoración del riesgo del ciclo de vida* Esta valoración del riesgo tiene lugar durante el ciclo de vida de desarrollo del sistema, y es un informe de las decisiones técnicas de diseño e implementación del sistema. Los resultados de la valoración pueden conducir a cambios en los requerimientos de seguridad y a agregar otros. Se identifican las vulnerabilidades conocidas y potenciales. Este conocimiento se emplea para la toma de decisiones informada sobre la funcionalidad del sistema y de cómo ésta se implementará, probará y desplegará.

3. *Valoración del riesgo operativo* Después de que un sistema se implementa y pone en operación, se debe realizar una valoración del riesgo para tomar en cuenta cómo se usa el sistema y las propuestas para los nuevos y cambiantes requerimientos. Las suposiciones referentes a los requerimientos operativos que se hicieron cuando

el sistema se especificaba pueden ser incorrectas. Los cambios en la organización significan que el sistema podría utilizarse en diferentes formas a las planeadas originalmente. Por consiguiente, la valoración del riesgo operativo conduce a nuevos requerimientos de seguridad que deben implementarse a medida que evoluciona el sistema.

La valoración preliminar del riesgo se enfoca en la derivación de requerimientos de seguridad. En el capítulo 12 se mostró cómo puede derivarse un conjunto inicial de requerimientos de seguridad a partir de una valoración preliminar del riesgo. Esta sección se concentra en la valoración del riesgo del ciclo de vida y operacional, para ilustrar cómo la especificación y el diseño de un sistema reciben influencia de la tecnología y de las formas como se utiliza el sistema.

Para realizar una valoración del riesgo, es necesario identificar las posibles amenazas a un sistema. Una forma de hacerlo es desarrollar un conjunto de "casos de mal uso" (Alexander, 2003; Sindre y Opdahl, 2005). Ya se examinó cómo utilizar los casos de uso, es decir, las interacciones típicas con un sistema, para derivar los requerimientos de este último. Los casos de mal uso son escenarios que representan interacciones maliciosas con un sistema. Al desarrollar casos de mal uso es posible analizar e identificar posibles amenazas y, por lo tanto, también determinar los requerimientos de seguridad del sistema. Pueden usarse junto con los casos de uso para establecer los requerimientos del sistema.

Pfleeger y Pfleeger (2007) clasifican las amenazas en cuatro categorías, las cuales constituyen un punto de partida para identificar posibles casos de mal uso. Esas categorías son las siguientes:

1. Amenazas de intercepción, que permiten a un atacante conseguir acceso a un activo. De este modo, un posible caso de mal uso para el MHC-PMS puede ser una situación en que un atacante logra tener acceso a los registros de un paciente que es una celebridad.
2. Amenazas de interrupción, las cuales permiten a un atacante tomar parte de la indisposición del sistema. Por ejemplo, un posible caso de mal uso es un ataque de negación de servicio al servidor de la base de datos del sistema.
3. Amenazas de modificación, que permiten a un atacante sabotear un activo del sistema. En el MHC-PMS, esto podría representarse con un caso de mal uso en que un atacante cambia la información en el registro de un paciente.
4. Amenazas de fabricación, las cuales permiten a un atacante insertar información falsa en un sistema. Posiblemente ésta no es una amenaza verosímil en el MHC-PMS, pero sin duda sería una amenaza en un sistema bancario, donde podrían agregarse transacciones falsas al sistema para transferir dinero a la cuenta bancaria del transgresor.

Los casos de mal uso no sólo son útiles en la valoración preliminar del riesgo, sino también pueden usarse para analizar la seguridad en términos del riesgo del ciclo de vida y el riesgo operativo. Proporcionan una base útil para efectuar ataques hipotéticos al sistema y valorar las implicaciones de seguridad de las decisiones de diseño realizadas.

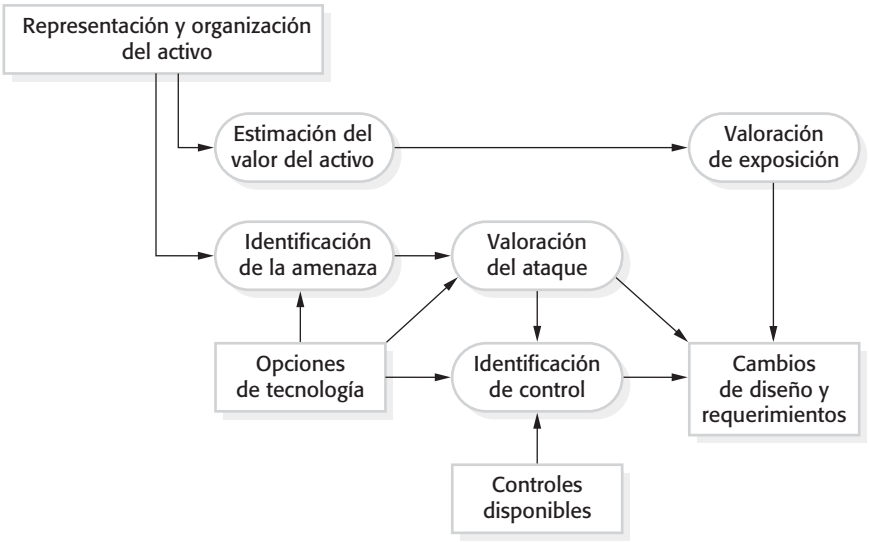


Figura 14.2 Análisis del riesgo del ciclo de vida

14.1.1 Valoración del riesgo del ciclo de vida

Con base en las políticas de seguridad de la organización, la valoración preliminar del riesgo debe identificar los requerimientos de seguridad más importantes para un sistema. Ello refleja cómo se debe implementar la política de seguridad en dicha aplicación, permite identificar los activos a proteger y ayuda a decidir qué enfoque podría usarse para ofrecer tal protección. Mantener la seguridad implica poner atención a detalle. Sin embargo, es imposible que los requerimientos iniciales de seguridad tomen en cuenta todos los detalles que afectan la seguridad.

La valoración del riesgo del ciclo de vida identifica los detalles de diseño e implementación que afectan la seguridad. Ésta es la diferencia importante entre valoración del riesgo del ciclo de vida y valoración preliminar del riesgo. La valoración del riesgo del ciclo de vida afecta la interpretación de los requerimientos de seguridad existentes, genera nuevos requerimientos e influye en el diseño global del sistema.

Cuando se valoran riesgos en esta etapa, es indispensable tener información mucho más detallada sobre qué se necesita proteger, y también conocer algo referente a las vulnerabilidades del sistema. Algunas de estas vulnerabilidades serán inherentes a las decisiones que se tomen en materia de diseño. Por ejemplo, una vulnerabilidad en todos los sistemas basados en contraseñas es que algún usuario autorizado revele su contraseña a otro usuario no autorizado. O bien, si una organización tiene una política para desarrollar software en C, sabrá que la aplicación puede tener vulnerabilidades, pues el lenguaje no incluye comprobación de límite de arreglo.

La valoración del riesgo de seguridad debe ser parte de las actividades del ciclo de vida de la ingeniería, desde la ingeniería de requerimientos hasta la implementación del sistema. El proceso que se sigue es similar al proceso de valoración preliminar del riesgo con la adición de actividades concernientes a la identificación y valoración de la vulnerabilidad del diseño. El resultado de la valoración del riesgo es un conjunto de decisiones de ingeniería que afectan el diseño o la implementación del sistema, o limitan la forma en que se usa.

En la figura 14.2 se muestra un modelo del proceso de análisis del riesgo del ciclo de vida, basado en el proceso de análisis preliminar del riesgo que se describe en la figura 12.9. La diferencia más importante entre dichos procesos es que ahora se tiene información de la representación y distribución de información, así como de la organización de la base de datos para los activos de alto nivel que deben protegerse. También se está al tanto de importantes decisiones de diseño, tales como el software a reutilizar, los controles y la protección de infraestructura, etcétera. Con base en esta información, el análisis permite identificar cambios necesarios en los requerimientos de seguridad y en el diseño del sistema para brindar protección adicional a los activos importantes del sistema.

Dos ejemplos ilustran cómo los requerimientos de protección reciben influencia de las decisiones concernientes a la representación y distribución de información:

1. Es posible tomar una decisión de diseño para separar la información personal de un paciente y la información de los tratamientos recibidos mediante una clave que vincule ambos registros. La información del tratamiento es mucho menos sensible que la información personal del paciente, de manera que quizá no necesite una protección tan profunda. Si la clave está protegida, entonces un atacante sólo podrá tener acceso a información de rutina, sin poder vincular esto con los datos individuales de un paciente.
2. Suponga que, al comienzo de la sesión, se toma una decisión de diseño para copiar registros de pacientes a un sistema cliente local. Esto permite que el trabajo continúe si el servidor no está disponible. Ello posibilita que un empleado de atención a la salud tenga acceso a los registros de los pacientes desde una laptop, incluso si no está disponible una conexión a red. Sin embargo, ahora tiene dos conjuntos de registros para proteger y las copias del cliente están sujetas a riesgos adicionales, como el robo de la laptop. Por lo tanto, habrá que pensar en qué controles ayudarían a reducir el riesgo. Por ejemplo, es posible que deban encriptarse los registros del cliente en la laptop.

Para ilustrar cómo las decisiones sobre las tecnologías de desarrollo influyen en la seguridad, suponga que el proveedor de atención a la salud decide construir un MHC-PMS mediante un sistema de información comercial para mantener registros de pacientes. Este sistema tiene que configurarse para cada tipo de clínica en la que se utilice. Esta decisión se toma porque parece ofrecer la funcionalidad más amplia en términos del costo de desarrollo más bajo y el tiempo de despliegue más rápido.

Cuando se elabora una aplicación con la reutilización de un sistema existente, deben aceptarse las decisiones de diseño tomadas por los desarrolladores de dicho sistema. Suponga que algunas de tales decisiones de diseño son las siguientes:

1. Los usuarios del sistema se autentican a través de una combinación de nombre/contraseña de acceso. No existe otro método de autenticación.
2. La arquitectura del sistema es cliente-servidor, y los clientes acceden a los datos mediante un navegador Web estándar en una PC cliente.
3. La información se presenta a los usuarios como un formato Web editable. Ellos pueden cambiar la información en el lugar y subir al servidor la información revisada.

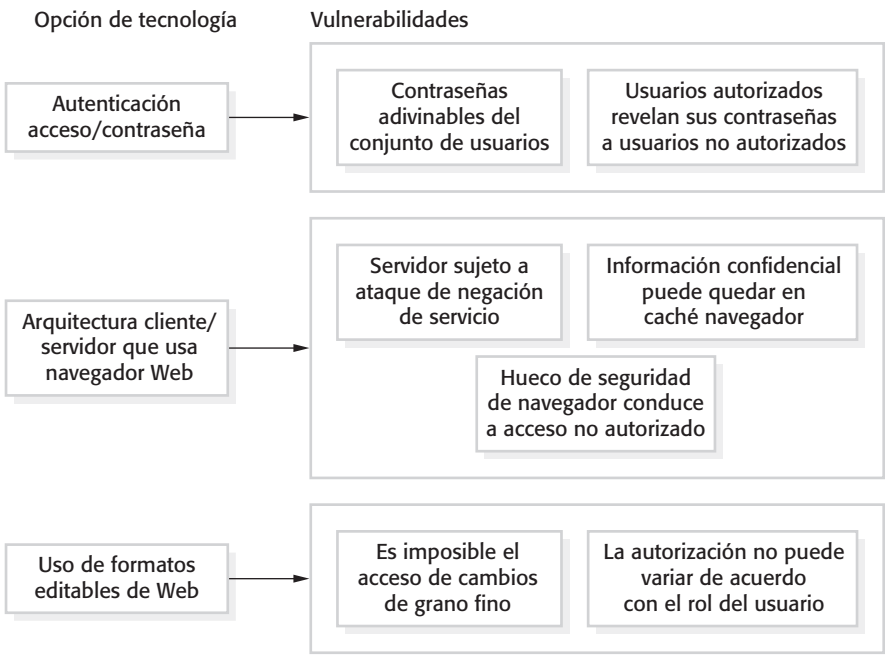


Figura 14.3 Vulnerabilidades asociadas con las opciones de tecnología

Para un sistema genérico, estas decisiones de diseño son perfectamente aceptables, pero un análisis del riesgo del ciclo de vida revela que tienen vulnerabilidades asociadas. En la figura 14.3 se presentan ejemplos de posibles vulnerabilidades.

Una vez identificadas las vulnerabilidades, debe tomarse entonces una decisión sobre qué pasos habrá que dar para reducir los riesgos asociados. Con frecuencia, esto implicará la toma de decisiones respecto a los requerimientos de seguridad de sistema adicional o el proceso operacional de usar el sistema. Aun cuando aquí no hay espacio para analizar todos los requerimientos que pueden proponerse al examinar las vulnerabilidades inherentes, algunos ejemplos de requerimientos son los siguientes:

1. Un programa de comprobación de contraseña debe estar disponible y funcionar continuamente. Las contraseñas de los usuarios que aparezcan en el diccionario del sistema deben identificarse; los usuarios con contraseñas vulnerables deben reportarse a los administradores del sistema.
2. El acceso al sistema sólo debe permitirse a computadoras cliente que aprobaron y registraron los administradores del sistema.
3. Todas las computadoras cliente deben tener un solo navegador Web instalado, y éste debe recibir la aprobación de los administradores del sistema.

Conforme se use un sistema comercial, no será posible incluir un verificador de contraseña en el sistema de aplicación en sí, de manera que se debe usar un sistema separado. Los verificadores de contraseñas analizan la fortaleza de las contraseñas de los usuarios cuando se configuran, y notifican a los usuarios si eligieron contraseñas vulnerables. Por consiguiente, las contraseñas vulnerables pueden identificarse de manera relativamente

rápida después de que se configuraron, y entonces será posible emprender una acción para garantizar que los usuarios cambien sus contraseñas.

El segundo requerimiento y el tercero significan que todos los usuarios siempre accederán al sistema a través del mismo navegador. Se puede decidir cuál es el navegador más seguro cuando el sistema se despliega e instala en todas las computadoras cliente. Las actualizaciones de seguridad se simplifican porque no hay necesidad de actualizar diferentes navegadores cuando se descubren y corrigen vulnerabilidades de seguridad.

14.1.2 Valoración del riesgo operativo

La valoración del riesgo de seguridad podría continuar a lo largo de la vida del sistema con la finalidad de identificar riesgos emergentes y cambios al sistema que puedan requerirse para lidiar con tales riesgos. Este proceso se llama valoración del riesgo operativo. Es probable que surjan nuevos riesgos debido a requerimientos cambiantes del sistema, cambios en la infraestructura del sistema, o cambios en el entorno donde se utiliza el sistema.

El proceso de valoración del riesgo operativo es similar al proceso de valoración del riesgo del ciclo de vida, pero implica agregar más información acerca del entorno en el que se emplea el sistema. El entorno es importante porque sus características pueden conducir al sistema a nuevos riesgos. Por ejemplo, suponga que un sistema se usará en un entorno donde los usuarios son interrumpidos con frecuencia. Un riesgo es que la interrupción signifique que el usuario deje de prestar atención a su computadora. En tal caso, es posible que una persona no autorizada consiga acceso a la información en el sistema. Entonces esto podría generar un requerimiento para que un protector de pantalla solicite una contraseña para operar después de un breve periodo de inactividad.

14.2 Diseño para la seguridad

En general, es cierto que resulta muy difícil adicionar seguridad a un sistema después de que se implementó. Por lo tanto, se deben tomar en cuenta los riesgos de seguridad durante el proceso de diseño del sistema. Esta sección se enfocará principalmente en los conflictos del diseño de un sistema, porque, por lo regular, en los libros de seguridad computacional no se concede a este tema la atención que merece. Los conflictos y errores de implementación también tienen una gran repercusión sobre la seguridad, pero éstos dependen normalmente de la tecnología específica utilizada. Se recomienda leer el libro de Viega y McGraw (2002), como una buena introducción a la programación para la seguridad.

Aquí el enfoque será sobre algunos conflictos generales, independientes de aplicación, relevantes para el diseño de sistemas seguros:

1. Diseño arquitectónico: ¿cómo afectan las decisiones de diseño arquitectónico la seguridad de un sistema?
2. Buena práctica: ¿qué se acepta como buena práctica al diseñar sistemas seguros?
3. Diseño para implementación: ¿qué soporte debe diseñarse en los sistemas para evitar la introducción de vulnerabilidades cuando un sistema se despliega para su uso?

Ataques de negación de servicio

Los ataques de negación de servicio tratan de derrumbar un sistema en red al bombardearlo con un enorme número de peticiones de servicio. Esto supone una carga adicional sobre el sistema, para la cual no está diseñado, y excluye las peticiones legítimas de servicio. En consecuencia, el sistema puede volverse inaccesible porque se cae ante la enorme carga, o bien, los administradores del sistema deben sacarlo de línea para detener el flujo de peticiones.

http://www.SoftwareEngineering-9.com/Web/Security/DoS.html

Desde luego, éstos no son los únicos conflictos de diseño importantes para la seguridad. Toda aplicación es diferente y el diseño de seguridad también debe tomar en cuenta el propósito, el carácter crítico y el entorno operacional de la aplicación. Por ejemplo, si se diseña un sistema militar, es necesario adoptar un modelo de clasificación de seguridad (secreto, ultrasecreto, etcétera). Si se diseña un sistema que mantiene información personal, debe considerarse la legislación sobre protección de datos que impone restricciones respecto a cómo se administra la información.

Existe una relación estrecha entre confiabilidad y seguridad. El uso de redundancia y diversidad, que es fundamental para lograr confiabilidad, puede significar que un sistema resista y se recupere de ataques dirigidos a características o diseños específicos de implementación. Mecanismos de apoyo a alto nivel de disponibilidad pueden ayudar al sistema a recuperarse de los llamados ataques de negación de servicio, en los que la meta del atacante es hacer caer al sistema y detener su funcionamiento correcto.

Diseñar un sistema para que sea seguro implica inevitablemente un compromiso. Desde luego, es posible desarrollar múltiples medidas de seguridad en un sistema, las cuales reducirán las probabilidades de que un ataque logre su cometido. Sin embargo, las medidas de seguridad requieren a menudo una gran labor adicional de computación y, además, afectan el rendimiento global de un sistema. Por ejemplo, se pueden reducir las posibilidades de que se muestre información confidencial al encriptar dicha información. No obstante, esto significa que los usuarios de la información tienen que esperar a que ésta se desencripte, lo cual hará más lento su trabajo.

También existen tensiones entre seguridad y usabilidad. En ocasiones, las medidas de seguridad requieren que el usuario recuerde y proporcione información adicional (por ejemplo, contraseñas múltiples). Sin embargo, algunas veces el usuario olvida dicha información, de manera que la seguridad adicional significa que no puede usar el sistema. Por consiguiente, los diseñadores deben encontrar un equilibrio entre seguridad, rendimiento y usabilidad. Esto dependerá del tipo de sistema y de dónde se utilizará. Por ejemplo, en un sistema militar, los usuarios están familiarizados con los sistemas de alta seguridad, y también están deseosos de aceptar y seguir procesos que requieren comprobaciones periódicas. Sin embargo, en un sistema de comercio de acciones, las interrupciones de la operación para comprobaciones de seguridad serían completamente inaceptables.

14.2.1 Diseño arquitectónico

Como se estudió en el capítulo 11, la elección de arquitectura de software puede tener profundos efectos sobre las propiedades emergentes de un sistema. Si se usa una arquitectura inadecuada, tal vez sea muy difícil mantener en el sistema la confidencialidad e

integridad de la información, o garantizar un nivel requerido de disponibilidad del sistema.

Al diseñar una arquitectura de sistema que conserve la seguridad, se deben considerar dos temas fundamentales:

1. Protección: ¿cómo debe organizarse el sistema de manera que puedan protegerse los activos críticos contra ataques externos?
2. Distribución: ¿cómo deben distribuirse los activos del sistema de manera que sean mínimos los efectos de un ataque?

Estos temas son potencialmente conflictivos. Si se colocan todos los activos en un lugar, entonces es posible construir capas de protección en torno a ellos. Puesto que se debe construir un solo sistema de protección, es posible costear un sistema fuerte con muchas capas de protección. Si a pesar de ello, dicha protección fracasa, entonces todos los activos estarán comprometidos. Añadir más capas de protección también afecta la usabilidad de un sistema, lo que puede significar que sea más difícil satisfacer los requerimientos de usabilidad y rendimiento del sistema.

Por otra parte, si se distribuyen los activos, será más costoso protegerlos, ya que deben implementarse sistemas de protección para cada copia. Normalmente, no es posible costear tantas capas de protección. Hay más posibilidades de que la protección se rompa. Pero, si esto sucede, no habrá una pérdida total. Es posible duplicar y distribuir activos de información de manera que, si una copia se corrompe o es inaccesible, puede usarse entonces la otra copia. No obstante, si la información es confidencial, el conservar copias adicionales aumenta el riesgo de que un intruso obtenga el acceso a esta información.

Para el sistema de registros de pacientes, es adecuado usar una arquitectura de base de datos centralizada. Para brindar protección, se emplea una arquitectura en capas con los activos críticos protegidos en el nivel más bajo del sistema, con varias capas de protección en torno a ellos. La figura 14.4 ilustra esto para el sistema de registros de pacientes en el que los activos críticos a proteger son los registros individuales de cada paciente.

Si un atacante quiere tener acceso a los registros de pacientes y modificarlos, debe penetrar tres capas del sistema:

1. *Protección a nivel de plataforma* El nivel superior controla el acceso a la plataforma donde opera el sistema de registro de pacientes. Por lo general, esto implica el ingreso de un usuario a una computadora particular. La plataforma también incluirá comúnmente soporte para mantener la integridad de los archivos en el sistema, respaldos, etcétera.
2. *Protección a nivel de aplicación* El siguiente nivel de protección se construye en la aplicación en sí. Implica el acceso de un usuario a la aplicación, su autenticación y la obtención de permiso para realizar acciones como ver o modificar datos. Puede estar disponible soporte de gestión de integridad específico de la aplicación.
3. *Protección a nivel de registro* Este nivel se demanda cuando se requiere acceso a registros específicos, e incluye comprobar que un usuario está autorizado para realizar las operaciones solicitadas sobre tal registro. La protección a este nivel también

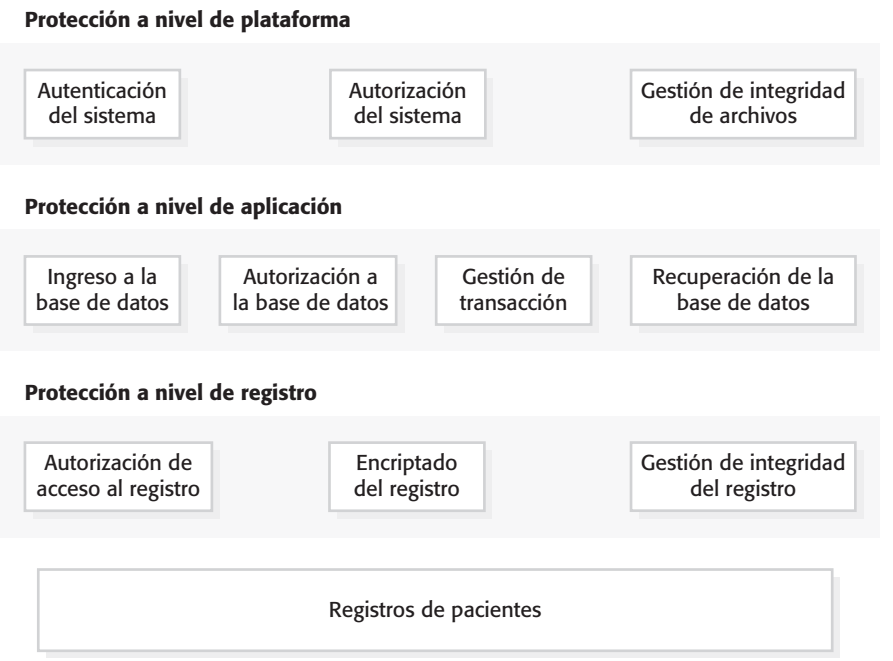


Figura 14.4 Arquitectura de protección en capas

podría implicar encriptado para garantizar que sea imposible consultar los registros utilizando un navegador de archivo. La comprobación de integridad mediante sumas de verificación criptográficas, por ejemplo, permite detectar cambios que se hayan realizado fuera de los mecanismos normales de actualización de registro.

El número de capas de protección que se necesitan en cualquier aplicación particular depende del carácter crítico de los datos. No todas las aplicaciones requieren protección a nivel registro y, por lo tanto, se usa comúnmente control de acceso de grano más grueso. Para lograr seguridad, no debe permitirse el uso de las mismas credenciales de usuario en cada nivel. Lo ideal sería que si se tiene un sistema basado en contraseñas, entonces la contraseña de la aplicación debe ser diferente tanto de la contraseña del sistema como de la contraseña a nivel de registro. Sin embargo, es difícil que los usuarios recuerden múltiples contraseñas y encuentran molestas las reiteradas peticiones de autenticación. Por ende, con frecuencia se tendrá que comprometer la seguridad en favor de la usabilidad del sistema.

Si la protección de datos es un requerimiento crítico, debe usarse en tal caso una arquitectura cliente-servidor, con los mecanismos de protección construidos en el servidor. No obstante, si se compromete la protección, es probable que las pérdidas asociadas con un ataque sean altas, al igual que los costos de recuperación (por ejemplo, tienen que emitirse nuevamente todas las credenciales de usuario). El sistema es vulnerable a los ataques de negación de servicio, que sobrecargan el servidor y hacen imposible que alguien acceda a la base de datos del sistema.

Si se considera que los ataques de negación de servicio son un riesgo mayor, se puede optar por una arquitectura de objeto distribuida para la aplicación. En esta situación, que se ilustra en la figura 14.5, los activos del sistema se distribuyen a través de algunas plataformas diferentes, con mecanismos de protección separados para cada una de ellas. Un

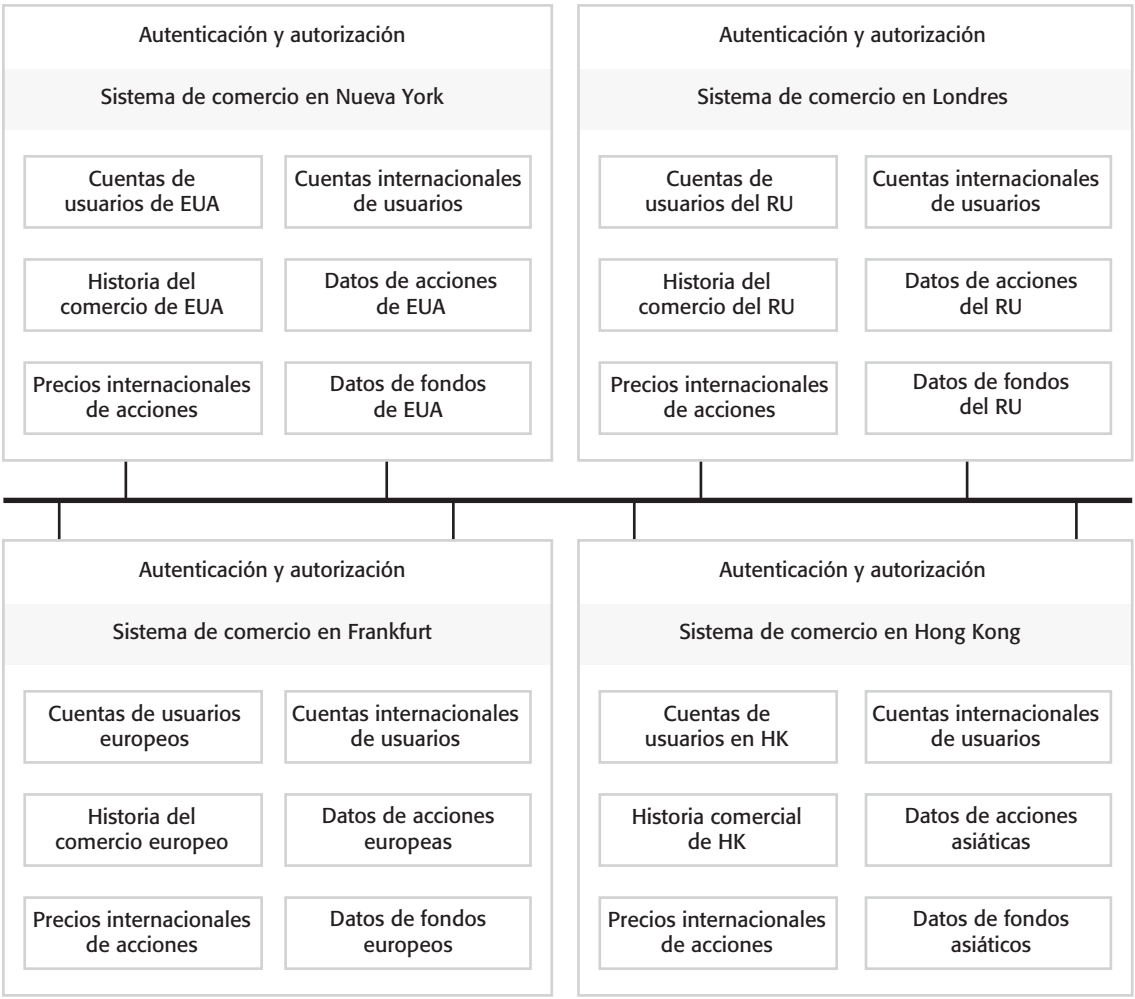


Figura 14.5 Activos distribuidos en un sistema de comercio de acciones

ataque en un nodo puede significar que algunos activos no estén disponibles, pero todavía es posible ofrecer algunos servicios del sistema. Es posible duplicar datos a través de los nodos en el sistema, de forma que se simplifica la recuperación de ataques.

La figura 14.5 muestra la arquitectura de un sistema bancario para comerciar acciones y fondos en los mercados de Nueva York, Londres, Frankfurt y Hong Kong. El sistema es distribuido, así que los datos de cada mercado se mantienen por separado. Los activos requeridos para soportar la actividad crítica de comercio de acciones (cuentas de usuarios y precios) se duplican y están disponibles en todos los nodos. Si un nodo del sistema sufre un ataque y deja de estar disponible, la actividad crítica del comercio de acciones puede transferirse a otro país y, por lo tanto, seguir disponible para los usuarios.

Ya se habló del problema de encontrar un equilibrio entre seguridad y rendimiento del sistema. Un problema de diseño seguro del sistema es que en muchos casos el estilo arquitectónico, que es más adecuado para satisfacer los requerimientos de seguridad, tal vez no sea el mejor para satisfacer los requerimientos de rendimiento. Por ejemplo, suponga que una aplicación tiene un requerimiento absoluto para mantener la confidencialidad de una

gran base de datos, y otro requerimiento para acceso muy rápido a dichos datos. Un alto nivel de protección sugiere la necesidad de capas de protección, lo que significa que debe haber comunicaciones entre las capas del sistema. Esto tiene una inevitable sobrecarga de rendimiento, lo que hace más lento el acceso a los datos. Si se usa una arquitectura alternativa, entonces quizá sea más difícil y costoso implementar protección y garantizar confidencialidad. Ante tal situación, es necesario discutir los conflictos inherentes con el cliente del sistema y acordar cómo se resolverán.

14.2.2 Lineamientos de diseño

No existen reglas rígidas y rápidas sobre cómo lograr seguridad del sistema. Diferentes tipos de sistemas requieren distintas medidas técnicas para lograr un nivel de seguridad que sea aceptable para el propietario del sistema. Las actitudes y los requerimientos de diversos grupos de usuarios afectan profundamente lo que es aceptable y lo que no lo es. Por ejemplo, en un banco, es probable que los usuarios acepten un alto nivel de seguridad, y más procedimientos de seguridad anti-intrusos en comparación con los usuarios en una universidad.

Sin embargo, existen lineamientos generales con amplia aplicabilidad al diseñar soluciones de seguridad de un sistema, los cuales encapsulan buenas prácticas de diseño para la ingeniería de sistemas seguros. Los lineamientos generales de diseño para seguridad, como los que se analizan más adelante, tienen dos usos principales:

1. Ayudan a crear conciencia acerca de los temas de seguridad en un equipo de ingeniería de software. Los ingenieros de software se enfocan con frecuencia en la meta a corto plazo de hacer el software operativo y entregarlo a los clientes. Para ellos es fácil pasar por alto los temas de seguridad. El conocimiento de dichos lineamientos puede significar que los temas de seguridad se consideren cuando se toman las decisiones de diseño de software.
2. Pueden usarse como una lista de verificación que será útil en el proceso de validación del sistema. A partir de los lineamientos de nivel superior estudiados aquí, es posible derivar cuestiones más específicas que examinen cómo se sometió a ingeniería la seguridad de un sistema.

Los 10 lineamientos de diseño, que se resumen en la figura 14.6, resultaron de varias fuentes (Schneier, 2000; Viega y McGraw, 2002; Wheeler, 2003). Aquí el enfoque es sobre los lineamientos que son aplicables en particular a los procesos de especificación y diseño del software. Principios más generales, como "Asegura el vínculo más débil del sistema", "Hazlo simple" y "Evita seguridad mediante oscuridad" también son importantes, pero menos relevantes directamente para la toma de decisiones de ingeniería.

Lineamiento 1: Decisiones de seguridad con base en una política explícita de seguridad

Una política de seguridad es un enunciado de alto nivel que establece las condiciones fundamentales de seguridad para una organización. Define el "qué" de la seguridad en lugar del "cómo", de manera que la política no debe definir los mecanismos a usar para

Lineamientos de seguridad
1. Decisiones de seguridad con base en una política explícita de seguridad.
2. Evite un solo punto de falla.
3. Seguridad en caso de falla.
4. Equilibrio entre seguridad y usabilidad.
5. Registre las acciones del usuario.
6. Use redundancia y diversidad para reducir el riesgo.
7. Valide todas las entradas.
8. Compartimente sus activos.
9. Diseñe para implementar.
10. Diseñe para facilitar la recuperabilidad.

Figura 14.6 Lineamientos de diseño para la ingeniería de sistemas seguros

ofrecer seguridad y reforzarla. En un inicio, todos los aspectos políticos de seguridad deben reflejarse en los requerimientos del sistema. En la práctica, en especial si se usa un proceso rápido de desarrollo de aplicación, es improbable que esto suceda. Por lo tanto, los diseñadores deben consultar la política de seguridad, ya que ésta constituye un marco para tomar y evaluar las decisiones de diseño.

Por ejemplo, suponga que se diseña un sistema de control de acceso para el MHC-PMS. La política de seguridad del hospital puede establecer que sólo el personal médico acreditado está autorizado para modificar los registros electrónicos de los pacientes. De ahí que el sistema deba incluir mecanismos que comprueben la acreditación de cualquiera que intente modificar el sistema, y que rechacen modificaciones de personas no acreditadas.

El problema que se enfrenta es que muchas organizaciones no tienen una política explícita de seguridad de sistemas. Con el tiempo, pueden hacerse cambios a los sistemas en respuesta a problemas identificados, aunque sin un documento de política global que guíe la evolución de un sistema. Ante tales situaciones, se requiere trabajar y documentar la política a partir de ejemplos, y confirmarla con los administradores de la compañía.

Lineamiento 2: Evite un solo punto de falla

En cualquier sistema crítico, una buena práctica de diseño es tratar de evitar un solo punto de falla. Esto significa que una sola falla en parte del sistema no debe dar por resultado una falla global del sistema. En términos de seguridad, significa que no debe apoyarse en un solo mecanismo para garantizar la seguridad; en vez de ello, se deben emplear muchas técnicas diferentes. En ocasiones esto se conoce como “defensa en profundidad”.

Por ejemplo, si se solicita una contraseña para autenticar a los usuarios de un sistema, también puede incluirse un mecanismo de autenticación pregunta/respuesta donde los usuarios deban registrar previamente preguntas y respuestas en el sistema. Después de la autenticación de contraseña, deben responder correctamente las preguntas antes de que

se les permita el acceso. Para proteger la integridad de los datos en un sistema, hay que mantener una bitácora ejecutable de los cambios realizados a los datos (véase el lineamiento 5). En caso de una falla, podrá reproducirse la bitácora para recrear el conjunto de datos. También es conveniente hacer una copia de todos los datos que se modifiquen antes de realizar algún cambio.

Lineamiento 3: Seguridad en caso de falla

Las fallas son inevitables en todos los sistemas y, en la misma forma en que los sistemas críticos de protección siempre deben ser a prueba de fallas, los sistemas críticos para la seguridad siempre deben ser "seguros en caso de falla". Cuando falle el sistema, no se deben usar procedimientos de estado de emergencia que sean menos seguros que el sistema en sí. La falla del sistema tampoco debe significar que un atacante acceda a datos a los que normalmente no tendría acceso.

Por ejemplo, en el sistema de información de pacientes se sugiere un requerimiento de que los datos de los pacientes deben descargarse a un sistema cliente al comienzo de una sesión clínica. Esto acelera el acceso y significa que el acceso es posible si el servidor no está disponible. Por lo general, el servidor borra estos datos al final de la sesión clínica. Sin embargo, si el servidor falla, entonces existe la posibilidad de que la información se conserve en el cliente. Un método de seguridad en caso de falla en dichas circunstancias consiste en encriptar todos los datos almacenados de los pacientes en el cliente. Esto significa que un usuario no autorizado no podrá leer los datos.

Lineamiento 4: Equilibrio entre seguridad y usabilidad

Las demandas de seguridad y usabilidad a menudo son contradictorias. Para hacer seguro un sistema, debe introducir comprobaciones de que los usuarios están autorizados para utilizar el sistema y que actúan en concordancia con políticas seguras. Inevitablemente, todo ello supone hacer peticiones a los usuarios: deben recordar nombres y contraseñas de acceso, usar solamente el sistema de ciertas computadoras, etcétera. Esto significa que los usuarios tardan más tiempo para iniciar el sistema y usarlo de manera efectiva. Conforme se agregan características de seguridad a un sistema, es inevitable que éste se vuelva menos usable. Se recomienda la lectura del libro de Cranor y Garfinkel (2005), que examina un amplio rango de conflictos en el área general de seguridad y usabilidad.

Se llega a un punto en que es contraproducente seguir agregando nuevas características de seguridad a costa de la usabilidad. Por ejemplo, si se requiere que los usuarios ingresen múltiples contraseñas o que cambien sus contraseñas a intervalos frecuentes por cadenas de caracteres imposibles de recordar, simplemente anotarán en algún lugar dichas contraseñas. Un atacante (en especial uno interno) podría encontrar las contraseñas que se anotaron y así conseguir acceso al sistema.

Lineamiento 5: Registre las acciones del usuario

Si es prácticamente posible hacerlo, debe mantener siempre una bitácora de las acciones del usuario. Esta bitácora, al menos, debe registrar quién hizo qué, los activos que utilizó, y la hora y fecha de acción. Como se explicó en el lineamiento 2, si mantiene esto como lista de comandos ejecutables, tendrá la opción de reproducir la bitácora para recuperarse de las fallas. Desde luego, también necesita herramientas que le permitan analizar la bitácora y detectar acciones potencialmente anómalas. Dichas herramientas pueden esca-

near la bitácora y descubrir las acciones anómalas, y además ayudar a detectar ataques y rastrear cómo el atacante consiguió acceso al sistema.

Además de ayudar a recuperarse de la falla, una bitácora de acciones del usuario es útil porque actúa como un disuasivo a los ataques internos. Si las personas saben que sus acciones se registran, entonces es menos probable que intenten actuar de formas no autorizadas. Esto es más efectivo para ataques casuales, como una enfermera que busca registros de pacientes, o para detectar ataques donde se robaron credenciales de usuario legítimas mediante ingeniería social. Desde luego, esto no es infalible, ya que los internos técnicamente habilidosos también pueden tener acceso a la bitácora y modificarla.

Lineamiento 6: Use redundancia y diversidad para reducir el riesgo

Redundancia significa conservar más de una versión del software o de los datos en un sistema. Diversidad, cuando se aplica al software, significa que las diferentes versiones no deben apoyarse en la misma plataforma o implementarse usando las mismas tecnologías. Por lo tanto, una vulnerabilidad de plataforma o tecnología no afectará a todas las versiones y, en consecuencia, no conducirá a una falla común. En el capítulo 13 se explicó cómo redundancia y diversidad son mecanismos fundamentales que se usan en la ingeniería de confiabilidad.

Ya se estudiaron ejemplos de redundancia: mantener información de pacientes tanto en el servidor como en el cliente, primero en el sistema de atención a la salud mental, y luego en el sistema distribuido de comercio de acciones que se muestra en la figura 14.5. En el sistema de registro de pacientes, podría usar varios sistemas operativos en el cliente y el servidor (por ejemplo, Linux en el servidor, Windows en el cliente). Esto garantiza que un ataque basado en una vulnerabilidad de sistema operativo no afectará tanto al servidor como al cliente. Desde luego, habrá que negociar tales beneficios contra el creciente costo administrativo de mantener diferentes sistemas operativos en una organización.

Lineamiento 7: Valide todas las entradas

Un ataque común a un sistema implica proporcionar al sistema entradas inesperadas que hacen que se comporte en forma no anticipada. Esto simplemente puede causar la caída de un sistema, lo que deriva en pérdida de servicio, o bien, las entradas podrían constituir un código malicioso que se ejecute en el sistema. Las vulnerabilidades de desbordamiento de buffer, que primero se demostraron en el gusano de Internet (Spafford, 1989) y que utilizan comúnmente los atacantes (Berghel, 2001), pueden activarse mediante cadenas largas de entrada. Otro ataque bastante común es el llamado “envenenamiento SQL”, en el que un usuario malicioso ingresa un fragmento SQL que interpreta un servidor.

Como se explicó en el capítulo 13, es posible evitar muchos de estos problemas al diseñar la validación de entrada en su sistema. En esencia, nunca debe aceptar cualquier entrada sin aplicar algunas verificaciones. Como parte de los requerimientos, tiene que definir las comprobaciones que se apliquen. Debe usar el conocimiento de la entrada para definir dichas comprobaciones. Por ejemplo, si se ingresa un sobrenombre, puede comprobar que no existen espacios embebidos y que el único signo de puntuación utilizado es el guión. También puede verificar el número de caracteres de entrada y rechazar las entradas muy largas. Por ejemplo, nadie tiene un nombre familiar con más de 40 caracteres ni direcciones con más de 100 caracteres de largo. Si usa menús para presentar entradas permitidas, evitará algunos de los problemas de validación de entrada.

Lineamiento 8: Compartimente sus activos

Compartimentar significa que no debe permitir el acceso a la información en un sistema con el criterio de todo o nada. En vez de ello, debe organizar en compartimentos la información en un sistema. Los usuarios sólo deben tener acceso a la información que necesitan, y no a toda la información en un sistema. Esto significa que es posible que los efectos de un ataque se contengan. Tal vez se pierda o se dañe alguna información, pero es improbable que resulte afectada toda la información en el sistema.

Por ejemplo, en el sistema de información de pacientes, debe diseñar el sistema de manera que, en cualquier clínica, el personal hospitalario tenga normalmente sólo acceso a los registros de los pacientes que tienen una cita en esa clínica. Por lo general, no deben tener acceso a todos los registros de pacientes en el sistema. Esto no sólo limita la pérdida potencial de ataques internos, sino también significa que, si un intruso roba sus credenciales, la cantidad de daño que pueda causar será limitada.

Por otro lado, tal vez también deba tener mecanismos en el sistema para garantizar el acceso inesperado, por ejemplo, de un paciente seriamente enfermo que requiere tratamiento urgente sin tener cita. En tales circunstancias, puede usar algún mecanismo seguro alternativo para superar la compartimentación en el sistema. En esas situaciones, donde la seguridad se relaja para mantener la disponibilidad del sistema, es esencial contar con un mecanismo de acceso para registrar el uso del sistema. Siendo así, se puede comprobar las bitácoras con la finalidad de rastrear cualquier uso no autorizado.

Lineamiento 9: Diseñe para implementar

Muchos problemas de seguridad surgen porque el sistema no está configurado correctamente al implementarse en su entorno operacional. Por lo tanto, siempre debe diseñar su sistema de forma que se incluyan instalaciones para simplificar la implementación en el entorno del cliente, y comprobar errores y omisiones potenciales de configuración en el sistema implementado. Éste es un tema importante, que se analiza a detalle en la sección 14.2.3.

Lineamiento 10: Diseñe para facilitar la recuperabilidad

Sin importar cuánto esfuerzo se use en mantener la seguridad de los sistemas, siempre debe diseñar su sistema con la idea de que podría ocurrir una falla de seguridad. De ahí que deba pensar en cómo recuperarse de posibles fallas y restaurar el sistema a un estado operativo seguro. Por ejemplo, podría incluir un sistema de autenticación de respaldo en caso de que se comprometa su autenticación de contraseña.

Suponga que una persona no autorizada, externa a la clínica, consigue acceso al sistema de registros de los pacientes, y usted no sabe cómo obtuvo una combinación de nombre y contraseña válida. Necesita reiniciar el sistema de autenticación y no sólo cambiar las credenciales usadas por el intruso. Esto es esencial, porque el intruso también puede conseguir acceso a otras contraseñas de usuario. Por lo tanto, es preciso asegurarse de que todos los usuarios autorizados cambien sus contraseñas. También debe cerciorarse de que la persona sin autorización no tiene acceso al mecanismo de cambio de contraseñas.

En consecuencia, debe diseñar su sistema para negar el acceso a todos (hasta que se cambien sus contraseñas y se autentique a los usuarios reales para cambiar la contraseña), y suponer que sus contraseñas seleccionadas pueden no ser seguras. Una forma de hacer esto es mediante un mecanismo de pregunta/respuesta, donde los usuarios deben responder preguntas para las cuales tienen respuestas previamente registradas. Esto sólo

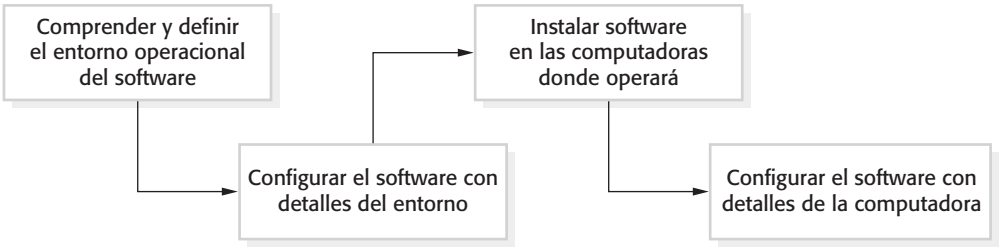


Figura 14.7 Implementación del software

se solicita al cambiar las contraseñas, lo que permite la recuperación de un ataque con relativamente poca perturbación del usuario.

14.2.3 Diseño para implementar

La implementación de un sistema implica configurar el software para funcionar en un entorno operacional, instalar el sistema en las computadoras en ese entorno y, posteriormente, configurar el sistema instalado para dichas computadoras (figura 14.7). La configuración puede ser un proceso simple que implique establecer algunos parámetros internos en el software para reflejar las preferencias del usuario. Sin embargo, en ocasiones, la configuración es compleja y requiere de la definición específica de los modelos y las reglas empresariales que afectan la ejecución del software.

Con frecuencia, en esta etapa del proceso se introducen accidentalmente vulnerabilidades en el software. Por ejemplo, durante la instalación, el software tiene a menudo que configurarse con una lista de usuarios permitidos. Cuando esta lista se entrega, consta simplemente de un acceso de administrador genérico como "admin" y una contraseña por defecto, como "password". Lo anterior facilita que un administrador configure el sistema. La primera acción debe ser introducir un nuevo nombre de acceso y contraseña, y borrar el nombre de acceso genérico. No obstante, es fácil olvidar hacer esto. Un atacante que conozca el acceso por defecto podrá obtener acceso privilegiado al sistema.

Configuración e implementación se ven a menudo como temas de administración del sistema y, por consiguiente, se consideran fuera del ámbito de los procesos de ingeniería de software. Desde luego, la buena práctica administrativa ayuda a evitar muchos problemas de seguridad que surgen de errores de configuración e implementación. Sin embargo, los diseñadores de software tienen la responsabilidad de "diseñar para la implementación". Siempre se debe brindar soporte interno para la implementación, lo cual reducirá la probabilidad de que los administradores del sistema (o usuarios) cometan errores al configurar el software.

Se recomiendan cuatro formas de incorporar soporte de implementación en un sistema:

1. *Incluir soporte para ver y analizar las configuraciones* Siempre es conveniente adicionar instalaciones en un sistema que permitan a los administradores o a los usuarios examinar la configuración actual del sistema. De manera sorprendente, esta facilidad está ausente en la mayoría de los sistemas de software, y los usuarios se frustran ante las dificultades de encontrar parámetros de configuración. Por ejemplo, en la versión del procesador de texto que se usó para escribir este capítulo, es imposible ver o imprimir en una sola pantalla los parámetros de todas las preferencias del sistema. No obstante, si un administrador es capaz de obtener una imagen completa

de una configuración, es más probable que especifique los errores y las omisiones. Lo ideal es que una pantalla de configuración también pueda destacar los aspectos de la configuración que sean potencialmente inseguros, por ejemplo, si no se estableció una contraseña.

2. *Minimizar los privilegios por defecto* El software debe diseñarse de forma que la configuración por defecto de un sistema ofrezca mínimos privilegios esenciales. De esta forma, el daño que cualquier atacante pudiera hacer quedará limitado. Por ejemplo, la autenticación por defecto del administrador del sistema sólo debe permitir el acceso a un programa que permita a un administrador establecer nuevas credenciales. Debe prohibir el acceso a alguna otra instalación del sistema. Una vez establecidas las nuevas credenciales, deben borrarse automáticamente por defecto el nombre y la contraseña.

3. *Localizar parámetros de configuración* Cuando se diseñe soporte para la configuración del sistema, habrá que asegurarse de que todo en una configuración que afecte la misma parte de un sistema se configure en el mismo lugar. Nuevamente con el ejemplo del procesador de texto, en la versión que se utilizó, pudo configurarse alguna información de seguridad, como una contraseña para controlar el acceso al documento, en el menú Preferencias/Seguridad. Otra información se configura en el menú Herramientas/Proteger documento. Si no se localiza la información de configuración, es fácil olvidar configurarla o, en algunos casos, ni siquiera se estará al tanto de que en el sistema se incluyen algunas instalaciones de seguridad.

4. *Proporcionar formas sencillas de corregir vulnerabilidades de seguridad* Hay que incluir mecanismos directos para actualizar el sistema y reparar las vulnerabilidades de seguridad que se hayan descubierto. Éstos podrían incluir comprobación automática para actualizaciones de seguridad, o la descarga de dichas actualizaciones tan pronto como estén disponibles. Es importante que los usuarios no puedan pasar por alto dichos mecanismos pues, inevitablemente, considerarán otro trabajo como más importante. Existen muchos ejemplos registrados de problemas de seguridad mayores que surgen (por ejemplo, falla completa de una red hospitalaria) porque los usuarios no actualizaron su software cuando se les pidió que lo hicieran.

14.3 Supervivencia del sistema

Hasta el momento, se estudió la ingeniería de seguridad desde la perspectiva de una aplicación que está bajo desarrollo. El proveedor y el desarrollador del sistema tienen control sobre todos los aspectos del sistema que pudieran atacarse. En realidad, como se sugiere en la figura 14.1, los modernos sistemas distribuidos se apoyan necesariamente en una infraestructura que incluye sistemas comerciales y componentes reutilizables desarrollados por diferentes organizaciones. La seguridad de dichos sistemas no depende sólo de las decisiones de diseño locales, sino también de la seguridad de aplicaciones externas, servicios Web y la infraestructura de red.

Esto significa que, sin importar cuánta atención se ponga a la seguridad, no es posible garantizar que un sistema podrá resistir ataques externos. En consecuencia, para sistemas complejos en red, debe suponerse que es posible la penetración y que la integridad del sistema no está garantizada. Por lo tanto, se debe pensar en cómo hacer que el sistema sea resistente, de manera que sobreviva para entregar servicios esenciales a los usuarios.

La supervivencia o resiliencia (Westmark, 2004) es una propiedad emergente de un sistema como totalidad, y no una propiedad de componentes individuales, que en sí mismos podrían no ser supervivientes. La supervivencia de un sistema refleja su capacidad para continuar la entrega de servicios empresariales esenciales o críticos para la misión, y para legitimar a los usuarios mientras está bajo ataque o después de que se dañó parte del sistema. El daño podría ser causado por un ataque o una falla del sistema.

El trabajo en la supervivencia del sistema apunta al hecho de que las vidas económica y social dependen de una infraestructura crítica controlada por computadoras. Ésta incluye la infraestructura para entregar servicios públicos (electricidad, agua, gas, etcétera) y la infraestructura para entregar y gestionar información (teléfonos, Internet, servicio postal, etcétera). Sin embargo, la supervivencia no es simplemente un asunto crítico de infraestructura . Cualquier organización que se apoye en sistemas de cómputo críticos en red debe interesarse por la forma como resultaría afectada si sus sistemas no sobreviven a un ataque malicioso o a una falla catastrófica del sistema. Por lo tanto, para sistemas empresariales críticos, el análisis y el diseño de supervivencia deben ser parte del proceso de ingeniería de seguridad.

Conservar la disponibilidad de los servicios críticos es la esencia de la supervivencia. Esto significa que usted debe conocer:

- los servicios del sistema más críticos para una empresa;
- la calidad mínima del servicio a preservar;
- cómo pueden comprometerse dichos servicios;
- cómo pueden protegerse tales servicios;
- cómo recuperarse rápidamente si los servicios dejan de estar disponibles.

Por ejemplo, en un sistema que administra la salida de ambulancias en respuesta a llamadas de emergencia, los servicios críticos son aquellos relacionados con la recepción de llamadas y el envío de ambulancias a la emergencia médica. Otros servicios, como el registro de las llamadas y la administración de la ubicación de las ambulancias, son menos críticos, ya sea porque no requieren procesamiento en tiempo real o porque existen mecanismos alternativos. Por ejemplo, para encontrar la ubicación de una ambulancia es posible llamar al personal de la unidad y preguntarle la ubicación.

Ellison y sus colaboradores (1999a; 199b; 2002) diseñaron un método de análisis llamado Survivable Systems Analysis (análisis de sistemas supervivientes). Se usa para valorar las vulnerabilidades en los sistemas y apoyar el diseño de arquitecturas y características de los sistemas que promueven la supervivencia de éstos. Estos investigadores argumentan que el logro de la supervivencia depende de tres estrategias complementarias:

1. *Resistencia* Evitar los problemas mediante la construcción de capacidades en el sistema para repeler ataques. Por ejemplo, un sistema puede usar certificados digitales para autenticar a los usuarios, lo que dificulta el hecho de que usuarios no autorizados tengan acceso.
2. *Reconocimiento* Detectar los problemas mediante la construcción de capacidades en el sistema para descubrir ataques y fallas, además de valorar el daño resultante. Por ejemplo, podrían asociarse sumas de verificación con datos críticos, de manera que pueda detectarse la corrupción de los datos.
3. *Recuperación* Tolerar los problemas por medio de la construcción de capacidades en el sistema para entregar servicios esenciales mientras está bajo ataque, y recuperar la

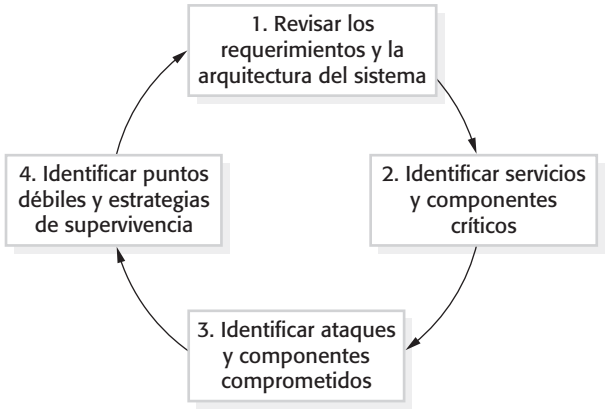


Figura 14.8 Etapas en el análisis de supervivencia

funcionalidad plena después de un ataque. Por ejemplo, los mecanismos de tolerancia a fallas en el desarrollo que usan diversas implementaciones de la misma funcionalidad pueden incluirse para lidiar con una pérdida de servicio de una parte del sistema.

El análisis de sistemas supervivientes es un proceso de cuatro etapas (figura 14.8), que analiza los requerimientos y la arquitectura actual o las propuestas del sistema; identifica servicios críticos, escenarios de ataque y "puntos débiles" del sistema, y propone cambios para mejorar la supervivencia de un sistema. Las actividades clave en cada una de dichas etapas son las siguientes:

1. *Comprensión del sistema* Para un sistema existente o propuesto, se revisan las metas del sistema (llamadas en ocasiones "objetivos de la misión"), los requerimientos y la arquitectura del sistema.
2. *Identificación de servicios críticos* Se identifican los servicios que siempre deben mantenerse y los componentes que se requieren para conservar dichos servicios.
3. *Simulación de ataque* Se identifican los escenarios o casos de uso para posibles ataques, junto con los componentes del sistema que resultarían afectados por dichos ataques.
4. *Análisis de supervivencia* Se identifican los componentes que son esenciales y que podrían resultar comprometidos por un ataque, así como las estrategias de supervivencia basadas en resistencia, reconocimiento y recuperación.

Ellison y sus colaboradores presentan un excelente estudio de caso del método basado en un sistema para apoyar el tratamiento de salud mental (1999b). Este sistema es similar al MHC-PMS que se usó como ejemplo en este libro. En vez de repetir su análisis, se utiliza el sistema de comercio de acciones, que se muestra en la figura 14.5, para ilustrar algunas de las características del análisis de supervivencia.

Como se observa en la figura 14.5, este sistema ya hizo alguna provisión de supervivencia. Las cuentas de usuarios y los precios de las acciones se duplican a través de los servidores, de manera que pueden realizarse pedidos incluso si el servidor local no está disponible. Suponga que la capacidad de que usuarios autorizados realicen pedidos de acciones es el servicio clave que debe mantenerse. Para garantizar que los usuarios confíen en el sistema, es esencial mantener la integridad. Los pedidos deben ser precisos y reflejar las ventas o compras reales hechas por un usuario del sistema.

Ataque	Resistencia	Reconocimiento	Recuperación
Usuario no autorizado realiza pedidos maliciosos	Para realizar pedidos, se solicita una contraseña de negociación diferente de la de acceso.	Enviar por correo electrónico copia del pedido al usuario autorizado, con número telefónico de contacto (de forma que puedan detectarse pedidos maliciosos). Mantener un historial de pedidos de usuario y verificar patrones de negociación inusuales.	Proporcionar mecanismos para "deshacer" automáticamente las negociaciones y restaurar las cuentas de usuario. Reembolsar a los usuarios las pérdidas que se deben a negociación maliciosa. Adquirir seguros contra pérdidas derivadas de ello.
Corrupción de base de datos de transacciones	Requerir la autorización de usuarios privilegiados mediante un mecanismo de autenticación más confiable, como certificados digitales.	Mantener copias de sólo lectura de las transacciones para una oficina en un servidor internacional. Comparar periódicamente las transacciones para detectar casos de corrupción. Mantener sumas de verificación criptográficas con todos los registros de transacción para detectar la corrupción.	Recuperar la base de datos a partir de copias de respaldo. Ofrecer un mecanismo para reproducir las negociaciones desde una hora especificada para recrear la base de datos de transacciones.

Figura 14.9 Análisis de supervivencia en un sistema de comercio de acciones

Para mantener este servicio de pedidos, existen tres componentes del sistema que se utilizan:

1. *Autenticación de usuario* Esto permite que usuarios autorizados ingresen al sistema.
2. *Cotización de precios* Esto permite la cotización del precio de compra y venta de una acción.
3. *Colocación de pedidos* Esto permite la realización de pedidos de compra y venta a un precio dado.

Obviamente, dichos componentes usan los activos de datos esenciales, como base de datos de cuentas de usuarios, una base de datos de precios y una base de datos de transacción de pedidos. Éstos deben sobrevivir a los ataques si se desea mantener el servicio.

Existen muchos tipos diferentes de ataques sobre este sistema que podrían cometerse. Considere aquí dos posibilidades:

1. Un usuario malicioso siente rencor contra un usuario acreditado del sistema. Consigue acceso al sistema usando sus credenciales. Coloca pedidos maliciosos y compra y vende acciones, con la intención de causar problemas al usuario autorizado.
2. Un usuario no autorizado corrompe la base de datos de transacciones al conseguir permiso para emitir directamente comandos SQL. Por lo tanto, es imposible la reconciliación de ventas y compras.

La figura 14.9 muestra ejemplos de estrategias de resistencia, reconocimiento y recuperación que ayudan a contrarrestar estos ataques.

Desde luego, aumentar la supervivencia o la resiliencia de un sistema cuesta dinero. Las compañías tal vez se muestren renuentes a invertir en supervivencia si nunca han sufrido un ataque serio o pérdidas asociadas. Sin embargo, así como es mejor comprar buenas cerraduras y una alarma para evitar que se introduzcan ladrones a su casa, también es mejor invertir en supervivencia antes de sufrir un ataque. El análisis de supervivencia todavía no es parte de la mayoría de los procesos de ingeniería de software pero, conforme más sistemas se convierten en críticos para la empresa, es probable que tales análisis se usen más ampliamente.

PUNTOS CLAVE

- La ingeniería de seguridad se enfoca en cómo desarrollar y mantener sistemas de software capaces de resistir ataques maliciosos, los cuales tienen la intención de dañar un sistema basado en computadoras o sus datos.
- Las amenazas a la seguridad pueden ser amenazas a la confidencialidad, integridad o disponibilidad de un sistema o sus datos.
- La gestión del riesgo de seguridad implica diseñar una arquitectura segura de sistema, seguir buenas prácticas para diseñar sistemas seguros e incluir funcionalidad para minimizar la posibilidad de introducir vulnerabilidades al implementar el sistema.
- El diseño de seguridad implica diseñar una arquitectura de sistema que conserve la seguridad, seguir buenas prácticas para el diseño de sistemas seguros e incluir funcionalidad para minimizar la posibilidad de introducir vulnerabilidades cuando se despliega el sistema.
- Los temas clave cuando se diseña una arquitectura segura de sistemas incluyen organizar la estructura del sistema para proteger los activos clave y distribuir los activos del sistema para minimizar las pérdidas que podría ocasionar un ataque.
- Los lineamientos de diseño de seguridad sensibilizan a los diseñadores del sistema ante los asuntos de seguridad que quizá pasaron por alto. Ofrecen una base para elaborar listas de verificación que permitan revisar la seguridad.
- Para apoyar la implementación segura debe existir una forma de implementar y analizar las configuraciones del sistema, localizar parámetros de configuración de forma que no se olviden configuraciones importantes, minimizar los privilegios por defecto asignados a los usuarios del sistema, y ofrecer formas para reparar vulnerabilidades de seguridad.
- La supervivencia del sistema refleja su capacidad de continuar la entrega de servicios esenciales para la empresa o críticos para la misión, con la finalidad de legitimar a los usuarios mientras el sistema se encuentra bajo ataque, o después de que parte del sistema se ha dañado.

LECTURAS SUGERIDAS

"Survivable Network System Analysis: A Case Study." Un excelente ensayo que introduce la noción de supervivencia del sistema; refiere un estudio de caso de un sistema de registros de tratamiento

de salud mental para ilustrar la aplicación de un método de supervivencia. (R. J. Ellison, R. C. Linger, T. Longstaff y N. R. Mead, *IEEE Software*, **16** (4), julio/agosto de 1999.)

Building Secure Software: How to Avoid Security Problems the Right Way. Un buen libro práctico que analiza la seguridad desde una perspectiva de programación. (J. Viega y G. McGraw, Addison-Wesley, 2002.)

Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd edition. Se trata de una discusión amplia y profunda de los problemas para construir sistemas seguros. Se enfoca en los sistemas y no en la ingeniería del software; ofrece una amplia cobertura de hardware y redes, mediante excelentes ejemplos extraídos de fallas de sistemas reales. (R. Anderson, John Wiley & Sons, 2008.)

EJERCICIOS

14.1. Explique las importantes diferencias entre ingeniería de seguridad de aplicación e ingeniería de seguridad de infraestructura.

14.2. Para el MHC-PMS, sugiera un ejemplo de activo, exposición, vulnerabilidad, ataque, amenaza y control.

14.3. Explique por qué hay necesidad de que la valoración del riesgo sea un proceso continuo, desde las primeras etapas de la ingeniería de requerimientos hasta el uso operativo de un sistema.

14.4. Con base en sus respuestas a la pregunta 2 acerca del MHC-PMS, valore los riesgos asociados con ese sistema y proponga dos requerimientos de sistema que permitan reducir tales riesgos.

14.5. Explique, con una analogía extraída de un contexto de ingeniería, no de software, por qué debe usarse un enfoque en capas para la protección de activos.

14.6. Explique por qué es importante usar tecnologías diversas para dar soporte a sistemas distribuidos en situaciones en que la disponibilidad del sistema es un asunto crítico.

14.7. ¿Qué es ingeniería social? ¿Por qué es difícil protegerse contra ella en las organizaciones grandes?

14.8. Para cualquier sistema de software comercial que use (por ejemplo, Microsoft Word), analice las instalaciones de seguridad incluidas y examine cualquier problema que encuentre.

14.9. Explique cómo pueden usarse las estrategias complementarias de resistencia, reconocimiento y recuperación para mejorar la supervivencia de un sistema.

14.10. Para el sistema de comercio de acciones que se describe en la sección 14.2.1, cuya arquitectura se ilustra en la figura 14.5, sugiera dos posibles ataques ulteriores sobre el sistema y proponga algunas estrategias que pudieran contrarrestar dichos ataques.

REFERENCIAS

Alberts, C. y Dorofee, A. (2002). *Managing Information Security Risks: The OCTAVE Approach*. Boston: Addison-Wesley.

Alexander, I. (2003). "Misuse Cases: Use Cases with Hostile Intent". *IEEE Software*, **20** (1), 58–66.

Anderson, R. (2008). *Security Engineering, 2nd edition*. Chichester: John Wiley & Sons.

Berghel, H. (2001). "The Code Red Worm". *Comm. ACM*, **44** (12), 15–19.

Bishop, M. (2005). *Introduction to Computer Security*. Boston: Addison-Wesley.

Cranor, L. y Garfinkel, S. (2005). *Security and Usability: Designing secure systems that people can use*. Sebastopol, Calif.: O'Reilly Media Inc.

Ellison, R., Linger, R., Lipson, H., Mead, N. y Moore, A. (2002). "Foundations of Survivable Systems Engineering". *Crosstalk: The Journal of Defense Software Engineering*, **12**, 10–15.

Ellison, R. J., Fisher, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. A. y Mead, N. R. (1999a). "Survivability: Protecting Your Critical Systems". *IEEE Internet Computing*, **3** (6), 55–63.

Ellison, R. J., Linger, R. C., Longstaff, T. y Mead, N. R. (1999b). "Survivable Network System Analysis: A Case Study". *IEEE Software*, **16** (4), 70–7.

Pfleeger, C. P. y Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.

Schneier, B. (2000). Secrets and Lies: *Digital Security in a Networked World*. Nueva York: John Wiley & Sons.

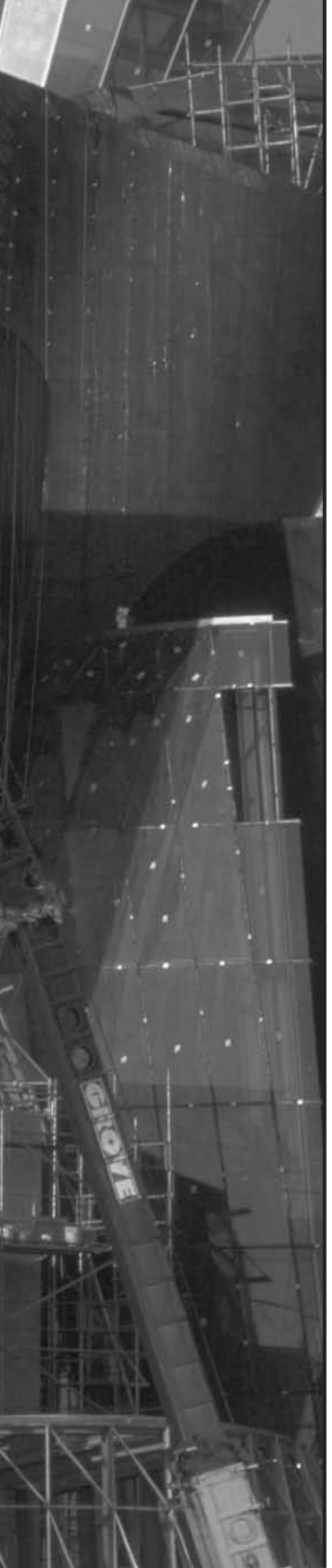
Sindre, G. y Opdahl, A. L. (2005). "Eliciting Security Requirements through Misuse Cases". *Requirements Engineering*, **10** (1), 34–44.

Spafford, E. (1989). "The Internet Worm: Crisis and Aftermath". *Comm ACM*, **32** (6), 678–87.

Viega, J. y McGraw, G. (2002). *Building Secure Software*. Boston: Addison-Wesley.

Westmark, V. R. (2004). "A Definition for Information System Survivability". 37th Hawaii Int. Conf. on System Sciences, Hawaii: 903–1003.

Wheeler, D. A. (2003). *Secure Programming for Linux and UNix HOWTO*. Web published: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html.



15

Garantía de confiabilidad y seguridad

Objetivos

El objetivo de este capítulo es describir las técnicas de verificación y validación usadas en el desarrollo de sistemas críticos. Al estudiar este capítulo:

- comprenderá cómo pueden utilizarse los diferentes enfoques del análisis estático en la verificación de sistemas de software críticos;
- entenderá los fundamentos de las pruebas de confiabilidad y seguridad, así como los problemas inherentes a las pruebas de los sistemas críticos;
- conocerá por qué es importante el aseguramiento del proceso, en especial para el software que debe certificar un organismo regulador;
- se introducirá en casos de protección y confiabilidad que presenten argumentos y evidencia de la protección y confiabilidad del sistema.

Contenido

15.1 Análisis estático
15.2 Pruebas de fiabilidad
15.3 Pruebas de seguridad
15.4 Aseguramiento del proceso
15.5 Casos de protección y confiabilidad

Garantizar la confiabilidad y seguridad es una actividad que se interesa por comprobar que un sistema crítico cubra los requerimientos de confiabilidad. Esto requiere procesos de verificación y validación (V&V) que buscan errores de especificación, diseño y programa que puedan afectar la disponibilidad, protección, fiabilidad o seguridad de un sistema.

La verificación y la validación de un sistema crítico tienen mucho en común con la validación de cualquier otro sistema de software. Los procesos V&V deben demostrar que el sistema cumple la especificación, y que los servicios y el comportamiento del sistema apoyan los requerimientos del cliente. Al hacerlo, descubren por lo general errores de requerimientos y diseño, así como bugs de programa que deben repararse. Sin embargo, por dos razones, los sistemas críticos requieren de pruebas y análisis especialmente rigurosos:

1. *Costos de la falla* Los costos y las consecuencias de la falla de los sistemas críticos son potencialmente mucho mayores que los sistemas no críticos. Los riesgos de falla del sistema se reducen al invertir más en verificación y validación del sistema. A menudo es menos costoso encontrar y eliminar los defectos antes de que el sistema se entregue, que sufragar los costos derivados de accidentes o interrupciones al servicio del sistema.

2. *Validación de atributos de confiabilidad* Tal vez usted deba elaborar un argumento formal para los clientes y para un organismo regulador de que el sistema satisface los requerimientos de confiabilidad especificados (disponibilidad, fiabilidad, protección y seguridad). En algunos casos, es posible que reguladores externos, tales como las autoridades de aviación nacional, deban certificar que el sistema es seguro antes de implementarse. Para obtener esta certificación, deberá demostrarse cómo se validó el sistema. En tal caso, quizá tenga que diseñar y realizar procedimientos V&V especiales que recopilen evidencia sobre la confiabilidad del sistema.

Por estas razones, los costos de verificación y validación para sistemas críticos con frecuencia son mucho mayores que para otras clases de sistemas. Normalmente, más de la mitad de los costos de desarrollo de un sistema crítico se destinan a procesos V&V.

Aunque los costos V&V son altos, se justifican, ya que por lo común son considerablemente menores que las pérdidas originadas por un accidente. Por ejemplo, en 1996, un sistema de software crítico para una misión en el cohete Ariane 5 falló, y numerosos satélites fueron destruidos. Aunque nadie resultó lesionado, las pérdidas totales de ese accidente fueron de cientos de millones de dólares. Una investigación descubrió que las deficiencias en el sistema V&V fueron en parte responsables de esta falla. Revisiones más efectivas, relativamente poco costosas, habrían descubierto el problema que causó el accidente.

Aun cuando el enfoque principal para garantizar la confiabilidad y seguridad se encuentra en la validación del sistema en sí, actividades relacionadas deben verificar que se siga el proceso definido para el desarrollo del sistema. Como se explicó en el capítulo 13, la calidad del sistema resulta afectada por la calidad de los procesos utilizados para desarrollar el sistema. En resumen, buenos procesos conducen a buenos sistemas.

El resultado de los procesos para garantizar la confiabilidad y seguridad es un conjunto de evidencia tangible, tal como los reportes de revisión, resultados de pruebas, etcétera, acerca de la confiabilidad de un sistema. Esta evidencia puede utilizarse posteriormente para justificar una decisión de que el sistema es bastante confiable y seguro

para implementar y usar. En ocasiones, la evidencia de la confiabilidad del sistema se integra en un caso de confiabilidad o protección. Esto se usa para convencer a un cliente o a un regulador externo de la confiabilidad o protección del sistema.

15.1 Análisis estático

Las técnicas de análisis estático son técnicas de verificación del sistema que no incluyen la ejecución de un programa. En vez de ello, funcionan sobre una representación fuente del software: un modelo de especificación o diseño, o el código fuente del programa. Las técnicas de análisis estático pueden usarse para comprobar la especificación y los modelos de diseño de un sistema con la finalidad de buscar errores antes de que esté disponible una versión ejecutable del sistema. También tienen la ventaja de que la presencia de errores no interrumpe la comprobación del sistema. Cuando se prueba un programa, los defectos pueden enmascarar u ocultar otros defectos, de manera que se debe eliminar un defecto detectado y luego repetir el proceso de prueba.

Como se estudió en el capítulo 8, tal vez la técnica de análisis estático usada con más frecuencia sea la revisión e inspección por pares, en la que un grupo de personas se encargan de comprobar una especificación, un diseño o un programa. Ellos analizan a detalle el diseño o código, y examinan posibles errores u omisiones. Otra técnica consiste en utilizar herramientas de modelado de diseño para comprobar anomalías en el UML, por ejemplo, el hecho de que el mismo nombre sea usado por diferentes objetos. Sin embargo, para sistemas críticos, pueden emplearse técnicas adicionales de análisis estático:

1. Verificación formal, en la que se producen argumentos matemáticamente rigurosos de que un programa se conforma a su especificación.
2. Comprobación de modelo, en la que se usa un verificador de teoremas para revisar una descripción formal del sistema en busca de inconsistencias.
3. Análisis automatizado de programa, en el que el código fuente de un programa se revisa por patrones que, según se sabe, son potencialmente erróneos.

Dichas técnicas están estrechamente relacionadas. La comprobación de modelos se apoya en un modelo formal del sistema que puede crearse a partir de una especificación formal. Los analizadores estáticos utilizan aseveraciones formales incrustadas en un programa como comentarios, para comprobar que el código asociado es inconsistente con dichas aseveraciones.

15.1.1 Verificación y métodos formales

Los métodos formales del desarrollo de software, como se estudió en el capítulo 12, se apoyan en un modelo formal que sirve como especificación del sistema. Dichos métodos formales se interesan principalmente por un análisis matemático de la especificación, por transformar la especificación a una representación más detallada, semánticamente equivalente, o por verificar de manera formal que una representación del sistema es semánticamente equivalente a otra representación.

Desarrollo de Cleanroom

El desarrollo de software Cleanroom (cuarto limpio) se basa en una verificación del software formal y pruebas estadísticas. El objetivo del proceso Cleanroom es obtener un software con cero defectos para garantizar que los sistemas entregados tengan un alto nivel de fiabilidad. En el proceso Cleanroom, cada incremento de software se especifica formalmente, y esta especificación se transforma en una implementación. La corrección del software se demuestra mediante un enfoque formal. En el proceso no hay prueba de unidad por defectos, y las pruebas del sistema se enfocan en la valoración de la fiabilidad del sistema.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

Los métodos formales pueden usarse en diferentes etapas durante el proceso V&V:

1. Una especificación formal del sistema puede desarrollarse y analizarse matemáticamente para identificar inconsistencias. Esta técnica es efectiva para detectar errores y omisiones de especificación. La comprobación del modelo, que se explica en la siguiente sección, es un enfoque al análisis de especificación.
2. Es posible verificar formalmente, mediante argumentos matemáticos, que el código de un sistema de software es consistente con esta especificación, pero ello requiere una especificación formal. Esto permite descubrir errores de programación y algunos de diseño.

Dada la amplia brecha semántica entre una especificación de sistema formal y el código del programa, es difícil probar que un programa desarrollado por separado es consistente con su especificación. En consecuencia, ahora el trabajo en la verificación del programa se basa en el desarrollo transformacional. En un proceso de desarrollo transformacional, una especificación formal se transforma mediante una serie de representaciones en un código de programa. Las herramientas de software apoyan el desarrollo de las transformaciones y ayudan a verificar que las correspondientes representaciones del sistema sean consistentes. Tal vez el método B sea el método transformacional formal usado más ampliamente (Abrial, 2005; Wordsworth, 1996). Se ha utilizado para el desarrollo de sistemas de control ferroviario y software de aviones.

Los defensores de los métodos formales afirman que el uso de dichos métodos conduce a sistemas más confiables y seguros. La verificación formal demuestra que el programa desarrollado satisface su especificación y que los errores de la implementación no comprometerán la confiabilidad del sistema. Si se desarrolla un método formal de sistemas concurrentes mediante el uso de una especificación escrita en un lenguaje como CSP (Schneider, 1999), es posible descubrir condiciones que quizá den por resultado bloqueos en el programa final, lo que permitirá resolverlos, algo muy difícil de lograr sólo con las pruebas.

Sin embargo, la especificación formal y las pruebas no garantizan que el software sea fiable en el uso práctico. Las razones son las siguientes:

1. La especificación tal vez no refleje los requerimientos reales de los usuarios del sistema. Como se estudió en el capítulo 12, los usuarios del sistema rara vez entienden las notaciones formales, de manera que no pueden leer directamente la especificación formal para encontrar errores y omisiones. Esto significa que hay una considerable probabilidad de que la especificación formal contenga errores y no sea una representación precisa de los requerimientos del sistema.

2. La prueba puede contener errores. Las pruebas del programa son extensas y complejas, de manera que, como programas extensos y complejos, por lo general contienen errores.
3. La prueba puede hacer suposiciones incorrectas sobre la forma de utilizar el sistema. Si éste no se utiliza como se anticipaba, la prueba podría ser inválida.

Verificar un sistema de software no trivial requiere gran cantidad de tiempo y experiencia matemática, así como herramientas de software especializadas, tales como los demostradores de teoremas. Por lo tanto, es un proceso costoso y, conforme aumenta el tamaño del sistema, los costos de la verificación formal se incrementan de manera desproporcionada. En consecuencia, muchos ingenieros de software consideran que la verificación formal no es efectiva en términos de costo. Suponen que puede lograrse el mismo nivel de confianza en el sistema, de manera menos costosa, si se utilizan otras técnicas de validación, tales como las inspecciones y las pruebas del sistema.

A pesar de sus desventajas, la posición en este texto es que los métodos formales y la verificación formal desempeñan un importante papel en el desarrollo de los sistemas de software críticos. Las especificaciones formales son bastante efectivas para descubrir aquellos problemas de especificación que son las causas más comunes de falla del sistema. Aunque la verificación formal todavía no es práctica para sistemas grandes, puede usarse en la verificación de los componentes críticos para la protección y la seguridad.

15.1.2 Comprobación del modelo

Verificar de manera formal un programa mediante un enfoque deductivo es difícil y costoso, pero se han desarrollado enfoques alternativos al análisis formal que se basan en una noción más restringida de la corrección. El más exitoso de estos enfoques se llama comprobación del modelo (Baier y Katoen, 2008). Éste se usa ampliamente para comprobar diseños de sistemas de hardware y cada vez más en sistemas de software críticos, como el software de control en los vehículos de la NASA para explorar la superficie de Marte (Regan y Hamilton, 2004) y el software de procesamiento de llamadas telefónicas (Chandra *et al*., 2002).

La comprobación del modelo implica la creación de un modelo de sistema y la comprobación de la corrección de dicho modelo mediante herramientas especializadas de software. Se han desarrollado muchas y diferentes herramientas de comprobación del modelo; para software, tal vez la que se utiliza más ampliamente sea SPIN (Holzmann, 2003). En la figura 15.1 se muestran las etapas que comprende la comprobación del modelo.

El proceso de comprobación del modelo implica construir un modelo formal de un sistema, por lo general como una máquina de estado finito extendida. Los modelos se expresan en el lenguaje de cualquier sistema usado de comprobación del modelo: por ejemplo, el comprobador de modelo SPIN emplea un lenguaje llamado Promela. Un conjunto de propiedades deseables del sistema se identifica y escribe en una notación formal, por lo general con base en lógica temporal. Un ejemplo de tal propiedad en el sistema de estación meteorológica a campo abierto es que el sistema siempre llegará al estado “transmitir” desde el estado “registrar”.

En tal caso, el comprobador de modelo explora todas las rutas a lo largo del modelo (es decir, todas las posibles transiciones de estado) y comprueba que se sostenga la propiedad para cada ruta. Si lo hace, entonces el comprobador de modelo confirma que el modelo es correcto con respecto a dicha propiedad. Si no se sostiene para una ruta particular, el comprobador del modelo presenta un contraejemplo que ilustra dónde no es verdadera la propiedad. La comprobación del modelo es particularmente útil en la

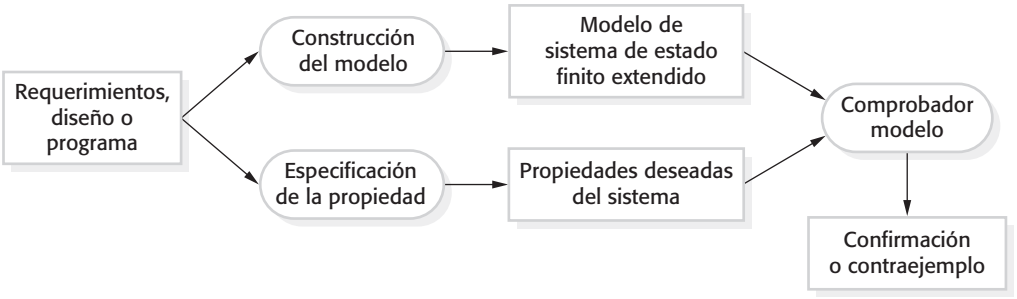


Figura 15.1 Comprobación del modelo

validación de sistemas concurrentes, que son notoriamente difíciles de probar debido a su sensibilidad al tiempo. El comprobador puede explorar transiciones concurrentes entremezcladas y descubrir problemas potenciales.

Un conflicto clave en la comprobación de modelos es la creación del modelo del sistema. Si el modelo debe crearse manualmente (a partir de un documento de requerimientos o de diseño), es un proceso costoso, pues la elaboración del modelo requiere de gran cantidad de tiempo. Además, existe la posibilidad de que el modelo creado no sea un modelo preciso de los requerimientos o del diseño. Por lo tanto, es mejor si el modelo puede diseñarse automáticamente a partir del código fuente del programa. El sistema Java Pathfinder (Visser *et al*., 2003) es un ejemplo de un sistema de comprobación de modelo que trabaja directamente desde una representación de código Java.

La comprobación del modelo es muy costosa desde el punto de vista computacional, pues usa un enfoque exhaustivo para comprobar todas las rutas a lo largo del modelo del sistema. Conforme aumenta el tamaño del sistema, sucede lo mismo con el número de estados, con un consecuente aumento en el número de rutas a comprobar. Esto significa que, para sistemas grandes, la comprobación del modelo podría resultar impráctica, debido al tiempo de cómputo requerido para operar las comprobaciones.

Sin embargo, a medida que mejoran los algoritmos que identifican aquellas partes del estado que no se han explorado para comprobar una propiedad particular, se volverá cada vez más práctico usar con frecuencia la comprobación del modelo en el desarrollo de sistemas críticos. En realidad, aunque no es aplicable a sistemas organizacionales orientados a datos, puede usarse para verificar sistemas de software embebidos que se modelan como máquinas de estado.

15.1.3 Análisis estático automático

Como se estudió en el capítulo 8, las inspecciones de programa están dirigidas con regularidad por listas de verificación de errores y heurística. Éstas identifican errores comunes en diferentes lenguajes de programación. Para algunos errores y heurísticas, es posible automatizar el proceso de comprobar programas contra dichas listas, lo que da por resultado el desarrollo de analizadores estáticos automatizados en que es posible encontrar fragmentos de código que sean incorrectos.

Las herramientas de análisis estático trabajan sobre el código fuente de un sistema y, al menos para ciertos tipos de análisis, no se requieren más entradas. Esto significa que los programadores no necesitan aprender notaciones especializadas para escribir especificaciones de programa para que los beneficios del análisis sean claros de inmediato. Esto hace que el análisis estático automatizado se introduzca con más facilidad en un proceso de desarrollo, que la verificación formal o la comprobación de modelo. Por lo tanto, quizás es la técnica de análisis estático de mayor uso.

Clase de fallas en el desarrollo	Comprobación de análisis estático
Fallas de datos	Variables usadas antes de inicialización. Variables declaradas pero nunca usadas. Variables asignadas dos veces pero nunca usadas entre asignaciones. Posibles violaciones de límites de arreglo. Variables no declaradas.
Fallas de control	Código inalcanzable. Ramas incondicionales en ciclos.
Fallas entrada/salida	Variables de salida dobles sin intervención de asignación.
Fallas de interfaz	Incompatibilidad del tipo de parámetro. Incompatibilidad del número de parámetro. No se usan resultados de funciones. Funciones y procedimientos no llamados.
Fallas de gestión de almacenamiento	Apuntadores no asignados. Aritmética de apuntadores. Fuga de memoria.

Figura 15.2 Comprobaciones de análisis estático automatizado

Los analizadores estáticos automatizados son herramientas de software que exploran el texto fuente de un programa y detectan posibles fallas en el desarrollo y anomalías. Examinan sintácticamente el texto del programa y, además, reconocen los diferentes tipos de enunciados en un programa. De esta forma, pueden detectar si los enunciados están bien formulados o no, hacer inferencias acerca del flujo de control en el programa y, en muchos casos, calcular el conjunto de todos los posibles valores para datos del programa. Éstos complementan las instalaciones de detección de error que proporciona el compilador de lenguaje, y pueden usarse como parte del proceso de inspección o como una actividad de proceso V&V separada. El análisis estático automatizado es más rápido y menos costoso que las revisiones de código detalladas. Sin embargo, no es capaz de descubrir algunas clases de errores que pudieran identificarse en sesiones de inspección del programa.

La intención del análisis estático automatizado es llamar la atención de un lector de código ante anomalías en el programa, tales como variables que se usan sin inicialización, variables que no se emplean o datos cuyo valor pudiera estar fuera de rango. En la figura 15.2 se muestran ejemplos de los problemas susceptibles de detección mediante análisis estático. Desde luego, las comprobaciones específicas realizadas son específicas del lenguaje de programación y dependen de qué se permite y qué no se permite en el lenguaje. Con frecuencia, las anomalías son resultado de errores u omisiones de programación, de manera que destacan aquello que pudiera salir mal cuando se ejecuta el programa. Sin embargo, debe quedar claro que dichas anomalías no necesariamente son fallas en el desarrollo del programa; pueden ser instrucciones deliberadas introducidas por el programador, o es posible que la anomalía no tenga consecuencias adversas.

Existen tres niveles de comprobación que pueden implementarse en analizadores estáticos:

1. *Comprobación de error característico* En este nivel, el analizador estático sabe acerca de errores comunes que cometen los programadores en lenguajes como Java o C. La herramienta analiza el código en busca de patrones que sean característicos de ese problema y los destaca para el programador. Aunque relativamente simple,

el análisis basado en errores comunes puede ser muy efectivo en términos de costo. Zheng y sus colaboradores (2006) estudiaron el uso de análisis estático contra una gran base de código en C y C++, y descubrieron que el 90% de los errores en los programas resultaron de 10 tipos de error característico.

2. *Comprobación de error definido por el usuario* En este enfoque, los usuarios del analizador estático pueden definir patrones de error, por lo tanto, se extienden los tipos de error que pueden detectarse. Esto es particularmente útil en situaciones donde debe mantenerse el orden (por ejemplo, el método A siempre debe llamarse antes que el método B). Con el tiempo, una organización podrá recolectar información acerca de bugs comunes que ocurren en sus programas y extender el análisis estático para destacar dichos errores.

3. *Comprobación de aserción* Éste es el enfoque más general y poderoso del análisis estático. Los desarrolladores incluyen aserciones formales (escritas con frecuencia en comentarios estilizados) en su programa, las cuales establecen relaciones que deben sostenerse en dicho punto del programa. Por ejemplo, puede incluirse una aserción que establezca que el valor de cierta variable debe encontrarse en el rango x..y. El analizador ejecuta simbólicamente el código y destaca los enunciados donde no se sostiene la aserción. Este enfoque se usa en analizadores como Splint (Evans y Larochelle, 2002) y el SPARK Examiner (Croxford y Sutton, 2006).

El análisis estático es efectivo para encontrar errores en programas, pero por lo regular genera un gran número de "falsos positivos". Se trata de secciones de código donde no hay errores, pero donde las reglas del analizador estático detectaron un potencial de error. Es posible reducir el número de falsos positivos al agregar más información al programa en forma de aserciones; desde luego, esto requiere trabajo adicional por parte del desarrollador del código. Habrá que trabajar en el tamizado de esos falsos positivos antes de que el código en sí pueda comprobarse en busca de errores.

El análisis estático es particularmente valioso para la comprobación de seguridad (Evans y Larochelle, 2002). Los analizadores estáticos pueden ajustarse a la medida para comprobar problemas bien conocidos, como desbordamiento de buffer o entradas sin verificar, que podrían aprovechar los atacantes. La comprobación de problemas bien conocidos es efectiva para mejorar la seguridad, pues la mayoría de los intrusos basan sus ataques en vulnerabilidades comunes.

Como se estudiará más adelante, las pruebas de seguridad son difíciles porque los atacantes a menudo actúan de forma inesperada, lo que dificulta que los examinadores puedan anticipar su comportamiento. Es posible que los analizadores estáticos incorporen experiencia de seguridad detallada que esté fuera del alcance de los examinadores, y que ésta se aplique antes de que un programa se ponga a prueba. Si se usa análisis estático, se pueden hacer afirmaciones que sean verdaderas para todas las posibles ejecuciones del programa, no sólo para aquellas que correspondan a las pruebas que ya se diseñaron.

Muchas organizaciones utilizan ahora el análisis estático de manera rutinaria en sus procesos de desarrollo de software. Microsoft introdujo el análisis estático en el desarrollo de controladores de dispositivo (Larus *et al*., 2003), donde las fallas de programa podrían tener un efecto serio. Ahora extendió el enfoque a través de un rango mucho más amplio de su software para buscar problemas de seguridad, así como errores que afectan la fiabilidad del programa (Ball *et al*., 2006). Muchos sistemas críticos, incluidos sistemas de aviones y nucleares, se someten por lo regular a análisis estático como parte del proceso V&V (Nguyen y Ourghanlian, 2003).

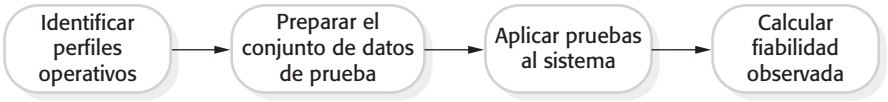


Figura 15.3 Medición de la fiabilidad

15.2 Pruebas de fiabilidad

Las pruebas de la fiabilidad son un proceso de pruebas cuya meta consiste precisamente en medir la fiabilidad de un sistema. Como se explicó en el capítulo 12, existen muchas métricas de fiabilidad, como POFOD, probabilidad de falla a pedido, y ROCOF, tasa de ocurrencia de falla. Dichas métricas permiten especificar cuantitativamente la fiabilidad requerida del software. Durante el proceso de pruebas de fiabilidad es posible comprobar si el sistema logró el nivel de fiabilidad requerido.

En la figura 15.3 se ilustra el proceso para medir la fiabilidad de un sistema. Este proceso incluye cuatro etapas:

1. Se inicia con el estudio de los sistemas existentes del mismo tipo para entender cómo se usan en la práctica. Esto es importante, pues se trata de medir la fiabilidad como la ve un usuario del sistema. Su meta es definir un perfil operativo. Este último identifica clases de entradas del sistema y la probabilidad de que dichas entradas ocurran en uso normal.
2. Luego se construye un conjunto de datos de prueba que reflejan el perfil operativo, lo que significa que se crean datos de prueba con la misma distribución de probabilidad que los datos de prueba para los sistemas que ya se estudiaron. Normalmente, se usará un generador de datos de prueba para apoyar este proceso.
3. El sistema se prueba usando estos datos y el número de cuenta, así como el tipo de fallas que ocurren. También se registran los tiempos de dichas fallas. Como se estudió en el capítulo 12, las unidades de tiempo elegidas deben ser adecuadas para la métrica de fiabilidad que se usa.
4. Después de observar un número estadísticamente significativo de fallas, es posible calcular la fiabilidad del software y trabajar con el valor adecuado de métrica de fiabilidad.

Este proceso de cuatro pasos en ocasiones se conoce como "prueba estadística". La meta de la prueba estadística es valorar la fiabilidad del sistema. Esto contrasta con las pruebas de defectos, estudiadas en el capítulo 8, donde la meta es descubrir fallas en el desarrollo del sistema. Prowell y sus colaboradores (1999), en su libro sobre ingeniería de software Cleanroom, ofrecen una amplia descripción de las pruebas estadísticas.

Este enfoque conceptualmente atractivo para la medición de la fiabilidad no es fácil de aplicar en la práctica. Las principales dificultades que surgen son:

1. *Incertidumbre del perfil operativo* Los perfiles operativos basados en experiencia con otros sistemas tal vez no sean un reflejo preciso del uso real del sistema.
2. *Altos costos de la generación de datos de prueba* A menos que el proceso sea totalmente automatizado, será muy costoso generar un gran volumen de datos requeridos en un perfil operativo.

3. *Incertidumbre estadística cuando se especifica alta fiabilidad* Debe generarse un número estadísticamente significativo de fallas para permitir mediciones de fiabilidad precisas. Cuando el software ya es fiable, ocurren muy pocas fallas y es difícil generar nuevas fallas.

4. *Reconocimiento de la falla* No siempre es evidente si ocurrió o no una falla de sistema. Si se tiene una especificación formal, es posible identificar desviaciones de dicha especificación, pero si la especificación está en lenguaje natural, podría haber ambigüedades que signifiquen que los observadores no estén de acuerdo en torno al hecho de que haya fallado el sistema.

Con mucho, la mejor forma de generar el gran conjunto de datos requeridos para la medición de la fiabilidad es usar un generador de datos de prueba, el cual puede configurarse para producir automáticamente entradas que coincidan con el perfil operativo. Sin embargo, por lo regular no es posible automatizar la producción de todos los datos de prueba para sistemas interactivos, porque las entradas con frecuencia son una respuesta a salidas del sistema. Los conjuntos de datos para dichos sistemas tienen que generarse de forma manual, con los correspondientes costos más elevados. Incluso donde es posible la automatización completa, podría necesitarse una cantidad significativa de tiempo para escribir los comandos para el generador de datos de prueba.

Las pruebas estadísticas pueden usarse en conjunto con inyección de fallas en el desarrollo para reunir datos acerca de cuán efectivo ha sido el proceso de pruebas de defecto. La inyección de fallas en el desarrollo (Voas, 1997) es la introducción deliberada de errores en un programa. Cuando el programa se ejecuta, ello conduce a fallas de programa y fallas asociadas. Entonces se analizan las fallas para descubrir si la raíz de la causa es uno de los errores agregados al programa. Si se descubre que el X% de las fallas en el desarrollo inyectadas conduce a fallas en la operación, entonces los defensores de la inyección de fallas en el desarrollo argumentarán que esto sugiere que el proceso de pruebas de defecto también habrá descubierto el X% de las fallas en el desarrollo reales en el programa.

Desde luego, ello supone que la distribución y el tipo de fallas en el desarrollo inyectadas coinciden con las fallas en el desarrollo reales que surgen en la práctica. Es razonable pensar que esto es cierto para fallas en el desarrollo que se deben a errores de programación, pero la inyección de fallas no es efectiva para predecir el número de fallas en el desarrollo que surgen de errores de requerimientos o de diseño.

La prueba estadística revela con frecuencia errores en el software que no se descubrieron mediante otros procesos V&V. Tales errores pueden significar que la fiabilidad de un sistema no cubre los requerimientos y que deben hacerse reparaciones. Después de completar dichas reparaciones, el sistema se pone nuevamente a prueba para revalorar su fiabilidad. Luego de repetir muchas veces este proceso de reparación y de nuevas pruebas, es posible extrapolar los resultados y predecir cuándo se logrará cierto nivel requerido de fiabilidad. Esto requiere ajustar los datos extrapolados a un modelo de crecimiento de fiabilidad, que muestre cómo la fiabilidad tiende a mejorar con el tiempo, lo que ayuda con la planeación de pruebas. En ocasiones, un modelo de crecimiento revela que nunca se alcanzará un nivel de fiabilidad, de forma que deberán renegociarse los requerimientos.

15.2.1 Perfiles operativos

El perfil operativo de un sistema de software refleja cómo éste se usará en la práctica. Consta de una especificación de clases de entrada y su probabilidad de ocurrencia. Cuando un nuevo sistema de software sustituye a un sistema automatizado existente, es

Modelado de crecimiento de fiabilidad

Un modelo de crecimiento de fiabilidad es un modelo de cómo cambia con el tiempo la fiabilidad del sistema durante el proceso de pruebas. Conforme se descubren fallas del sistema, las fallas en el desarrollo subyacentes que causan dichas fallas en la operación se reparan, de manera que la fiabilidad del sistema debe mejorar durante las pruebas y la depuración del sistema. Para predecir la fiabilidad, el modelo conceptual de crecimiento de fiabilidad debe traducirse en ese caso en un modelo matemático.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/RGM.html

razonablemente sencillo valorar el patrón probable de uso del nuevo software. Debe corresponder al uso existente, con cierta concesión para la nueva funcionalidad que (presumiblemente) se incluye en el nuevo software. Por ejemplo, es posible especificar un perfil operativo para sistemas de conmutación telefónica, porque las compañías de telecomunicaciones conocen los patrones de llamadas que deben manejar dichos sistemas.

Normalmente, el perfil operativo es tal que las entradas con la mayor probabilidad de generarse caen en un número reducido de clases, como se muestra a la izquierda de la figura 15.4. Existe un gran número de clases en que las entradas son sumamente improbables, pero no imposibles. Éstas se muestran a la derecha de la figura 15.4. La elipsis (. . .) significa que existen muchas más entradas inusuales de las que se presentan.

Musa (1998) analiza el desarrollo de los perfiles operativos en los sistemas de telecomunicación. Como en dicho dominio hay un gran historial de recopilación de datos de uso, el proceso de desarrollo de perfil operativo es relativamente directo. De forma simple, refleja el historial de datos de uso. Para obtener un sistema que requirió casi 15 personas-año de esfuerzo, se desarrolló un perfil operativo en aproximadamente 1 persona-mes. En otros casos, la generación del perfil operativo tarda más tiempo (dos o tres personas-año), pero el costo se distribuye sobre algunas versiones del sistema. Musa considera que su compañía tenía al menos un rendimiento que rebasaba 10 veces la inversión requerida para el desarrollo de un perfil operativo.

Sin embargo, cuando un sistema de software es nuevo e innovador, resulta difícil anticipar cómo se usará. Por consiguiente, es prácticamente imposible crear un perfil operativo preciso. Numerosos y diferentes usuarios, con distintas expectativas, antecedentes y experiencia pueden usar el nuevo sistema. No hay un historial de uso de base de datos. Dichos usuarios pueden utilizar el sistema en formas que los desarrolladores del sistema no anticiparon.

Desde luego, es posible desarrollar un perfil operativo preciso para algunos tipos de sistema, tales como los sistemas de telecomunicación, que tienen un patrón estandarizado de uso. No obstante, para otro tipo de sistemas, existen muchos y diferentes usuarios, cada uno con su forma peculiar de usar el sistema. Como se estudió en el capítulo 10, diferentes usuarios tendrán impresiones muy disímiles de la fiabilidad, porque usan el sistema de distintas formas.

El problema se complica más porque los perfiles operativos no son estáticos, sino cambiantes conforme se usa el sistema. A medida que los usuarios aprenden sobre el nuevo sistema y se sienten más en confianza con él, comienzan a usarlo de formas más sofisticadas. Debido a esto, muchas veces es imposible desarrollar un perfil operativo confiable. En consecuencia, no es posible estar seguro de la precisión de algunas mediciones de fiabilidad, pues éstas podrían basarse en suposiciones incorrectas acerca de las formas en que se usa el sistema.

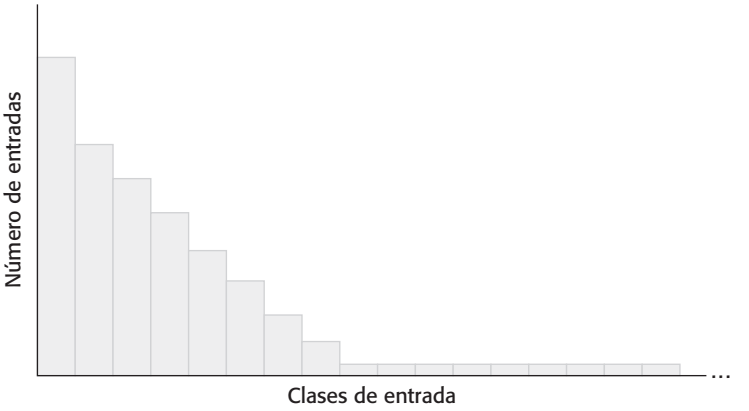


Figura 15.4 Un perfil operativo

15.3 Pruebas de seguridad

La valoración de la seguridad del sistema es cada vez más importante, pues más y más sistemas críticos se habilitan en Internet, por lo que cualquier persona con una conexión de red podría ingresar a ellos. Todos los días se presentan historias de ataques en sistemas basados en Web, y virus y gusanos se distribuyen regularmente mediante protocolos de Internet.

Todo esto significa que los procesos de verificación y validación para sistemas basados en Web deben enfocarse en la evaluación de la seguridad, en la que se pone a prueba la habilidad del sistema para resistir diferentes tipos de ataques. Sin embargo, como explica Anderson (2001), este tipo de valoración de la seguridad es muy difícil de realizar. Por ende, los sistemas se implementan a menudo con vacíos en la seguridad. Los atacantes aprovechan esos vacíos para lograr el acceso al sistema, o causar daño al sistema o a sus datos.

En esencia, existen dos razones que hacen muy difíciles las pruebas de seguridad:

1. Los requerimientos de seguridad, como algunos requerimientos de protección, se enuncian como "no debe". Esto es, especifican lo que no debe ocurrir, ya que es algo que se opone a la funcionalidad o al comportamiento requerido del sistema. Por regla general, no es posible definir este comportamiento no deseado como simples restricciones a comprobar por el sistema.

   Si hay recursos disponibles, es posible demostrar, al menos en principio, que un sistema cumple con sus requerimientos funcionales. Sin embargo, es imposible probar que un sistema no hace algo. Sin importar la cantidad de pruebas, las vulnerabilidades en la seguridad pueden permanecer en un sistema después de que se implementa. Desde luego, es posible generar requerimientos funcionales diseñados para proteger al sistema contra algunos tipos conocidos de ataques. No obstante, es imposible derivar requerimientos para tipos de ataques desconocidos o no anticipados. Incluso en sistemas que se hayan usado durante años, un atacante ingenioso es capaz de descubrir nuevas formas de ataque y penetrar a lo que se consideraba un sistema seguro.

2. Las personas que atacan un sistema son inteligentes y buscan activamente vulnerabilidades que puedan aprovechar. Quieren experimentar con el sistema y utilizan artilugios que se alejan mucho de la actividad y el uso normales del sistema. Por ejemplo, en un campo de apellido podrían ingresar 1,000 caracteres con una mezcla de letras, signos de puntuación y números. Más aún, una vez que encuentran una vulnerabilidad, podrían intercambiar información sobre ésta y aumentar el número de atacantes potenciales.

Los atacantes a menudo intentan descubrir las suposiciones que hacen los desarrolladores del sistema, para entonces contradecir dichas suposiciones y observar lo que sucede. Están en una posición para usar y explorar un sistema durante cierto periodo y analizarlo mediante herramientas de software para descubrir vulnerabilidades que puedan aprovechar. De hecho, es muy probable que tengan más tiempo para buscar vulnerabilidades que los ingenieros de pruebas del sistema, pues estos últimos también deben enfocarse en realizar las pruebas del sistema.

Por esta razón, el análisis estático es particularmente útil como herramienta de prueba de seguridad. Un análisis estático de un programa puede guiar rápidamente al equipo de prueba hacia áreas de un programa que incluyen errores y vulnerabilidades. Las anomalías reveladas en el análisis estático pueden corregirse directamente, o bien, ayudan a identificar pruebas necesarias para revelar si dichas anomalías representan en realidad un riesgo para el sistema.

Para comprobar la seguridad de un sistema, puede usarse una combinación de pruebas, análisis basado en herramientas y verificación formal:

1. *Pruebas basadas en la experiencia* En este caso, el sistema se analiza contra tipos de ataque que conoce el equipo de validación. Esto implica el desarrollo de casos de prueba o el examen del código fuente de un sistema. Por ejemplo, para comprobar que el sistema no es susceptible al bien conocido ataque de envenenamiento SQL, se prueba el sistema usando entradas que incluyan comandos SQL. Para comprobar que no ocurrirán errores de desbordamiento de buffer, se examinan todos los buffers de entrada para ver si el programa comprueba que las asignaciones a los elementos del buffer están dentro de los límites.

   Este tipo de validación se realiza habitualmente en conjunto con la validación basada en herramientas, donde estas últimas brindan información que ayuda a enfocar las pruebas del sistema. Pueden crearse listas de verificación de conocidos problemas de seguridad para auxiliar en el proceso. La figura 15.5 brinda algunos ejemplos de preguntas que ayudan a impulsar las pruebas basadas en la experiencia. En una lista de verificación de problemas de seguridad también deberían incluirse las comprobaciones acerca de si se siguieron los lineamientos de diseño y programación para seguridad (capítulo 14).

2. *Equipos tigre* Ésta es una forma de pruebas basadas en la experiencia en las que es posible apoyarse en experiencia externa al equipo de desarrollo para probar un sistema de aplicación. Se establece un "equipo tigre", al que se le impone el objetivo de violar la seguridad del sistema. Ellos simulan ataques al sistema y usan su ingenio para descubrir nuevas formas de comprometer la seguridad del sistema. Los miembros del equipo tigre deben tener experiencia previa con pruebas de seguridad y descubrir debilidades de seguridad en los sistemas.

3. *Pruebas basadas en herramientas* Para este método se usan varias herramientas de seguridad, tales como verificadores de contraseña que permiten analizar el sistema. Los verificadores de contraseñas detectan contraseñas inseguras, por ejemplo, los nombres comunes o las cadenas de letras consecutivas. Este enfoque en realidad

Lista de verificación de seguridad
1. ¿Todos los archivos que se crean en la aplicación tienen permisos adecuados de acceso? Los permisos erróneos de acceso pueden conducir a que usuarios no autorizados tengan acceso a esos archivos.
2. ¿El sistema termina automáticamente las sesiones de usuario luego de un periodo de inactividad? Las sesiones que se mantienen activas permiten el acceso no autorizado a través de una computadora sin atender.
3. Si el sistema está escrito en un lenguaje de programación sin comprobación de límites de arreglo, ¿existen situaciones donde pueda explotarse el desbordamiento de buffer? El desbordamiento de buffer permite que los atacantes envíen cadenas de código al sistema que luego se ejecutan.
4. Si se establecen contraseñas, ¿el sistema comprueba que éstas sean "fuertes"? Las contraseñas fuertes constan de letras, números y signos de puntuación mezclados, y no son entradas normales de diccionario. Resultan más difíciles de romper que las contraseñas simples.
5. ¿Las entradas del entorno del sistema siempre se comprueban contra una especificación de entradas? El procesamiento incorrecto de entradas mal formadas es una causa común de vulnerabilidades de seguridad.

Figura 15.5 Ejemplos de entradas en una lista de verificación de seguridad

es una extensión de la validación basada en la experiencia, donde la experiencia con las fallas de seguridad se concentra en las herramientas utilizadas. Desde luego, el análisis estático es otro tipo de prueba basada en herramientas.

4. *Verificación formal* Un sistema puede verificarse contra una especificación de seguridad formal. Sin embargo, como en otras áreas, la verificación formal de la seguridad no se usa de manera amplia.

Inevitablemente, las pruebas de la seguridad están limitadas por el tiempo y los recursos disponibles del equipo de pruebas. Esto significa que, por lo regular, es conveniente adoptar un enfoque basado en el riesgo para las pruebas de seguridad y enfocarse en lo que se consideran los riesgos más significativos que enfrenta el sistema. Si se dispone de un análisis de los riesgos de seguridad para el sistema, puede usarse para impulsar el proceso de pruebas. Así como las pruebas del sistema contra los requerimientos de seguridad se derivan de dichos riesgos, el equipo de pruebas también debería tratar de romper el sistema al adoptar enfoques alternativos que amenacen los activos del sistema.

Es muy difícil para los usuarios finales de un sistema verificar su seguridad. Por consiguiente, algunos órganos gubernamentales de Estados Unidos y Europa han establecido conjuntos de criterios de evaluación de la seguridad que los evaluadores especializados pueden revisar (Pfleeger y Pfleeger, 2007). Los proveedores de productos de software envían sus productos para que sean evaluados y certificados de acuerdo con dichos criterios. Por lo tanto, si usted tiene un requerimiento para un nivel particular de seguridad, es recomendable elegir un producto que se haya validado a dicho nivel. Sin embargo, en la práctica, esos criterios se usan principalmente en sistemas militares y, por el momento, aún no logran mucha aceptación comercial.

15.4 Aseguramiento del proceso

Como se estudió en el capítulo 13, la experiencia demuestra que los procesos confiables conducen a sistemas confiables. Esto es, si un proceso se basa en buenas prácticas de

Regulación del software

Los gobiernos crearon organismos reguladores para garantizar que la industria privada no se beneficie del incumplimiento de los estándares nacionales de protección y seguridad o de algunos otros. Existen organismos reguladores en muchas industrias, como la de energía nuclear, la de aviación y la banca. Conforme los sistemas de software se vuelven cada vez más importantes en la infraestructura fundamental de los países, dichos reguladores se ocupan cada vez más de los casos de protección y confiabilidad para sistemas de software.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Regulation.html

ingeniería de software, entonces es más probable que el producto de software resultante sea confiable. Desde luego, un buen proceso no garantiza la confiabilidad. Sin embargo, la evidencia de que se usa un proceso confiable aumenta la confianza global de que un sistema es confiable. El aseguramiento del proceso se ocupa tanto de la recolección de información sobre los procesos usados durante el desarrollo del sistema, como de los resultados de dichos procesos. Esta información proporciona evidencia del análisis, las revisiones y las pruebas que se realizaron durante el desarrollo del software.

El aseguramiento del proceso incluye dos aspectos:

1. ¿Se tienen los procesos correctos? ¿Los procesos de desarrollo del sistema usados en la organización incluyen controles adecuados y subprocesos V&V para el tipo de sistema a desarrollar?
2. ¿Los procesos se realizan de manera correcta? ¿La organización efectuó el trabajo de desarrollo como se definió en sus descripciones de proceso de software, y se obtuvieron los resultados definidos a partir de los procesos de software?

Las compañías que tienen amplia experiencia en la ingeniería de sistemas críticos hicieron evolucionar sus procesos para reflejar una buena práctica de verificación y validación. En algunos casos, esto implicó discusiones con el regulador externo para acordar qué procesos debían utilizarse. Aunque hay un considerable grado de variación en los procesos entre compañías, las actividades que se esperaría ver en los procesos de desarrollo de sistemas críticos incluyen gestión de requerimientos, administración del cambio y control de la configuración, modelado de sistemas, revisiones e inspecciones, planeación de pruebas y análisis de cobertura de pruebas. La noción de mejora del proceso, en la que se introducen e institucionalizan las buenas prácticas en los procesos, se estudia en el capítulo 26.

El otro aspecto del aseguramiento del proceso es comprobar que los procesos se realizaron de manera adecuada. Por lo regular, esto implica asegurar que los procesos se documenten adecuadamente, y verificar esta documentación del proceso. Por ejemplo, parte de un proceso confiable puede implicar inspecciones formales de programa. La documentación en cada inspección debe incluir listas de verificación para impulsar la inspección, una lista de las personas implicadas, los problemas identificados durante la inspección y las acciones requeridas.

En consecuencia, demostrar que se usó un proceso confiable incluye generar gran cantidad de evidencia documental sobre el proceso y el software a desarrollar. La necesidad de esta amplia documentación significa que los procesos ágiles rara vez se emplean

Concesión de licencias para ingenieros de software

En algunas áreas de ingeniería, los especialistas en la protección de sistemas deben ser ingenieros con licencia; los ingenieros sin experiencia y mal calificados no están autorizados para asumir la responsabilidad de la seguridad. Esto no se aplica actualmente a los ingenieros de software, aunque existe mucha controversia acerca de la concesión de licencias a los ingenieros de software en muchos lugares de Estados Unidos (Knight y Leveson, 2002). Sin embargo, futuros estándares de proceso para desarrollo de software crítico para la protección requerirán que los ingenieros encargados de estos proyectos de seguridad sean ingenieros certificados, con un nivel mínimo definido de capacitación y experiencia.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Licensing.html

en los sistemas donde se requiere la certificación de la protección o la confiabilidad. Los procesos ágiles se enfocan en el software en sí y argumentan (correctamente) que una gran cantidad de documentación de proceso nunca se usa en realidad después de que se produce. Sin embargo, es necesario crear actividades de proceso de evidencia y documentos al usar información de proceso como parte de un caso de protección o confiabilidad de sistema.

15.4.1 Procesos para garantizar la protección

La mayor parte del trabajo en el aseguramiento del proceso se realizó en el área de desarrollo de sistemas críticos de protección. Por dos razones, es importante que un proceso de desarrollo de sistemas críticos de protección incluya procesos V&V que concuerden con el análisis y la garantía de protección:

1. Como los accidentes son acontecimientos inusuales en los sistemas críticos, puede ser prácticamente imposible simularlos durante las pruebas de un sistema. No es conveniente apoyarse en pruebas extensas para reproducir las condiciones que podrían conducir a un accidente.
2. En ocasiones, como se estudió en el capítulo 12, los requerimientos de protección son requerimientos enunciados como “no debe”, que excluyen el comportamiento no protegido del sistema. Es imposible demostrar de manera concluyente, mediante las pruebas y otras acciones de validación, que se cumplen dichos requerimientos.

En todas las etapas del proceso de desarrollo de software deberían incluirse actividades específicas para garantizar la protección. Dichas actividades registran los análisis realizados y a la persona o personas responsables de dichos análisis. Las actividades para garantizar la protección que se incorporan en los procesos de software incluyen las siguientes:

1. Registro y monitorización de peligros que rastrean los riesgos a partir de análisis preliminares de peligros, mediante pruebas y validación del sistema.
2. Revisiones de protección, que se efectúan a lo largo del proceso de desarrollo.
3. Certificación de protección, en la que se legitima formalmente la protección de los componentes críticos. Esto implica un grupo externo al equipo de desarrollo del

sistema que examina la evidencia disponible y decide si un sistema o componente debe considerarse protegido o no antes de que esté disponible para utilizarse.

Con la finalidad de apoyar tales procesos para garantizar la protección, debe designarse a los ingenieros encargados de proteger el proyecto que tengan responsabilidad explícita para los aspectos de protección de un sistema. Esto significa que dichos profesionales serán responsables si ocurre una falla en el sistema relacionada con la protección. Ellos deben demostrar que las actividades para garantizar la protección se realizaron de manera adecuada.

Los ingenieros encargados de la protección trabajan con los responsables de calidad con el propósito de garantizar que se usó un sistema de administración de la configuración detallado para rastrear toda la documentación relacionada con la protección, y para mantener el paso con la documentación técnica asociada. Esto es esencial en todos los procesos confiables. No tiene mucho sentido mantener procedimientos de validación rigurosos si una falla de la administración de la configuración significa que se entrega el sistema equivocado al cliente. En los capítulos 24 y 25 se cubre el tema de la administración de la configuración y la calidad.

El proceso de análisis del peligro, que es parte esencial del desarrollo de sistemas críticos para la protección, es un ejemplo de un proceso para garantizar la protección. El análisis del peligro se ocupa de identificar los riesgos, su probabilidad de ocurrencia y la probabilidad de que cada uno de ellos conduzca a un accidente. Si existe un código de programa que compruebe y maneje cada peligro, entonces es posible argumentar que dichos peligros no derivarán en accidentes. Tales explicaciones pueden complementarse con argumentos de protección, como se estudia más adelante en este capítulo. Ahí donde se requiera certificación externa antes de usar un sistema (por ejemplo, en una aeronave), el hecho de que pueda documentarse este seguimiento por lo general es una condición de certificación.

El documento central de protección que debe elaborarse es la bitácora de peligro. Este documento proporciona evidencia de cómo se consideraron los peligros identificados durante el desarrollo del software. Esta bitácora de peligro se usa en cada etapa del proceso de desarrollo del software para documentar cómo dicha etapa toma en cuenta los peligros. En la figura 15.6 se muestra un ejemplo simplificado de una entrada de bitácora de peligro para el sistema de administración de insulina. Este formato documenta el proceso de análisis de peligro y muestra los requerimientos de diseño que se generaron durante el proceso. Tales requerimientos de diseño tienen la intención de garantizar que el sistema de control nunca administre una sobredosis de insulina mediante una bomba especial.

Como se muestra en la figura 15.6, los individuos que tienen responsabilidades en materia de protección deben identificarse de manera explícita. Esto es importante por dos razones:

1. Cuando se identifica a las personas, éstas aparecen como responsables de sus acciones. Esto significa que se verán inducidas a tener más cuidado, porque cualquier problema puede rastrearse hacia su trabajo.
2. En caso de un accidente, podría haber procedimientos legales o una demanda. Es importante identificar quién fue responsable de garantizar la protección, de forma que pueda rendir cuentas de sus acciones.

Bitácora de peligro					Página 4: Impresa 20.02.2009
Sistema: Sistema de bomba de insulina *Ingeniero de seguridad:* James Brown			*Archivo:* bomba de insulina/protección/Bitácora de peligro *Bitácora de la versión:* 1/3		
Peligro identificado	Sobredosis de insulina administrada a un paciente				
Identificado por	Jane Williams				
Clase de criticidad	1				
Riesgo identificado	Alto				
Árbol de fallas identificado	Sí	*Fecha*	24.01.07	*Ubicación*	Bitácora de peligro, Página 5
Creadores de árbol de fallas	Jane Williams y Bill Smith				
Árbol de fallas verificado	SÍ	*Fecha*	28.01.07	*Verificador*	James Brown

Requerimientos de diseño de protección del sistema

1. El sistema debe incluir software de autocomprobación que probará el sistema de sensores, el reloj y el sistema de administración de insulina.
2. El software de autocomprobación debe ejecutarse una vez por minuto.
3. En caso de que el software de autocomprobación descubra una falla de desarrollo en alguno de los componentes del sistema, debe emitirse una advertencia audible y la pantalla de la bomba debe indicar el nombre del componente donde se descubrió la falla. Debe suspenderse la administración de insulina.
4. El sistema debe incorporar un sistema de modificación manual que permita al usuario del sistema modificar la dosis calculada de insulina que debe administrar el sistema.
5. La cantidad de modificación no debe ser mayor a un valor preestablecido (maxOverride), que se determina cuando el personal médico configura el sistema.

Figura 15.6 Entrada simplificada de una bitácora de peligro

15.5 Casos de protección y confiabilidad

Los procesos para garantizar la seguridad y la confiabilidad generan mucha información. Ésta incluye resultados de pruebas, información sobre los procesos de desarrollo utilizados, registros de reuniones de revisión, etcétera. Tal información proporciona evidencia de la seguridad y confiabilidad de un sistema, y se usa para ayudar a decidir si el sistema es suficientemente confiable para su uso operacional.

Los casos de protección y confiabilidad son documentos estructurados que establecen argumentos y evidencias detallados de que un sistema está protegido, o que se logró un nivel requerido de seguridad o confiabilidad. Algunas veces se llaman casos de aseguramiento. En esencia, un caso de protección o confiabilidad reúne toda la evidencia disponible que indica que un sistema es confiable. Para muchos tipos de sistemas críticos, la producción de un caso de protección es un requerimiento legal. El caso debe satisfacer a un órgano regulador o de certificación antes de que el sistema pueda implementarse.

La responsabilidad del organismo regulador radica en comprobar que un sistema completado es tan seguro o confiable como ejecutable, de manera que su papel principal entra en juego cuando se completa un proyecto de desarrollo. Sin embargo, los reguladores y

desarrolladores rara vez trabajan en aislamiento; se comunican con el equipo de desarrollo para establecer lo que debe incluirse en el caso de protección. El regulador y los desarrolladores, en conjunto, examinan los procesos y procedimientos para asegurarse de que se realicen y documenten a satisfacción del organismo regulador.

Los casos de confiabilidad se diseñan por lo general durante y después del proceso de desarrollo del sistema. En ocasiones esto puede causar problemas si las actividades del proceso de desarrollo no producen evidencia de la confiabilidad del sistema. Graydon y sus colaboradores (2007) argumentan que el desarrollo de un caso de protección y confiabilidad debe integrarse estrechamente con el diseño y la implementación del sistema. Esto significa que las decisiones acerca del diseño del sistema pueden estar influidas por los requerimientos del caso de confiabilidad. Es posible evitar las opciones de diseño que pueden aumentar significativamente las dificultades y los costos del desarrollo del caso.

Los casos de confiabilidad son generalizaciones de casos de seguridad del sistema. Un caso de protección es un conjunto de documentos que incluyen una descripción del sistema a certificar, información sobre los procesos empleados para desarrollar el sistema y, de manera esencial, argumentos lógicos que demuestren que es probable que el sistema esté protegido. De manera más sucinta, Bishop y Bloomfield (1998) definen un caso de protección como:

> *Una recopilación de la evidencia documentada que ofrece un argumento convincente y válido de que un sistema está adecuadamente protegido para una aplicación dada en un entorno determinado.*

La organización y los contenidos de un caso de protección o confiabilidad dependen del tipo de sistema que deba certificarse y su contexto de operación. La figura 15.7 muestra una posible estructura para un caso de protección, pero en esta área no hay estándares industriales que se utilicen ampliamente para los casos de protección. Las estructuras de casos de protección varían, dependiendo de la industria y la madurez del dominio. Por ejemplo, los casos de protección nuclear se han requerido durante muchos años. Son muy completos y se presentan de una forma que es familiar para los ingenieros nucleares. Sin embargo, los casos de protección para dispositivos médicos se introdujeron mucho más recientemente. Su estructura es más flexible y suelen estar menos detallados que los casos nucleares.

Desde luego, el software en sí no es peligroso. Sólo cuando se incrusta en un gran sistema basado en computadora o sociotécnico en que las fallas de software pueden generar fallas de otro equipo o en procesos que pueden causar lesión o muerte. Por lo tanto, un caso de protección de software siempre es parte de un caso de protección de sistema más amplio que demuestra la protección del sistema global. Cuando elabore un caso de protección de software, deberá relacionar las fallas del software con fallas de sistema más amplias y demostrar que dichas fallas de software no ocurrirán, o que no se propagarán de tal forma que generen fallas peligrosas del sistema.

15.5.1 Argumentos estructurados

La decisión de si un sistema es o no suficientemente confiable para emplearse debe basarse en argumentos lógicos. Dichos argumentos deben demostrar que la evidencia presentada apoya las afirmaciones sobre la seguridad y confiabilidad de un sistema. Tales afirmaciones pueden ser absolutas (el evento X ocurrirá o no ocurrirá) o probabilísticas (la probabilidad de ocurrencia del evento Y es 0.n). Un argumento vincula la evidencia y la afirmación.

Capítulo	Descripción
Descripción del sistema	Un panorama del sistema y una descripción de sus componentes críticos.
Requerimientos de protección	Los requerimientos de protección extraídos de la especificación de requerimientos del sistema. También pueden incluirse detalles de otros requerimientos relevantes de sistema.
Análisis de peligro y riesgo	Documentos que describen los peligros y riesgos que se identificaron, y las medidas tomadas para reducir el riesgo. Análisis de peligro y bitácoras de peligro.
Análisis de diseño	Conjunto de argumentos estructurados (véase la sección 15.5.1) que justifican por qué es seguro el diseño.
Verificación y validación	Descripción de los procedimientos V&V utilizados y, en su caso, los planes de pruebas para el sistema. Resúmenes de los resultados de las pruebas que muestran defectos detectados y corregidos. Si se usaron métodos formales, una especificación formal del sistema y cualquier análisis de dicha especificación. Registros de análisis estáticos del código fuente.
Reportes de revisión	Registros de todas las revisiones de diseño y protección.
Competencias de equipo	Evidencia de la competencia de todos los miembros del equipo implicados en el desarrollo y la validación de los sistemas relacionados con la protección.
Proceso QA	Registros de los procesos de aseguramiento de la calidad (véase el capítulo 24) realizados durante el desarrollo del sistema.
Procesos de administración del cambio	Registros de todos los cambios propuestos, acciones tomadas y, en su caso, justificación de la protección de tales cambios. Información acerca de procedimientos de administración de la configuración y bitácoras de administración de la configuración.
Casos de protección asociados	Referencias a otros casos de protección que puedan repercutir en el caso de protección.

Figura 15.7 Contenido de un caso de protección de software

Como se muestra en la figura 15.8, un argumento es una relación entre lo que se piensa que es el caso (la afirmación) y un cuerpo de evidencia que se recolectó. El argumento, en esencia, explica por qué la afirmación, que es una aseveración acerca de la seguridad o confiabilidad del sistema, puede inferirse a partir de la evidencia disponible.

Por ejemplo, la bomba de insulina es un dispositivo esencial para la protección, cuya falla podría causar lesiones a un usuario. En muchos países, esto significa que una autoridad reguladora (en el Reino Unido, el Medical Devices Directorate) debe cerciorarse de la protección del sistema antes de que el dispositivo pueda venderse y usarse. Para tomar esta decisión, el organismo regulador valora el caso de protección del sistema, que presenta argumentos estructurados de que la operación normal del sistema no causará daño a un usuario.

Los casos de protección se apoyan por lo general en argumentos estructurados basados en afirmaciones. Por ejemplo, el siguiente argumento puede usarse para justificar una afirmación de que los cálculos realizados por el software de control no conducirán a la administración de una sobredosis de insulina a un usuario del dispositivo. Desde luego, éste es un argumento muy simplificado. En un caso de protección real más detallado, se presentarían informes sobre la evidencia.

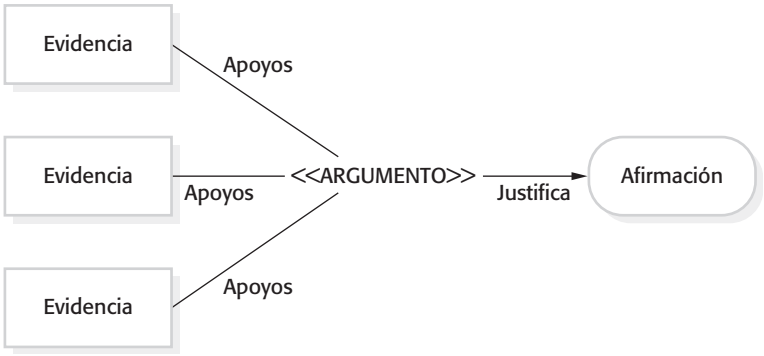


Figura 15.8 Argumentos estructurados

Afirmación: La máxima dosis individual calculada por la bomba de insulina no superará maxDose, donde maxDose se valoró como una dosis individual segura para un paciente particular.

Evidencia: Argumento de protección para el programa de control del software de la bomba de insulina (más adelante en esta sección se analizan los argumentos de protección).

Evidencia: Conjuntos de datos de prueba para la bomba de insulina. En 400 pruebas, el valor de currentDose se calculó correctamente y nunca superó maxDose.

Evidencia: Reporte de análisis estático para el programa de control de la bomba de insulina. El análisis estático del software de control no reveló anomalías que afectaran el valor de currentDose, la variable del programa que contiene la dosis de insulina a administrar.

Argumento: La evidencia presentada indica que la máxima dosis de insulina que puede calcularse es igual a maxDose.

Por lo tanto, es razonable suponer, con un alto nivel de confianza, que la evidencia justifica la afirmación de que la bomba de insulina no calcularía una dosis de insulina a administrar que supere la máxima dosis individual.

Observe que la evidencia presentada es tanto redundante como diversa. El software se verifica mediante numerosos y diferentes mecanismos con significativo traslape entre ellos. Como se estudió en el capítulo 13, el uso de procesos redundantes y diversos aumenta la confianza. Si hay omisiones y errores que no se detectan por medio de un proceso de validación, hay una buena oportunidad de que se encuentren por uno de los otros.

Por supuesto, normalmente habrá muchas afirmaciones acerca de la confiabilidad y seguridad de un sistema, en que la validez de una afirmación depende con frecuencia de que otras afirmaciones sean válidas o no. Por consiguiente, las afirmaciones pueden organizarse en una jerarquía. La figura 15.9 señala parte de esa jerarquía de afirmación para la bomba de insulina. Con la finalidad de demostrar que es válida una afirmación de alto nivel, primero se debe trabajar por medio de los argumentos para afirmaciones de nivel más bajo. Si es posible probar que cada una de tales afirmaciones de nivel inferior está justificada, entonces es posible inferir que están justificadas las afirmaciones de nivel superior.

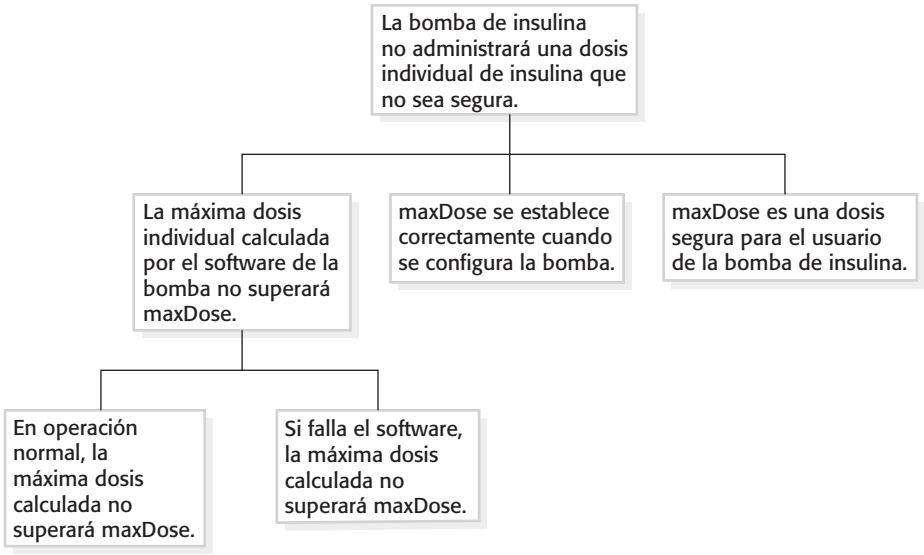


Figura 15.9 Jerarquía de afirmaciones de protección para la bomba de insulina

15.5.2 Argumentos estructurados de protección

Los argumentos estructurados de protección son un tipo de argumento estructurado, que señala que un programa cumple sus obligaciones de protección. En un argumento de protección, no es necesario probar que el programa funciona como se pretende. Sólo es necesario demostrar que la ejecución del programa no derivará en un estado inseguro. Esto significa que los argumentos de protección son menos costosos de elaborar que los argumentos de corrección. No deben considerarse todos los estados del programa, simplemente habrá que concentrarse en estados que podrían conducir a un accidente.

Una suposición general que subyace en el trabajo referente a la protección del sistema es que el número de fallas en el desarrollo del sistema que pueden conducir a peligros críticos para la protección es considerablemente menor que el número total de fallas en el desarrollo que pueden existir en el sistema. La garantía de protección puede concentrarse en aquellas fallas en el desarrollo con peligro potencial. Si es posible probar que dichas fallas no ocurrirán o que, si ocurren, el peligro asociado no causará un accidente, entonces el sistema es seguro. Ésta es la base de los argumentos estructurados de protección.

Los argumentos estructurados de protección tienen la intención de demostrar que, en condiciones de ejecución normales, un programa debería ser seguro. Por lo general, se basan en contradicción. Los pasos recomendados para elaborar un argumento de protección son los siguientes:

1. Comience por suponer que un estado inseguro, el cual se identificó mediante el análisis de peligro del sistema, puede alcanzarse al ejecutar el programa.
2. Escriba un predicado (una expresión lógica) que defina ese estado inseguro.
3. Luego analice metódicamente un modelo del sistema o el programa, y pruebe que, para todas las rutas del programa que conduzcan a dicho estado, la condición de finalización de dichas rutas contradice el predicado de estado no protegido. Si éste es el caso, es incorrecta la suposición inicial de un estado inseguro.

```
— La dosis de insulina a administrar es función del
— nivel de azúcar en la sangre, la dosis previa administrada y
— el momento de administración de la dosis previa
currentDose = computeInsulin () ;

// Comprobación de protección - ajustar currentDose si es necesario.

// enunciado if 1
if (previousDose == 0)
{
   if (currentDose > maxDose/2)
      currentDose = maxDose/2 ;
}
else
   if (currentDose > (previousDose * 2) )
      currentDose = previousDose * 2 ;

// enunciado if 2

if ( currentDose < minimumDose )
   currentDose = 0 ;
else if ( currentDose > maxDose )
   currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Figura 15.10 Cálculo de dosis de insulina mediante comprobaciones de protección

4. Cuando se repite este análisis para todos los peligros identificados, entonces existe clara evidencia de que el sistema es seguro.

Los argumentos estructurados de protección pueden aplicarse a diferentes niveles, desde los requerimientos a través de los modelos de diseño hasta el código. En el nivel de requerimientos, se trata de demostrar que no hay requerimientos de protección faltantes, y que los requerimientos no hacen suposiciones inválidas acerca del sistema. A nivel del diseño, es conveniente analizar un modelo de estado del sistema para encontrar estados inseguros. A nivel de código, considere todas las rutas con el código crítico de protección para demostrar que la ejecución de todas las rutas conduce a una contradicción.

Como ejemplo, considere el código de la figura 15.10, que puede ser parte de la implementación del sistema de administración de insulina. El código calcula la dosis de insulina a administrar, luego aplica algunas comprobaciones de protección para reducir la probabilidad de que se inyecte una sobredosis de insulina. Desarrollar un argumento de protección para este código implica probar que la dosis de insulina administrada nunca será mayor que el máximo nivel seguro para una dosis individual. Esto se establece para cada usuario diabético en el tratamiento con sus consejeros médicos.

Para demostrar que existe protección, no se tiene que probar que el sistema administra la dosis “correcta”, simplemente que nunca administra una sobredosis al paciente. Se trabaja sobre la suposición de que maxDose es el nivel protegido para dicho usuario del sistema.

Para elaborar el argumento de protección, identifique el predicado que defina el estado inseguro, que es currentDose > maxDose. Luego demuestre que todas las rutas del programa conducen a una contradicción de esta afirmación. Si éste es el caso, la condición

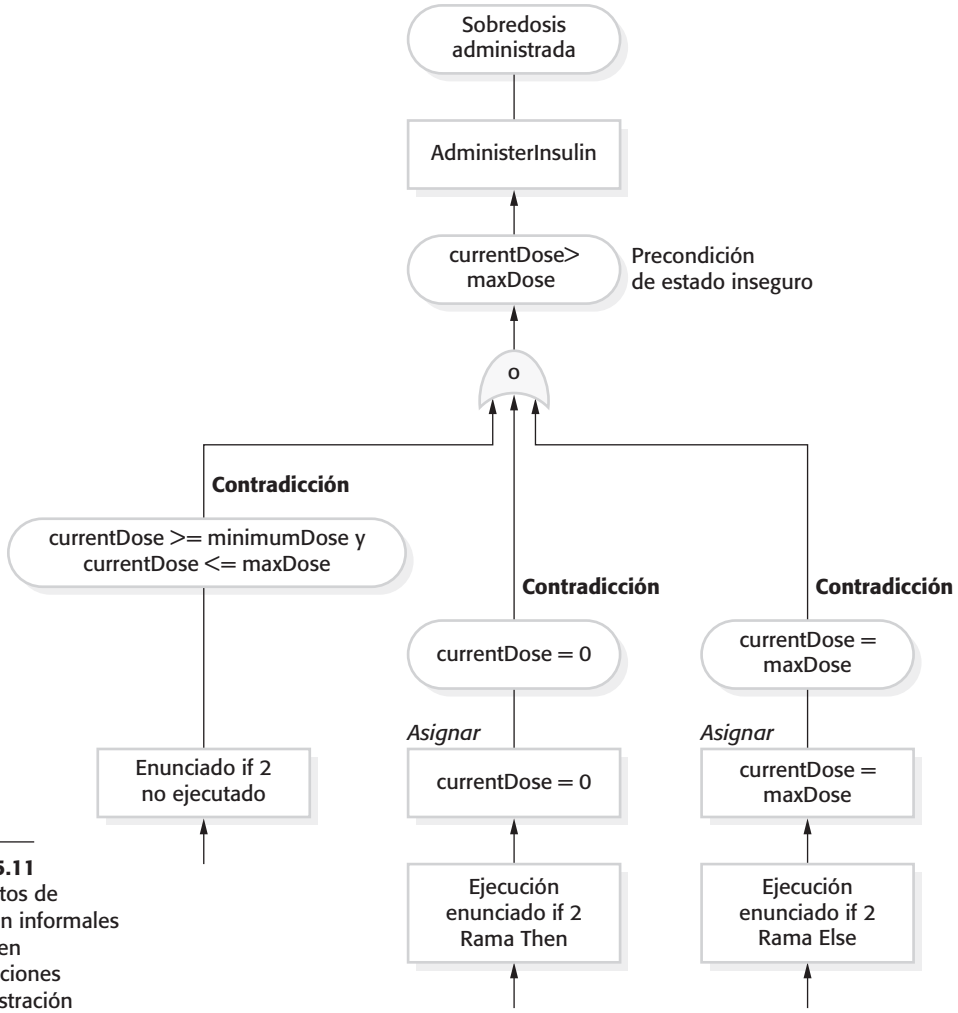


Figura 15.11 Argumentos de protección informales basados en contradicciones de demostración

insegura no puede ser verdadera. Si es posible hacer esto, podemos estar seguros de que el programa no calculará una dosis inadecuada de insulina. En la figura 15.11 se estructuran y presentan gráficamente los argumentos de protección.

Para elaborar un argumento estructurado de un programa que no efectúa un cálculo inseguro, primero identifique todas las posibles rutas a lo largo del código que podrían conducir al estado potencialmente no protegido. Trabaje hacia atrás a partir de ese estado no protegido y considere la última asignación a todas las variables de estado en cada ruta que conduzca a dicho estado no protegido. Si es posible probar que ninguno de los valores de esas variables es inseguro, entonces se demostrará que la suposición inicial (de que el cálculo es inseguro) es incorrecta.

Trabajar hacia atrás es importante porque significa que usted puede ignorar todos los estados, además de los estados finales, que conduzcan a la condición de salida para el código. Los valores anteriores no son importantes para la protección del sistema. En este ejemplo, sólo necesita preocuparse por el conjunto de posibles valores de currentDose

inmediatamente antes de que se ejecute el método administerInsulin. Puede ignorar los cálculos, como el enunciado if 1 de la figura 15.10, en el argumento de protección, porque sus resultados están sobrescritos en posteriores enunciados del programa.

En el argumento de protección que se muestra en la figura 15.11, existen tres posibles rutas del programa que conducen a la solicitud del método administerInsulin. Debe demostrarse que la cantidad de insulina administrada nunca supera maxDose. Considere todas las posibles rutas del programa hacia administerDose:

1. Ninguna rama del enunciado if 2 se ejecuta. Esto sólo puede ocurrir si currentDose es mayor que o igual a minimumdose y menor que o igual a maxDose. Ésta es la post-condición: una afirmación que es verdadera después de que se ejecuta el enunciado.
2. Se ejecuta la rama then del enunciado if 2. En este caso, se ejecuta la asignación establecer currentDose a cero. Por lo tanto, su post-condición es currentDose = 0.
3. Se ejecuta la rama else-if del enunciado if 2. En este caso, se ejecuta la asignación establecer currentDose a maxDose. Por lo tanto, después de ejecutar este enunciado, se sabe que la post-condición es currentDose = maxDose.

En los tres casos, las post-condiciones contradicen la precondición insegura de que la dosis administrada es mayor que maxDose. Por consiguiente, puede afirmarse que el cálculo es seguro.

Los argumentos estructurados se usan en la misma forma para demostrar que ciertas propiedades de seguridad de un sistema son verdaderas. Por ejemplo, si usted quiere demostrar que un cálculo nunca conducirá al cambio de permisos sobre un recurso, podrá usar un argumento de seguridad estructurado para comprobarlo. Sin embargo, la evidencia de los argumentos estructurados es menos confiable para la validación de seguridad. Se debe a que hay una posibilidad de que el atacante logre corromper el código del sistema. En tal caso, el código ejecutado no es el código que usted consideró como seguro.

PUNTOS CLAVE

- El análisis estático es un enfoque de V&V que examina el código fuente (u otra presentación) de un sistema, y que busca errores y anomalías. Permite que todas las partes de un programa se verifiquen, no sólo aquellas que se ejercitan mediante pruebas del sistema.
- La comprobación del modelo es un enfoque formal al análisis estático que comprueba exhaustivamente todos los estados en un sistema en busca de errores potenciales.
- Las pruebas estadísticas se usan para evaluar la fiabilidad del software. Se apoyan en probar el sistema con un conjunto de datos de prueba que reflejen el perfil operativo del software. Los datos de prueba pueden generarse automáticamente.

- La validación de la seguridad es difícil porque los requerimientos de seguridad establecen lo que no debe ocurrir en un sistema, en lugar de lo que debe ocurrir. Más aún, los atacantes del sistema son inteligentes y, para sondear las debilidades, dedican más tiempo que el que se destina a las pruebas de seguridad.
- La validación de la seguridad puede realizarse mediante análisis basados en la experiencia, análisis basados en herramientas o "equipos tigre", que simulan ataques a un sistema.
- Es importante tener un proceso certificado bien definido para el desarrollo de sistemas críticos de protección. El proceso debe incluir la identificación y la monitorización de peligros potenciales.
- Los casos de protección y confiabilidad reúnen toda la evidencia que demuestre que un sistema es seguro y confiable. Los casos de protección se requieren cuando un regulador externo debe certificar el sistema antes de su uso.
- Por lo general, los casos de protección se basan en argumentos estructurados. Los argumentos estructurados de protección demuestran que una condición peligrosa identificada nunca ocurrirá, al considerar todas las rutas del programa que conduzcan a una condición no protegida y al probar que la condición es insostenible.

LECTURAS SUGERIDAS

Software Reliability Engineering: More Reliable Software, Faster and Cheaper, 2nd edition. Probablemente este libro sea clave para el uso de perfiles operativos y modelos de fiabilidad para la valoración de la fiabilidad. Incluye detalles de experiencias con pruebas estadísticas. (J. D. Musa, McGraw-Hill, 2004.)

"NASA's Mission Reliable". Una explicación de cómo la NASA usa el análisis estático y la comprobación de modelos para asegurar la fiabilidad del software de sus naves espaciales. (P. Regan y S. Hamilton, *IEEE Computer*, **37** (1), enero de 2004.) http://dx.doi.org/10.1109/MC.2004.1260727.

Dependability cases. Introducción basada en ejemplos a la definición de un caso de confiabilidad. (C. B. Weinstock, J. B. Goodenough, J. J. Hudak, Software Engineering Institute, CMU/SEI-2004-TN-016, 2004.) http://www.sei.cmu.edu/publications/documents/04.reports/04tn016.html.

How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. Un libro breve que ofrece buenos y prácticos consejos sobre cómo correr pruebas de seguridad en aplicaciones en red. (M. Andrews y J. A. Whittaker, Addison-Wesley, 2006.)

"Using static analysis to find bugs". Este ensayo describe a Findbugs, un analizador estático Java que utiliza técnicas sencillas para encontrar violaciones potenciales a la seguridad y errores en tiempo de operación. (N. Ayewah et al., *IEEE Software*, **25** (5), Sept/Oct 2008.) http://dx.doi.org/10.1109/MS.2008.130.

EJERCICIOS

15.1. Explique cuándo resulta efectivo en términos de costos usar la especificación y la verificación formales en el desarrollo de sistemas de software críticos de protección. ¿Por qué considera que los ingenieros de sistemas críticos están en contra del uso de los métodos formales?

15.2. Sugiera una lista de condiciones que pudieran detectarse mediante un analizador estático para Java, C++ o cualquier otro lenguaje de programación que utilice. Comente acerca de esta lista en comparación con la lista presentada en la figura 15.2.

15.3. Explique por qué es prácticamente imposible validar las especificaciones de fiabilidad cuando se expresan en términos de un número muy pequeño de fallas durante la vida total de un sistema.

15.4. Mencione por qué asegurar la fiabilidad de un sistema no es una garantía de protección.

15.5. Con ejemplos, explique por qué las pruebas de seguridad son un proceso muy difícil.

15.6. Sugiera cómo validaría un sistema de protección de contraseñas para una aplicación que desarrolló. Explique la función de cualquier herramienta que considere útil.

15.7. El MHC-PMS tiene que asegurarse contra ataques que puedan revelar información confidencial de los pacientes. Algunos de esos ataques se analizaron en el capítulo 14. Con esa información, extienda la lista de verificación de la figura 15.5 para guiar a los examinadores del MHC-PMS.

15.8. Mencione cuatro tipos de sistemas que requieran casos de protección de software y exponga por qué se requieren.

15.9. El mecanismo de control de cerrado de puerta en una instalación de almacenamiento de desechos nucleares está diseñado para una operación protegida. Garantiza que la entrada al almacén sólo se permita cuando se colocan los escudos de radiación o cuando el nivel de radiación en el lugar cae por debajo de cierto valor dado (dangerLevel). De este modo:

i. Si los escudos de radiación de control remoto se encuentran en su lugar dentro del recinto, un operador autorizado podrá abrir la puerta.

ii. Si el nivel de radiación en el recinto está por debajo de un valor especificado, un operador autorizado podrá abrir la puerta.

iii. Un operador autorizado se identifica mediante el ingreso de un código de entrada autorizado.

El código que se muestra en la figura 15.12 (véase a continuación) controla el mecanismo de cerrado de la puerta. Observe que el estado seguro es que la entrada no se permite. Con el enfoque que se explicó en la sección 15.5.2, desarrolle un argumento de protección para este código. Use los números de línea para referirse a enunciados específicos. Si encuentra que el código no está protegido, sugiera cómo debe modificarse para protegerlo.

```
1       entryCode = lock.getEntryCode () ;
2       if (entryCode == lock.authorizedCode)
3       {
4             shieldStatus = Shield.getStatus ();
5             radiationLevel = RadSensor.get ();
6             if (radiationLevel < dangerLevel)
7                   state = safe;
8             else
9                   state = unsafe;
10            if (shieldStatus == Shield.inPlace() )
11                  state = safe;
12            if (state == safe)
13                  {
14                        Door.locked = false ;
15                        Door.unlock ();
16                  }
17            else
18            {
19                  Door.lock ( );
20                  Door.locked := true ;
21            }
22      }
```

Figura 15.12 Código de cerrado de la puerta

15.10. Suponga que usted es parte de un equipo que desarrolló el software para una planta química, el cual falló y causó un serio problema de contaminación. A su jefe lo entrevistan en televisión y él afirma que el proceso de validación fue exhaustivo y que no hubo fallas en el desarrollo del software. Asegura que los problemas pueden deberse a procedimientos operacionales erróneos. Un periodista acude a usted para recabar su opinión. Explique cómo manejaría la entrevista.

REFERENCIAS

Abrial, J. R. (2005). *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Chichester, UK: John Wiley & Sons.

Baier, C. y Katoen, J.-P. (2008). *Principles of Model Checking*. Cambridge, Mass.: MIT Press.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., S. K., R. y Ustuner, A. (2006). "Thorough Static Analysis of Device Drivers". *Proc. EuroSys* 2006, Leuven, Bélgica.

Bishop, P. y Bloomfield, R. E. (1998). "A methodology for safety case development". *Proc. Safety-critical Systems Symposium*, Birmingham, UK: Springer.

Chandra, S., Godefroid, P. y Palm, C. (2002). "Software model checking in practice: An industrial case study". *Proc. 24th Int. Conf. on Software Eng*. (ICSE 2002), Orland, Fla.: IEEE Computer Society, 431–41.

Croxford, M. y Sutton, J. (2006). "Breaking Through the V and V Bottleneck". *Proc. 2nd Int. Eurospace—Ada-Europe Symposium on Ada in Europe*, Frankfurt, Alemania: Springer-LNCS, 344–54.

Evans, D. y Larochelle, D. (2002). "Improving Security Using Extensible Lightweight Static Analysis". *IEEE Software*, **19** (1), 42–51.

Graydon, P. J., Knight, J. C. y Strunk, E. A. (2007). "Assurance Based Development of Critical Systems". *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, Edimburgo, Escocia: 347–57.

Holzmann, G. J. (2003). *The SPIN Model Checker*. Boston: Addison-Wesley.

Knight, J. C. y Leveson, N. G. (2002). "Should software engineers be licensed?" *Comm*. ACM, **45** (11), 87–90.

Larus, J. R., Ball, T., Das, M., Deline, R., Fahndrich, M., Pincus, J., Rajamani, S. K. y Venkatapathy, R. (2003). "Righting Software". *IEEE Software*, **21** (3), 92–100.

Musa, J. D. (1998). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. Nueva York: McGraw-Hill.

Nguyen, T. y Ourghanlian, A. (2003). "Dependability assessment of safety-critical system software by static analysis methods". *Proc. IEEE Conf. on Dependable Systems and Networks (DSN' 2003)*, San Francisco, Calif.: IEEE Computer Society, 75–9.

Pfleeger, C. P. y Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.

Prowell, S. J., Trammell, C. J., Linger, R. C. y Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.

Regan, P. y Hamilton, S. (2004). "NASA's Mission Reliable". *IEEE Computer*, **37** (1), 59–68.

Schneider, S. (1999). *Concurrent and Real-time Systems: The CSP Approach*. Chichester, UK: John Wiley and Sons.

Visser, W., Havelund, K., Brat, G., Park, S. y Lerda, F. (2003). "Model Checking Programs". *Automated Software Engineering J*., **10** (2), 203–32.

Voas, J. (1997). "Fault Injection for the Masses". *IEEE Computer*, **30** (12), 129–30.

Wordsworth, J. (1996). *Software Engineering with* B. Wokingham: Addison-Wesley.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. y Vouk, M. A. (2006). "On the value of static analysis for fault detection in software". *IEEE Trans. on Software Eng*., **32** (4), 240–5.

PARTE 3

Ingeniería de software avanzada

Esta parte del libro se titula "Ingeniería de software avanzada" porque el lector debe comprender las bases de la disciplina, estudiadas en los capítulos 1 a 9, para enriquecerse de la información que aquí se expone. Muchos de los temas examinados reflejan la práctica de la ingeniería de software industrial en el desarrollo de sistemas distribuidos y de tiempo real.

La reutilización de software se ha convertido ahora en el paradigma de desarrollo dominante para los sistemas de información basados en Web y los sistemas empresariales. El enfoque más común es la reutilización COTS, en la que un gran sistema se configura para las necesidades de una organización y en la que se requiere poco o nada de desarrollo de software original. El capítulo 16 se introduce en el tema general de reutilización y se enfoca en la reutilización de sistemas COTS.

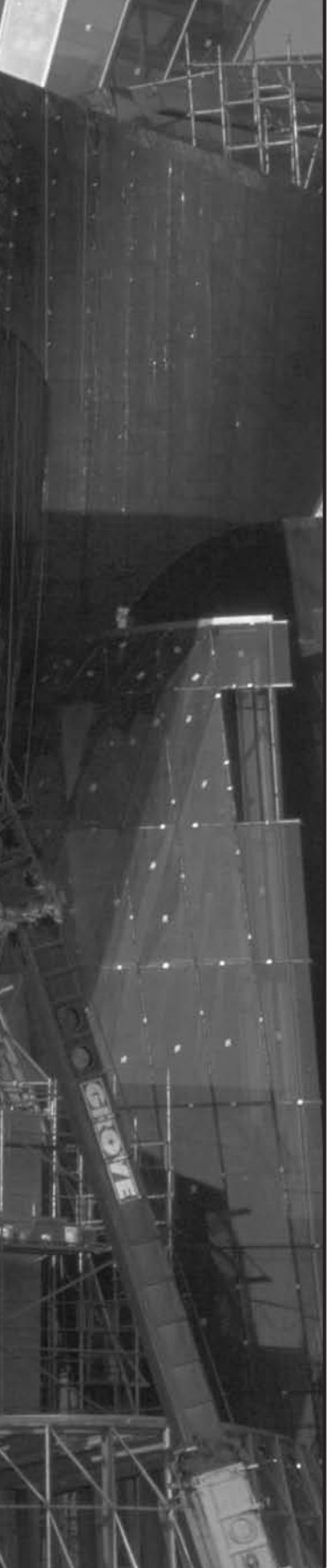
El capítulo 17 se refiere también a la reutilización de componentes de software en vez de sistemas de software completos. La ingeniería de software basada en componentes es un proceso de configuración, donde se desarrolla un nuevo código para integrar los componentes reutilizables. En este capítulo se explica lo que se entiende por componente y por qué se necesitan modelos de componente estándar para la reutilización efectiva de componentes. Además, se analiza el proceso general de la ingeniería de software basada en componentes y los problemas de la composición de componentes.

La mayoría de los grandes sistemas son actualmente sistemas distribuidos, y el capítulo 18 se ocupa de los conflictos y problemas de construir sistemas distribuidos. Se introduce el enfoque cliente-servidor como un paradigma fundamental de la ingeniería de sistemas distribuidos, y se exponen varias formas de implementar este estilo arquitectónico. La sección final de este capítulo refiere cómo el hecho de proporcionar software, para un servicio de aplicación distribuido, cambiará de forma radical el mercado de los productos de software.

El capítulo 19 introduce el tema relacionado de las arquitecturas orientadas al servicio, que vinculan las nociones de distribución y reutilización. Los servicios son componentes de software de reutilización, cuya funcionalidad está disponible a través de Internet para una amplia gama de clientes. En este capítulo se explica qué implica la creación de servicios (ingeniería de servicios) y la composición de servicios para elaborar nuevos sistemas de software.

Los sistemas embebidos son los ejemplos de sistemas de software usados más ampliamente, y el capítulo 20 se ocupa de este importante tema. Presenta el concepto de un sistema embebido en tiempo real y describe tres patrones arquitectónicos utilizados en el diseño de sistemas embebidos. Más adelante, detalla el proceso de análisis de temporización (*timing*) y concluye con un estudio de los sistemas operativos en tiempo real.

Finalmente, el capítulo 21 versa sobre el desarrollo de software orientado a aspectos (AOSD, por las siglas de *Aspect-Oriented Software Development*). El AOSD se relaciona también con la reutilización y propone un nuevo enfoque, basado en aspectos, para organizar y estructurar sistemas de software. Aun cuando todavía no es una corriente principal en la ingeniería de software, el AOSD tiene el potencial de mejorar significativamente los enfoques actuales para la implementación de software.



16

Reutilización de software

Objetivos

Los objetivos de este capítulo son introducir al lector en la reutilización de software y describir los enfoques para el desarrollo de sistemas basados en la reutilización de sistemas a gran escala. Al estudiar este capítulo:

- comprenderá los beneficios y problemas de la reutilización del software cuando se desarrollan nuevos sistemas;
- entenderá el concepto de framework (estructura) de aplicación como un conjunto de objetos reutilizables y cómo pueden usarse los frameworks en el desarrollo de aplicaciones;
- se introducirá a las líneas de producto de software, que se constituyen con una arquitectura central común y componentes configurables de reutilización;
- aprenderá cómo pueden desarrollarse sistemas al configurar y componer sistemas de software de aplicación comercial.

Contenido

16.1 Panorama de la reutilización
16.2 Frameworks de aplicación
16.3 Líneas de productos de software
16.4 Reutilización de productos COTS

La ingeniería de software basada en la reutilización es una estrategia en la que se engrana el proceso de desarrollo para reutilizar el software existente. Aunque la reutilización se propuso como una estrategia de desarrollo hace más de 40 años (McIlroy, 1968), es sólo a partir de 2000 cuando el "desarrollo con reutilización" se convirtió en la norma de los nuevos sistemas empresariales. El movimiento hacia el desarrollo basado en la reutilización fue en respuesta a las demandas para reducir los costos de producción y mantenimiento del software, entregar los sistemas con mayor rapidez y aumentar la calidad del software. Cada vez más compañías consideran su software como un activo valioso. Así, promueven su reutilización para incrementar el rendimiento en las inversiones de software.

La disponibilidad de software reutilizable aumentó drásticamente. El movimiento de fuente abierta significó que se dispone de una enorme base de código de reutilización a bajo costo. Esto puede ser en la forma de librerías de programa o aplicaciones completas. Existen muchos sistemas de aplicación de dominio específico disponibles que pueden ajustarse y adaptarse a las necesidades de una compañía específica. Algunas grandes compañías ofrecen varios componentes de reutilización para sus clientes. Los estándares, como los estándares de servicios Web, han facilitado el desarrollo de servicios generales y su reutilización a través de varias aplicaciones.

La ingeniería de software basada en la reutilización es un enfoque al desarrollo que trata de maximizar la reutilización del software existente. Las unidades de software que se reutilizan pueden ser de tamaños sustancialmente diferentes. Por ejemplo:

1. *Reutilización del sistema de aplicación* Todo un sistema de aplicación puede reutilizarse al incorporarlo sin cambios en otros sistemas o al configurar la aplicación para diferentes clientes. De manera alternativa, pueden desarrollarse familias de aplicación que, aunque tengan una arquitectura común, se ajustan a clientes específicos. Más adelante, en este capítulo, se hablará de la reutilización de componentes.

2. *Reutilización de componentes* Los componentes de una aplicación, que varían en tamaño desde subsistemas hasta objetos individuales, pueden reutilizarse. Por ejemplo, un sistema de identificación de patrones desarrollado como parte de un sistema de procesamiento de texto puede reutilizarse en un sistema de administración de base de datos. La reutilización de componentes se desarrolla en los capítulos 17 y 19.

3. *Reutilización de objetos y funciones* Pueden reutilizarse los componentes de software que implementan una sola función, tal como una función matemática, o una clase de objeto. Esta forma de reutilización en torno a librerías estándar ha sido común durante los últimos 40 años. Muchas librerías de funciones y clases están disponibles de manera gratuita. Las clases y funciones en dichas librerías se reutilizan al vincularlas con un código de aplicación de desarrollo reciente. Este enfoque es particularmente efectivo en áreas como algoritmos matemáticos y gráficas, donde se necesita experiencia especializada para desarrollar objetos y funciones eficientes.

Los sistemas y componentes de software son entidades potencialmente reutilizables, pero su naturaleza específica significa que en ocasiones es costoso modificarlos para una nueva situación. Una forma complementaria de reutilización es la "reutilización de concepto" en la que, en vez de reutilizar un componente de software, se reutiliza una idea, una vía, un funcionamiento o un algoritmo. El concepto que se reutiliza se representa en una notación abstracta (por ejemplo, un modelo de sistema), que no incluye detalles de implementación. Por lo tanto, puede configurarse y adaptarse para varias situaciones. El concepto de reutilización puede incorporarse como en el diseño de patrones de diseño (que se estudiaron en el capítulo 7), productos de sistema configurables y generadores de

Beneficio	Explicación
Confiabilidad creciente	El software de reutilización, que se experimentó y ensayó en sistemas operativos, debe ser más confiable que el software nuevo. Sus fallas de diseño e implementación debieron descubrirse y corregirse.
Reducción de riesgo de proceso	Se conoce ya el costo del software existente, mientras que el de desarrollo siempre es una cuestión de juicio. Éste es un factor importante para la gestión del proyecto, ya que reduce el margen de error en la estimación de costos del proyecto. Esto es particularmente cierto cuando se reutilizan componentes de software relativamente grandes, como los subsistemas.
Uso efectivo de especialistas	En lugar de hacer el mismo trabajo una y otra vez, los especialistas de aplicación pueden desarrollar software de reutilización que encapsule su conocimiento.
Cumplimiento de estándares	Algunos estándares, como los de la interfaz de usuario, pueden implementarse como un conjunto de componentes de reutilización. Por ejemplo, si los menús en una interfaz de usuario se implementan con componentes reutilizables, todas las aplicaciones presentan a los usuarios los mismos formatos de menú. El uso de interfaces de usuario estándar mejora la confiabilidad, porque los usuarios cometen menos errores cuando se les presenta una interfaz familiar.
Desarrollo acelerado	Llevar un sistema al mercado tan rápido como sea posible con frecuencia es más importante que los costos totales de desarrollo. La reutilización de software puede acelerar la producción del sistema, ya que pueden reducirse los tiempos de desarrollo y validación.

Figura 16.1 Beneficios de la reutilización de software

programa. Cuando se reutilizan conceptos, el proceso de reutilización incluye una actividad donde los conceptos abstractos se ejemplifican para crear componentes ejecutables de reutilización.

Una ventaja evidente de la reutilización de software es que deberían reducirse los costos totales de desarrollo. Habrá que especificar, diseñar, implementar y validar menos componentes de software. Sin embargo, la reducción del costo es sólo una ventaja de la reutilización. En la figura 16.1 se mencionan otras ventajas de la reutilización de los activos de software.

No obstante, existen costos y problemas asociados con la reutilización (figura 16.2). Hay un costo significativo asociado con entender si un componente es adecuado o no para su reutilización en una situación particular, y probar dicho componente para garantizar su confiabilidad. Tales costos adicionales significan que las reducciones en los costos totales del desarrollo mediante la reutilización pueden ser menores que lo previsto.

Como se estudió en el capítulo 2, los procesos de desarrollo de software deben adaptarse para tomar en cuenta la reutilización. En particular, debe haber una etapa de corrección de requerimientos en la que se modifiquen los requerimientos del sistema para reflejar el software de reutilización disponible. Las etapas de diseño e implementación del sistema pueden incluir también actividades explícitas para buscar y evaluar componentes candidatos para su reutilización.

La reutilización de software es más efectiva cuando se planea como parte de un programa de reutilización de toda la organización. Un programa de reutilización implica la creación de activos reutilizables y la adaptación de procesos de desarrollo para incorporar dichos activos en el software nuevo. La importancia de la planeación de reutilización se reconoce desde hace varios años en Japón (Matsumoto, 1984), donde la reutilización era parte integral del enfoque “fabril” japonés al desarrollo de software (Cusamano, 1989).

Problema	Explicación
Costos crecientes de mantenimiento	Si no está disponible el código fuente de un sistema o componente de software de reutilización, entonces los costos de mantenimiento podrían ser superiores, porque los elementos reutilizados del sistema pueden volverse cada vez más incompatibles con los cambios del sistema.
Falta de apoyo de herramientas	Algunas herramientas de software no apoyan el desarrollo con reutilización. Tal vez sea difícil o imposible integrar dichas herramientas con un sistema de librería de componentes. El proceso de software supuesto por dichas herramientas quizá no tome en cuenta la reutilización. Esto es particularmente cierto para herramientas que dan apoyo a la ingeniería de sistemas embebidos, aunque menos cierto para herramientas de desarrollo orientadas a objetos.
Síndrome "no se inventó aquí"	Algunos ingenieros de software prefieren rescribir los componentes, porque consideran que pueden mejorarlos. Esto en parte tiene que ver con la confianza y en parte con el hecho de que escribir software original se observa como más desafiante que reutilizar el software de alguien más.
Creación, mantenimiento y uso de una librería de componentes	Suele ser costoso dotar a una librería de componentes de reutilización y garantizar que los desarrolladores de software puedan utilizar esta librería. Hay que adaptar procesos de desarrollo para asegurar que se use la librería.
Descubrimiento, comprensión y adaptación de componentes de reutilización	Deben descubrirse componentes de software en una librería, entenderse y, en ocasiones, adaptarse para trabajar en un nuevo entorno. Los ingenieros deben estar ampliamente seguros de encontrar un componente en la librería antes de incluir una búsqueda de componentes como parte de su proceso de desarrollo normal.

Figura 16.2 Problemas con la reutilización

Compañías tales como Hewlett-Packard han tenido también mucho éxito en sus programas de reutilización (Griss y Wosser, 1995), y su experiencia se documentó en un libro de Jacobson *et al*. (1997).

16.1 Panorama de la reutilización

Durante los últimos 20 años, se han desarrollado muchas técnicas para apoyar la reutilización de software. Dichas técnicas aprovechan el hecho de que los sistemas en el mismo dominio de aplicación son similares y tienen potencial de reutilización (esa reutilización es posible a diferentes niveles desde simples funciones hasta completas aplicaciones), y el hecho de que estándares para componentes reutilizables facilitan la reutilización. La figura 16.3 establece algunas formas posibles de implementar la reutilización de software, y en la figura 16.4 se describe brevemente cada una de ellas.

Teniendo en cuenta este arreglo de técnicas para la reutilización, la pregunta clave es: ¿Cuál es la técnica más adecuada para usar en una situación particular? Desde luego, esto depende de los requerimientos del sistema a desarrollar, la tecnología y los activos reutilizables disponibles, y la experiencia del equipo de desarrollo. Los factores clave que deben considerarse al planear la reutilización son:

1. *El calendario de desarrollo para el software* Si el software debe desarrollarse rápidamente, usted debe tratar de reutilizar sistemas comerciales en vez de

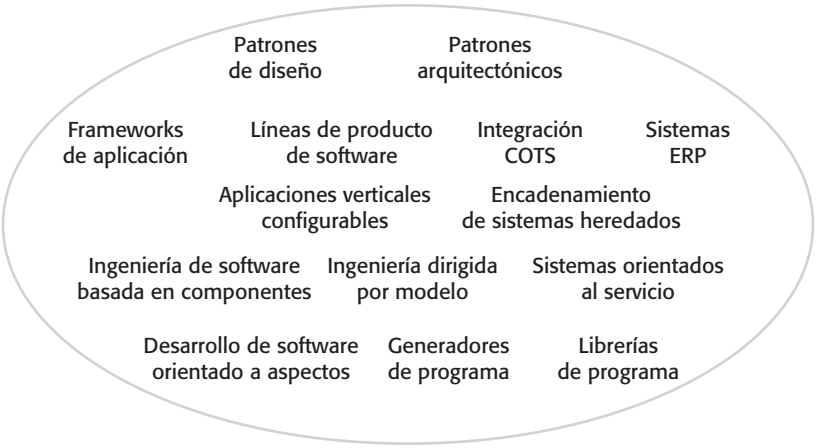


Figura 16.3 Panorama de reutilización

componentes individuales. Éstos son activos reutilizables de grano grueso. Aunque el ajuste con los requerimientos tal vez sea imperfecto, este enfoque minimiza la cantidad de desarrollo requerido.

2. *La vida esperada del software* Si usted desarrolla un sistema de prolongada duración, debe enfocarse en la capacidad de mantenimiento del sistema. No sólo debe considerar los beneficios inmediatos de la reutilización, sino también las implicaciones a largo plazo.

   Durante su vida útil, podrá adaptar el sistema a nuevos requerimientos, lo que significará hacer cambios a partes del sistema. Si no tiene acceso al código fuente, es preferible evitar los componentes COTS (véase la sección 16.4) y los sistemas de proveedores externos, pues éstos tal vez no sean capaces de continuar con el servicio de apoyo para el software de reutilización.

3. *Los antecedentes, las habilidades y la experiencia del equipo de desarrollo* Todas las tecnologías de reutilización son bastante complejas, y es necesario mucho tiempo para entenderlas y usarlas de manera efectiva. Por consiguiente, si el equipo de desarrollo tiene habilidades en un área particular, probablemente es ahí donde deban enfocarse.

4. *La criticidad del software y sus requerimientos no funcionales* Para un sistema crítico que deba certificarse mediante un regulador externo, quizá deba crear un caso de confiabilidad para el sistema (como se estudió en el capítulo 15). Esto es difícil si no tiene acceso al código fuente del software. Si su software cuenta con requerimientos rigurosos de rendimiento, tal vez sea imposible usar estrategias tales como la reutilización basada en generador, donde el código se forma a partir de una representación de reutilización específica de dominio de un sistema. Estos sistemas a menudo crean un código relativamente ineficiente.

5. *El dominio de aplicación* En algunos dominios de aplicación, tales como los sistemas de fabricación e información médica, existen muchos productos genéricos que pueden reutilizarse al configurarlos para una situación local. Si trabaja en tal dominio, siempre debe considerarlo como una opción.

Enfoque	Descripción
Patrones arquitectónicos	Se usan arquitecturas de software estándar que apoyan tipos comunes de sistemas de aplicación, tales como la base de las aplicaciones. Se describen en los capítulos 6, 13 y 20.
Patrones de diseño	Las abstracciones genéricas que ocurren a través de las aplicaciones se representan como patrones de diseño que muestran objetos e interacciones abstractas y concretas. Se detallan en el capítulo 7.
Desarrollo basado en componentes	Se desarrollan sistemas al integrar componentes (colecciones de objetos) que se conforman a estándares de modelo de componentes. Se estudian en el capítulo 17.
Frameworks de aplicación	Colecciones de clases abstractas y concretas adaptadas y extendidas para crear sistemas de aplicación.
Encadenamiento de sistemas heredados	Los sistemas heredados (véase el capítulo 9) se "enlazan" al definir un conjunto de interfaces y proporcionar acceso a estos sistemas heredados a través de dichas interfaces.
Sistemas orientados a servicios	Se desarrollan sistemas mediante la vinculación de servicios compartidos, que pueden proporcionarse externamente. Se tratan en el capítulo 19.
Líneas de producto de software	Un tipo de aplicación se generaliza en torno a una arquitectura común, de forma que pueda adaptarse para diferentes clientes.
Reutilización de productos COTS	Los sistemas se desarrollan al configurar e integrar sistemas de aplicación existentes.
Sistemas ERP	Los sistemas a gran escala que encapsulan funcionalidad empresarial genérica y reglas se configuran para una organización.
Aplicaciones verticales configurables	Se diseñan sistemas genéricos de manera que puedan configurarse a las necesidades específicas de clientes del sistema.
Librerías de programa	Librerías de clase y función que implementan abstracciones de uso común están disponibles para reutilización.
Ingeniería dirigida por modelo	El software se representa como modelos de dominio y modelos independientes de implementación, y se genera un código a partir de dichos modelos. Se refiere en el capítulo 5.
Generadores de programa	Un sistema generador incrusta conocimiento de un tipo de aplicación y se usa para generar sistemas en dicho dominio a partir de un modelo de sistema suministrado por el usuario.
Desarrollo de software orientado a aspectos	Cuando se compila el programa, los componentes compartidos se hilvanan dentro de una aplicación en lugares diferentes. Se expone en el capítulo 21.

Figura 16.4 Enfoques que apoyan la reutilización de software

6. *La plataforma en la que operará el sistema* Algunos modelos de componentes, como .NET, se especifican para plataformas Microsoft. De igual modo, sistemas genéricos de aplicación pueden ser específicos de plataforma y sólo podrá reutilizarlos si su sistema está diseñado para la misma plataforma.

Reutilización basada en generador

La reutilización basada en generador consiste en incorporar conceptos y conocimiento reutilizables en herramientas automatizadas, y ofrecer una forma sencilla de que los usuarios de la herramienta integren un código específico con este conocimiento genérico. Por lo general, este enfoque es más efectivo en aplicaciones específicas de dominio. Soluciones conocidas a los problemas en dicho dominio se incrustan en el sistema generador y el usuario las selecciona para crear un nuevo sistema.

http://www.SoftwareEngineering-9.com/Web/Reuse/Generator.html

La gama de técnicas disponibles de reutilización es tal que, en la mayoría de las situaciones, existe la posibilidad de cierta reutilización de software. Ya sea que se logre o no la reutilización, con frecuencia es un conflicto administrativo más que técnico. Los administradores pueden estar ansiosos de comprometer sus requerimientos para permitir el uso de componentes de reutilización. Es posible que no comprendan los riesgos asociados con la reutilización, tanto como entienden los riesgos del desarrollo original. Aunque los riesgos del desarrollo de nuevo software pueden ser elevados, algunos administradores prefieren los riesgos conocidos a los desconocidos.

16.2 Frameworks de aplicación

Los primeros entusiastas del desarrollo orientado a objetos sugerían que uno de los beneficios clave de usar un enfoque orientado a objetos era que éstos pudieran reutilizarse en diferentes sistemas. Sin embargo, la experiencia demostró que los objetos normalmente son muy pequeños y están especializados para una aplicación particular. Entender y adaptar el objeto tarda más tiempo que volver a implementarlo. Ahora se ha vuelto claro que la reutilización orientada a objetos tiene mejor apoyo en un proceso de desarrollo orientado a objetos a través de abstracciones de grano más grueso, llamadas frameworks (estructuras).

Como sugiere el nombre, un framework es una estructura genérica que se extiende para crear una aplicación o un subsistema más específico. Schmidt y sus colaboradores (2004) definen un framework como:

> *. . . un conjunto integrado de artefactos de software (tales como clases, objetos y componentes), que colaboran en la facilitación de una arquitectura de reutilización para una familia de aplicaciones relacionadas.*

Los frameworks brindan soporte para características genéricas que es probable que sean utilizadas en todas las aplicaciones de un tipo similar. Por ejemplo, un framework de interfaz de usuario ofrecerá soporte para manejo de evento de interfaz, e incluirá un conjunto de artilugios que pueden usarse para construir despliegues. Entonces se permite al desarrollador especializar éstos al agregar funcionalidad específica para una aplicación particular. Por ejemplo, en un framework de interfaz de usuario, el desarrollador define plantillas de despliegue adecuadas para la aplicación a implementar.

Los frameworks apoyan la reutilización de diseño en la que ofrecen una arquitectura que sirve de esqueleto para la aplicación, así como la reutilización de clases específicas

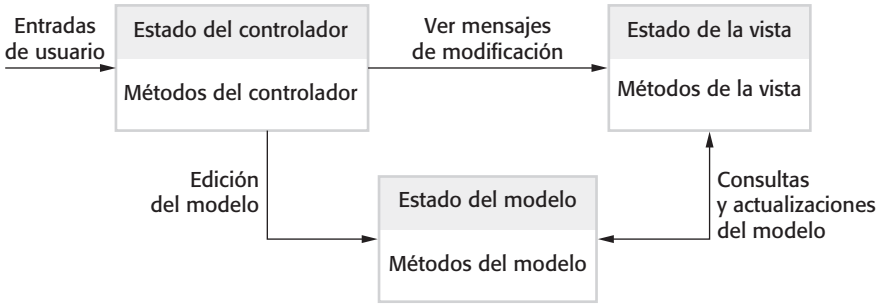


Figura 16.5 Patrón Modelo-Vista-Controlador

en el sistema. La arquitectura se define por las clases de objetos y sus interacciones. Las clases se reutilizan directamente y pueden extenderse utilizando características como la herencia.

Los frameworks se implementan como una colección de clases de objetos concretos y abstractos en un lenguaje de programación orientado a objetos. Por lo tanto, los frameworks son específicos del lenguaje. Existen frameworks disponibles en todos los lenguajes de programación orientados a objetos de uso común (por ejemplo, Java, C#, C++, así como lenguajes dinámicos como Ruby y Python). De hecho, un framework puede incorporar muchos otros frameworks, cada uno de los cuales se diseña para soportar el desarrollo de parte de la aplicación. Es posible usar un framework para crear una aplicación completa o implementar parte de una aplicación, como la interfaz de usuario gráfica.

Fayad y Schmidt (1997) analizan tres clases de frameworks:

1. *Frameworks de infraestructura de sistema* Dichos frameworks apoyan el desarrollo de infraestructuras de sistema como comunicaciones, interfaces de usuario y compiladores (Schmidt, 1997).
2. *Frameworks de integración de middleware* Consisten en un conjunto de estándares y clases de objetos asociados que soportan comunicación de componentes e intercambio de información. Los ejemplos de este tipo de framework incluyen .NET de Microsoft y Enterprise Java Beans (EJB). Dichos frameworks brindan soporte para modelos estandarizados de componentes, como se estudia en el capítulo 17.
3. *Frameworks de aplicación empresarial* Se ocupan de dominios de aplicación específicos, tales como los sistemas de telecomunicaciones o financieros (Baumer *et al*., 1997). El conocimiento del dominio de la aplicación integra y apoya el desarrollo de aplicaciones de usuario final.

Los frameworks de aplicación Web (WAF) son un tipo de framework más reciente e importante. Ahora los WAF, que apoyan la construcción de sitios Web dinámicos, están ampliamente disponibles. La arquitectura de un WAF se basa por lo general en el patrón compuesto Modelo-Vista-Controlador (MVC) (Gamma *et al*., 1995), que se muestra en la figura 16.5.

El patrón MVC se propuso originalmente en la década de 1980, como un enfoque al diseño GUI que permitía múltiples presentaciones de un objeto y separaba estilos de interacción con cada una de estas presentaciones. Permite la separación del estado de

aplicación de la interfaz de usuario de la aplicación. Un framework MVC permite la presentación de datos en diferentes formas y admite la interacción con cada una de dichas presentaciones. Cuando los datos se modifican a través de una de las presentaciones, se modifica el modelo del sistema y los controladores asociados con cada punto de vista actualizan su presentación.

A menudo, los frameworks son implementaciones de patrones de diseño, como se estudió en el capítulo 7. Por ejemplo, un framework MVC incluye el patrón Observer, el patrón Strategy, el patrón Composite y algunos otros que analizan Gamma y sus colaboradores (1995). La naturaleza general de los patrones y su uso de clases abstractas y concretas permite la extensibilidad. Es casi seguro que, sin patrones, los frameworks serían poco prácticos.

Los frameworks de aplicación Web incorporan regularmente uno o más frameworks especializados que apoyen las características específicas de aplicación. Aunque cada framework incluye funcionalidad sutilmente diferente, la mayoría de los frameworks de aplicación Web soportan las siguientes características:

1. *Seguridad* Los WAF pueden incluir clases para ayudar a implementar autenticación de usuario (login) y el control de acceso para garantizar que los usuarios sólo puedan tener acceso a la funcionalidad que permite el sistema.
2. *Páginas Web dinámicas* Se ofrecen clases para ayudar a definir las plantillas de la página Web y dotar dinámicamente a éstas con datos específicos de la base de datos del sistema.
3. *Soporte de base de datos* Los frameworks usualmente no incluyen una base de datos, sino suponen que se usará una base de datos separada, como MySQL. El framework puede proporcionar clases que ofrezcan una interfaz abstracta a diferentes bases de datos.
4. *Gestión de sesión* Clases para crear y administrar sesiones (algunas interacciones con el sistema por parte de un usuario) por lo general son parte de un WAF.
5. *Interacción de usuarios* La mayoría de los frameworks Web brindan ahora soporte AJAX (Holdener, 2008), que permite la creación de páginas Web más interactivas.

Para extender un framework no es necesario cambiar el código de éste. En vez de ello, usted agrega clases concretas que heredan operaciones de clases abstractas en el framework. Además, quizá deba definir callbacks (comunicaciones de regreso). Los callbacks son métodos que se solicitan en respuesta a eventos reconocidos por el framework. Schmidt y sus colaboradores (2004) llaman a esto "inversión de control". Los objetos framework, y no los objetos específicos de aplicación, son los responsables del control en el sistema. En respuesta a eventos de la interfaz del usuario, base de datos, etcétera, dichos objetos framework recurren a "métodos gancho" que entonces se vinculan a la funcionalidad que proporciona el usuario. La funcionalidad específica de la aplicación responde de una forma adecuada al evento (figura 16.6). Por ejemplo, un framework tendrá un método que manipule un clic del ratón desde el entorno. Este método llama a un método gancho, que usted debe configurar con la finalidad de llamar a los métodos de aplicación adecuados para manejar el clic del ratón.

Las aplicaciones que se construyen utilizando frameworks pueden ser la base para una posterior reutilización a través del concepto de líneas de producto de software o familias

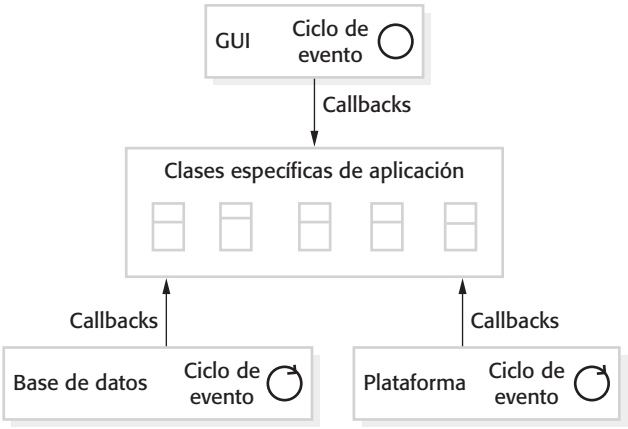


Figura 16.6 Inversión de control en frameworks

de aplicación. Puesto que dichas aplicaciones se construyen mediante un framework, la modificación de los miembros de la familia para crear instancias del sistema con frecuencia es un proceso directo. Implica rescribir clases y métodos concretos que deberán adicionarse al framework.

Sin embargo, los frameworks usualmente son más generales que las líneas de producto de software, enfocadas en una familia específica de sistemas de aplicación. Por ejemplo, es posible usar un framework basado en Web para construir diversos tipos de aplicaciones basadas en Web. Uno de éstos podría ser una línea de productos de software que soporte escritorios de ayuda basados en Web. Entonces esta "línea de productos de escritorio de ayuda" puede especializarse aún más para ofrecer tipos particulares de soporte de escritorio de ayuda.

Los frameworks son un enfoque efectivo a la reutilización, aunque costosos para introducirse en procesos de desarrollo de software. Se consideran inherentemente complejos, y aprender a usarlos es una actividad que tal vez dure muchos meses. También es posible que sea difícil y costoso evaluar frameworks disponibles para elegir los más adecuados. Depurar las aplicaciones basadas en frameworks resulta complicado, porque tal vez no se comprenda cómo interactúan los métodos framework. Éste es un problema general con el software de reutilización. Las herramientas de depuración pueden brindar datos sobre los componentes del sistema de reutilización que un desarrollador no entiende.

16.3 Líneas de productos de software

Uno de los enfoques más efectivos para la reutilización es la creación de líneas de productos de software o familias de aplicación. Una línea de productos de software es un conjunto de aplicaciones con una arquitectura común y componentes compartidos, con cada aplicación especializada para reflejar diferentes requerimientos. El sistema central se diseña para configurarse y adaptarse a las necesidades de los diferentes clientes del sistema. Esto puede implicar la configuración de algunos componentes, implementar componentes adicionales y modificar varios de los componentes de acuerdo con nuevos requerimientos.

Desarrollar aplicaciones mediante la adaptación de una versión genérica de la aplicación significa que se reutiliza una alta proporción del código de aplicación. Además, la experiencia de la aplicación muchas veces es transferible de un sistema a otro. En consecuencia, cuando los ingenieros de software se reúnen en un equipo de desarrollo, se acorta su proceso de aprendizaje. Las pruebas se simplifican porque éstas también pueden reutilizarse para gran parte de la aplicación, lo que en consecuencia reduce el tiempo total del desarrollo de la aplicación.

Las líneas de producto de software surgen por lo general de aplicaciones existentes. Esto es, una organización desarrolla una aplicación y, luego, cuando se requiere un sistema similar, el código de ésta se reutiliza de manera informal en la nueva aplicación. El mismo proceso se usa conforme se desarrollan otras aplicaciones similares. Sin embargo, el cambio tiende a corromper la estructura de la aplicación, así que, a medida que se desarrollan más instancias nuevas, se vuelve cada vez más difícil crear una nueva versión. Por consiguiente, puede tomarse entonces una decisión para diseñar una línea de productos genéricos. Esto implica identificar la funcionalidad común en instancias de producto e incluirla en una aplicación base, que después se usa para un desarrollo futuro. Esta aplicación base se estructura deliberadamente para simplificar la reutilización y la reconfiguración.

Desde luego, los frameworks de aplicación y las líneas de productos de software tienen mucho en común. Ambos soportan una arquitectura y componentes comunes, y requieren un nuevo desarrollo para crear una versión específica de un sistema. Las principales diferencias entre dichos enfoques son las siguientes:

1. Los frameworks de aplicación se apoyan en características orientadas a objetos, como herencia y polimorfismo, para implementar extensiones al framework. Por lo general, el código framework no se modifica y las posibles modificaciones están limitadas a lo que permite el framework. Las líneas de producto de software no necesariamente se crean mediante un enfoque orientado a objetos. Los componentes de aplicación cambian, se borran o rescriben. No hay límites, al menos en principio, para que puedan realizarse cambios.

2. Los frameworks de aplicación se enfocan principalmente en brindar apoyo técnico antes que dominio específico. Por ejemplo, existen frameworks de aplicación para crear aplicaciones basadas en Web. Una línea de productos de software por lo general incrusta información detallada de dominio y de plataforma. Por ejemplo, podría haber una línea de productos de software que se ocupe de aplicaciones basadas en Web para la administración de registros de salud.

3. Las líneas de producto de software generalmente son aplicaciones de control para equipo. Por ejemplo, puede haber una línea de productos de software para una familia de impresoras. Esto significa que la línea de productos debe dar soporte para interfaz de hardware. Los frameworks de aplicación con frecuencia están orientados al software, y pocas veces ofrecen soporte para interfaz de hardware.

4. Las líneas de productos de hardware constituyen una familia de aplicaciones relacionadas, propiedad de la misma organización. Cuando usted crea una nueva aplicación, su punto de partida es habitualmente el miembro más cercano de la familia de aplicación, no la aplicación central genérica.

Si desarrolla una línea de productos de software usando un lenguaje de programación orientado a objetos, en ese momento puede usar un framework de aplicación como base del

sistema. Usted crea el núcleo de la línea de productos al extender el framework con componentes específicos de dominio, a través de sus mecanismos internos. Entonces hay una segunda fase de desarrollo, donde se crean versiones del sistema para diferentes clientes.

Pueden desarrollarse varios tipos de especialización de una línea de productos de software:

1. *Especialización de plataforma* Se elaboran versiones de la aplicación para diferentes plataformas. Por ejemplo, pueden existir versiones de aplicación para plataformas Windows, Mac OS y Linux. En este caso, la funcionalidad de la aplicación por lo regular no cambia; sólo se modifican aquellos componentes que hacen interfaz con el hardware y el sistema operativo.
2. *Especialización de entorno* Se crean versiones de aplicación para manejar entornos operacionales particulares y dispositivos periféricos. Por ejemplo, un sistema para los servicios de emergencia puede existir en diferentes versiones, dependiendo del sistema de comunicaciones en los vehículos. En este caso, los componentes del sistema cambian para reflejar la funcionalidad del equipo de comunicación utilizado.
3. *Especialización funcional* Se crean versiones de aplicación para clientes específicos que tengan diferentes requerimientos. Por ejemplo, un sistema de automatización bibliotecario se podrá modificar en función de si se utiliza en una biblioteca pública, una biblioteca de consulta o una biblioteca universitaria. En tal caso, es posible variar los componentes que implementan la funcionalidad y agregar al sistema nuevos componentes.
4. *Especialización de proceso* El sistema se adapta para hacer frente a los procesos empresariales específicos. Por ejemplo, un sistema de pedidos puede adaptarse para enfrentarse con un proceso de pedidos centralizado en una compañía y con un proceso distribuido en otra.

La arquitectura de una línea de productos de software con frecuencia refleja un estilo o patrón arquitectónico general, específico para una aplicación. Por ejemplo, considere un sistema de línea de productos que se diseñe con la finalidad de manejar salidas de vehículos para servicios de emergencia. Los operadores de este sistema reciben las llamadas de los incidentes, localizan el vehículo adecuado para responder a la emergencia y envían el vehículo al sitio del incidente. Los desarrolladores de tal sistema pueden comercializar versiones de éste para los servicios de policía, bomberos y ambulancias.

Este sistema de envío de vehículos es un ejemplo de una arquitectura de gestión de recursos (figura 16.7). En la figura 16.8 se ejemplifica dicha estructura de cuatro capas, y se muestran los módulos que es posible incluir en una línea de productos de sistemas de envío de vehículos. Los componentes en cada nivel en el sistema de línea de productos son los siguientes:

1. A nivel de interacción, existen componentes que ofrecen una interfaz de pantalla de operador y una interfaz con los sistemas de comunicaciones usados.
2. A nivel de la gestión I/O (nivel 2), hay componentes que manejan autenticación del operador, generan reportes de incidentes y vehículos enviados, permiten salida de mapas y planeación de rutas, y ofrecen un mecanismo para que los operadores consulten las bases de datos del sistema.

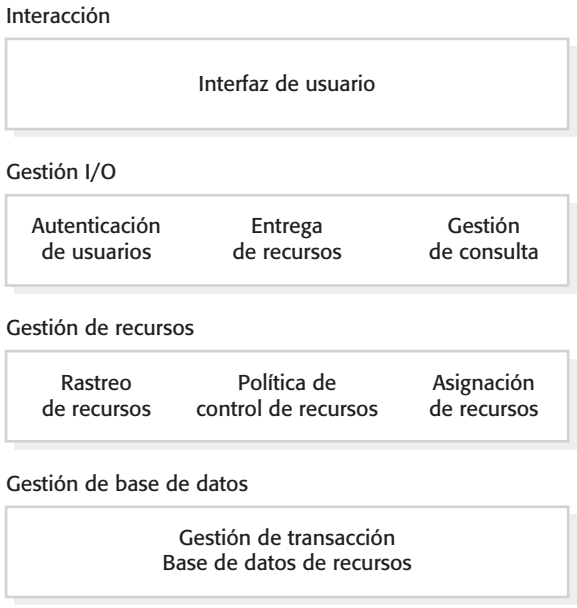


Figura 16.7
Arquitectura de un sistema de asignación de recursos

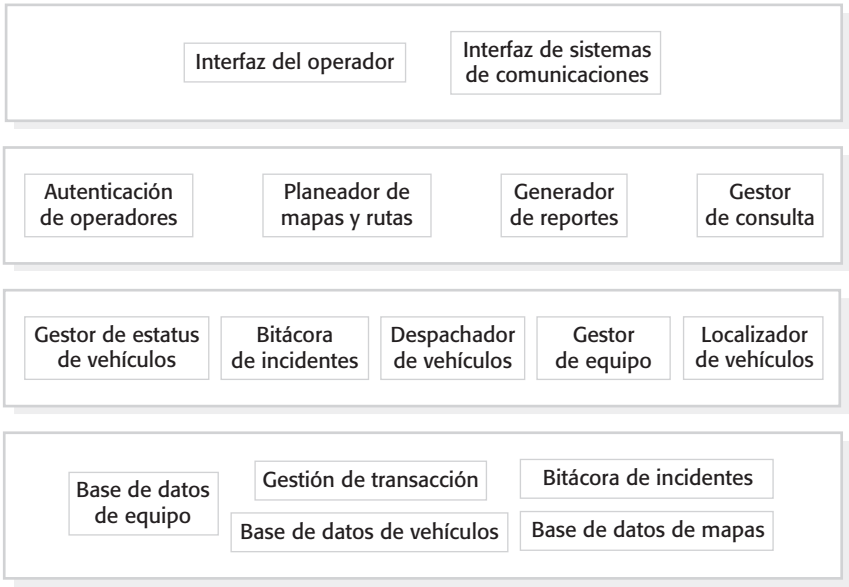


Figura 16.8
Arquitectura de línea de productos de un sistema de envío de vehículos

3. A nivel de gestión de recursos (nivel 3) existen componentes que permiten la localización y el envío de vehículos, componentes para actualizar el estatus de los vehículos y el equipo, y un componente para registrar detalles de incidentes.
4. A nivel de base de datos, además del apoyo de gestión de transacción usual, existen bases de datos separadas de vehículos, equipo y mapas.

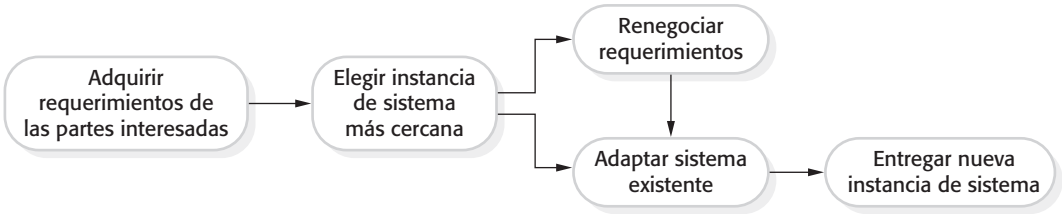


Figura 16.9 Desarrollo de instancias de producto

Para crear una versión específica de este sistema, habrá que modificar los componentes individuales. Por ejemplo, la policía tiene un mayor número de vehículos, pero menor número de tipos de vehículos, mientras que los bomberos cuentan con muchos tipos de vehículos especializados. En consecuencia, tal vez sea necesario definir una estructura diferente de base de datos de vehículos cuando se implemente un sistema para esos distintos servicios.

La figura 16.9 muestra los pasos considerados en la extensión de una línea de productos de software para crear una nueva aplicación. Tales pasos en este proceso general son los siguientes:

1. *Adquirir requerimientos de las partes interesadas* Puede comenzar con un proceso de ingeniería de requerimientos normal. Sin embargo, puesto que ya existe un sistema, éste deberá comprobarse para luego hacer que los interesados lo experimenten y expresen sus requerimientos como modificaciones a las funciones ofrecidas.
2. *Seleccionar el sistema existente que se ajuste más a los requerimientos* Cuando se crea un nuevo miembro de una línea de productos, puede comenzar con la instancia más cercana al producto. Se analizan los requerimientos y se elige modificar al miembro de la familia más cercano.
3. *Renegociar los requerimientos* Conforme surgen más detalles de cambios requeridos y el proyecto se planea, puede haber alguna renegociación de requerimientos para minimizar los cambios necesarios.
4. *Adaptar el sistema existente* Se desarrollan nuevos módulos para el sistema existente, y los módulos de sistema existentes se adaptan para cumplir con los nuevos requerimientos.
5. *Entregar al nuevo miembro de la familia* Se entrega al cliente la nueva instancia de la línea de producto. En esta etapa, usted debería documentar las características clave, de manera que pueda usarse en el futuro como base para otros desarrollos de sistema.

Cuando usted crea un nuevo miembro de una línea de productos, es probable que encuentre un compromiso entre reutilizar la mayor cantidad de aplicación genérica posible y satisfacer los requerimientos detallados de las partes interesadas. Cuanto más detallados sean los requerimientos del sistema, menos probabilidad tendrá de que los componentes existentes cumplan tales requerimientos. Sin embargo, si las partes interesadas desean ser flexibles y limitar las modificaciones que requiera el sistema, por lo general se podrá entregar el sistema más rápido y a menor costo.

Las líneas de productos de software se diseñan para reconfigurarse, y esta reconfiguración puede implicar el agregar o eliminar componentes del sistema, definir parámetros y

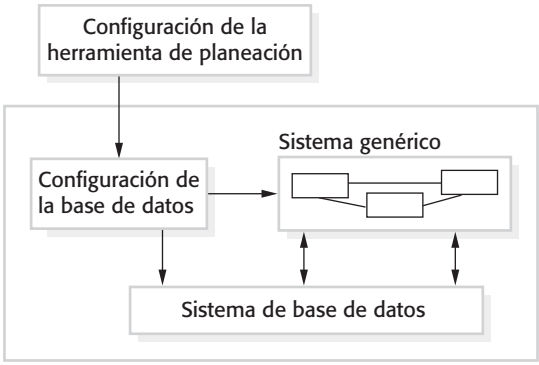


Figura 16.10
Configuración a tiempo de diseño

restricciones para componentes del sistema, e incluir conocimiento de procesos empresariales. Esta configuración puede ocurrir en diferentes etapas en el proceso de desarrollo:

1. *Configuración a tiempo de diseño* La organización que desarrolla el software modifica una línea común de productos básicos mediante el desarrollo, la selección o la adaptación de componentes para crear un nuevo sistema para el cliente.
2. *Configuración a tiempo de implementación* Se diseña un sistema genérico para la configuración por parte de un cliente o de algún consultor que trabaje con el cliente. El conocimiento de los requerimientos específicos del cliente y el entorno operacional del sistema se incrustan en un conjunto de archivos de configuración que usa el sistema genérico.

Cuando un sistema se configura a tiempo de diseño, el proveedor comienza ya sea con un sistema genérico o con una instancia de producto existente. Al modificar y extender los módulos en este sistema se crea un sistema específico que entrega la funcionalidad requerida por el cliente. Esto implica, por lo general, cambiar y extender el código fuente del sistema, de manera que es posible mayor flexibilidad que con la configuración a tiempo de implementación.

La configuración a tiempo de implementación supone usar una herramienta para crear una configuración específica del sistema que se registre en una base de datos o como un conjunto de archivos de configuración (figura 16.10). El sistema en ejecución consulta esta base de datos cuando se ejecuta, así que su funcionalidad puede especializarse para su contexto de ejecución.

Existen muchos niveles de configuración en tiempo de implementación que un sistema puede proveer:

1. Selección de componentes, donde se seleccionan los módulos en un sistema que ofrecen la funcionalidad requerida. Por ejemplo, en un sistema de información de pacientes, es posible seleccionar un componente de gestión de imagen que permita vincular imágenes médicas (rayos X, tomografías, etcétera) con el registro médico del paciente.
2. Definición de flujo de trabajo y reglas, donde se definen los flujos de trabajo (cómo se procesa la información, etapa por etapa) y reglas de validación que deben aplicarse a la información que los usuarios ingresen o que genere el sistema.

3. Definición de parámetros, donde se detallan los valores de parámetros específicos del sistema, que reflejan la instancia de la aplicación que se crea. Por ejemplo, puede especificarse la longitud máxima de los campos para la entrada de datos por parte de un usuario o las características del hardware unido al sistema.

La configuración de la implementación a través de despliegue en pantalla puede ser muy compleja, y la labor de configurar el sistema para un cliente tal vez tarde muchos meses. Los grandes sistemas configurables pueden apoyar el proceso de configuración al ofrecer herramientas de software, tales como herramientas de planeación de configuración, para apoyar el proceso de configuración. En la sección 16.4.1 se estudia con más detalle la configuración de la implementación a través de despliegue en pantalla. Esto cubre la reutilización de sistemas COTS que deben configurarse para trabajar en diferentes entornos potenciales.

La configuración a tiempo de diseño se usa cuando es imposible utilizar las instalaciones existentes de la configuración de implementación a través de despliegue en pantalla en un sistema para desarrollar una nueva versión del sistema. Sin embargo, con el tiempo, cuando se crean varios miembros de la familia con funcionalidad comparable, se puede decidir si se refactoriza la línea central de productos para incluir funcionalidad que se haya implementado en varios miembros de la familia de aplicación. Entonces se hace configurable esta nueva funcionalidad cuando se implementa el sistema.

16.4 Reutilización de productos COTS

Un producto COTS (por las siglas de *Comercial-Off-The-Shelf*) es un sistema de software que puede adaptarse a las necesidades de diferentes clientes sin cambiar el código fuente del sistema. Prácticamente todo el software de escritorio y una gran variedad de productos del servidor son software COTS. Puesto que este software se diseña para uso general, incluye regularmente muchas características y funciones. Por consiguiente, tiene el potencial de reutilización en diferentes entornos como parte de diversas aplicaciones. Torchiano y Morisio (2004) descubrieron que los productos de código abierto se usaron a menudo como productos COTS. Esto es, los sistemas abiertos se empleaban sin cambiar ni observar el código fuente.

Los productos COTS se adaptan al usar mecanismos de configuración internos que permiten que la funcionalidad del sistema se adecue a necesidades específicas del cliente. Por ejemplo, en un sistema de registro de pacientes en un hospital, pueden definirse formatos de entrada y reportes de salida separados para diferentes tipos de pacientes. Otras características de configuración permiten al sistema aceptar plug-ins que extiendan la funcionalidad o comprueben las entradas del usuario para garantizar que son válidas.

Este enfoque para la reutilización del software se adoptó ampliamente en grandes compañías durante los últimos 15 años, puesto que ofrece beneficios significativos sobre el desarrollo de software personalizado:

1. Al igual que sucede con otros tipos de reutilización, es posible la implementación más rápida de un sistema fiable.

2. Es posible ver qué funcionalidad ofrece la aplicación, de manera que es más fácil juzgar si es probable que sea adecuada o no. Quizás otras compañías ya usen las aplicaciones, de manera que existen antecedentes de experiencia con el sistema.
3. Se evitan algunos riesgos de desarrollo al usar software existente. Sin embargo, este enfoque tiene sus propios riesgos, como se verá más adelante.
4. Las empresas pueden enfocarse en su actividad central sin tener que dedicar muchos recursos al desarrollo de sistemas TI.
5. Conforme evolucionan las plataformas operativas, las actualizaciones de tecnología se pueden simplificar, pues éstas son responsabilidad del proveedor del producto COTS y no del cliente.

Desde luego, este enfoque a la ingeniería de software entraña ciertos problemas:

1. Tienen que adaptarse los requerimientos para reflejar la funcionalidad y el modo de operación del producto COTS. Esto puede conducir a cambios bruscos en los procesos empresariales existentes.
2. El producto COTS puede basarse en suposiciones que sean casi imposibles de cambiar. Por lo tanto, el cliente debe adaptar su empresa para reflejar dichas suposiciones.
3. Elegir el sistema COTS correcto para una empresa puede ser un proceso difícil, en especial porque muchos productos COTS no están debidamente documentados. Tomar la decisión equivocada podría ser desastroso, ya que tal vez sea imposible hacer funcionar el nuevo sistema como se requiere.
4. Quizá no haya experiencia local para apoyar el desarrollo de los sistemas. En consecuencia, el cliente deberá apoyarse en el proveedor y en consultores externos para obtener consejos de desarrollo. Estos consejos podrían estar sesgados y dirigidos a vender productos y servicios, y no a satisfacer las necesidades reales del cliente.
5. Los proveedores de productos COTS controlan el soporte y la evolución del sistema. Pueden salir del negocio, perder el control o incluso hacer cambios que ocasionen dificultades a los clientes.

La reutilización de software basado en COTS cada vez se ha vuelto más común. La vasta mayoría de los nuevos sistemas de procesamiento de información empresarial se construyen ahora utilizando COTS, en vez de emplear un enfoque orientado a objetos. Aunque con frecuencia hay problemas con este enfoque para el desarrollo de sistemas (Tracz, 2001), las historias de éxito (Baker, 2002; Balk y Kedia, 2000; Brownsword y Morris, 2003; Pfarr y Reis, 2002) indican que la reutilización basada en COTS reduce el esfuerzo y el tiempo para implementar el sistema.

Existen dos tipos de reutilización de productos COTS, a saber: sistemas de solución COTS y sistemas integrados COTS. Los primeros consisten en una aplicación genérica de un solo proveedor que se configura de acuerdo con los requerimientos del cliente. Los sistemas integrados COTS implican la integración de dos o más sistemas COTS (quizá

Sistemas de solución COTS	Sistemas COTS integrados
Un solo producto que ofrece la funcionalidad requerida por un cliente.	Muchos productos heterogéneos de sistema se integran para ofrecer funcionalidad personalizada.
Basados en una solución genérica y procesos estandarizados.	Pueden desarrollarse soluciones flexibles para procesos del cliente.
El desarrollo se enfoca en la configuración del sistema.	El desarrollo se enfoca en la integración del sistema.
El proveedor del sistema es responsable del mantenimiento.	El dueño del sistema es responsable del mantenimiento.
El proveedor del sistema ofrece la plataforma para el sistema.	El dueño del sistema ofrece la plataforma para el sistema.

Figura 16.11 Sistemas de solución COTS y COTS integrados

de diferentes proveedores) para crear un sistema de aplicación. La figura 16.11 resume las diferencias entre estos diferentes enfoques.

16.4.1 Sistemas de solución COTS

Los sistemas de solución COTS son sistemas de aplicación genéricos que pueden diseñarse para dar apoyo a un tipo de empresa particular, actividad empresarial o, en ocasiones, a toda la empresa. Por ejemplo, es posible generar un sistema de solución COTS para los dentistas que se encargue de registrar citas, llevar expedientes, programar llamadas a pacientes, etcétera. A mayor escala, un sistema de Planeación de Recursos Empresariales (ERP, por las siglas de *Enterprise Resource Planning*) puede apoyar todas las actividades de fabricación, pedidos y servicio de atención a clientes en una compañía grande.

Los sistemas de solución COTS específicos de dominio, como los sistemas que brindan apoyo a una función empresarial (por ejemplo, manejo de documentos), ofrecen la funcionalidad que requieren algunos usuarios potenciales. Sin embargo, tales sistemas incorporan también suposiciones internas acerca de cómo trabajan los usuarios, y esto puede causar problemas en situaciones específicas. Por ejemplo, un sistema que apoye en la tarea de registrar a estudiantes en las universidades puede suponer que los estudiantes se registrarán para obtener un grado determinado en una universidad. Sin embargo, si varias universidades colaboran para ofrecer grados conjuntos, entonces será casi imposible representar esto en el sistema.

Los sistemas ERP, como los producidos por SAP y BEA, son sistemas integrados a gran escala, diseñados para dar apoyo a prácticas empresariales, tales como pedidos y facturación, manejo de inventarios y fechas de producción (O'Leary, 2000). El proceso de configuración para estos sistemas incluye la recopilación de información detallada acerca de la empresa y los procesos empresariales del cliente, e incrustan esto en una base de datos de configuración. Con frecuencia, ello requiere conocimiento detallado de anotaciones y herramientas de configuración, y por lo general lo realizan consultores que trabajan junto con los clientes del sistema.

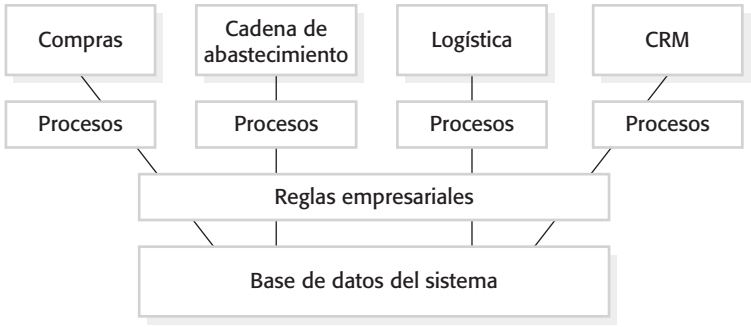


Figura 16.12 Arquitectura de un sistema ERP

Un sistema ERP genérico contiene algunos módulos que pueden componerse en diferentes formas con la finalidad de crear un sistema para un cliente. El proceso de configuración implica elegir cuáles módulos deben incluirse, configurar dichos módulos individuales, definir los procesos y las reglas empresariales, y definir la estructura y organización de la base de datos del sistema. En la figura 16.12 se presenta un modelo de la arquitectura global de un sistema ERP que apoya algunas funciones empresariales.

Las características clave de esta arquitectura son:

1. Algunos módulos para dar apoyo a diferentes funciones empresariales. Estos módulos de grano grueso pueden apoyar a departamentos o divisiones enteros de la empresa. En el ejemplo que se señala en la figura 16.12, los módulos seleccionados para su inclusión en el sistema son: un módulo que apoya las compras, un módulo que apoya la gestión de la cadena de abastecimiento, un módulo logístico que apoya la entrega de bienes, y un módulo de gestión de atención al cliente (CRM, por las siglas de *Customer Relationship Management*) para mantener información sobre la clientela.
2. Un conjunto definido de procesos empresariales, asociados con cada módulo, que se relaciona con actividades en dicho módulo. Por ejemplo, puede haber una definición del proceso de pedidos que especifique cómo crear y aprobar los pedidos. Esto precisará los roles y las actividades implicados en el levantamiento de pedidos.
3. Una base de datos común que mantiene información acerca de todas las funciones empresariales relacionadas. Esto significa que no debe ser necesario duplicar la información, como los detalles de un cliente, en diferentes partes de la empresa.
4. Un conjunto de reglas empresariales que se aplican a todos los datos en la base de datos. Por lo tanto, cuando se ingresan datos desde una función, dichas reglas deben garantizar que esto es consistente con los datos requeridos por otras funciones. Por ejemplo, tal vez exista una regla empresarial que establece que alguien de mayor jerarquía que la persona que hace la solicitud debe aprobar todas las solicitudes de gastos.

Los sistemas ERP se usan casi en todas las grandes compañías para apoyar algunas o todas sus funciones. Por consiguiente, se trata de una forma de reutilización de

software ampliamente difundida. No obstante, la limitación evidente de este enfoque a la reutilización es que la funcionalidad del sistema se restringe a la funcionalidad del núcleo genérico. Más aún, los procesos y las operaciones de una compañía tienen que expresarse en el lenguaje de configuración del sistema, y puede haber desajuste de concordancia entre los conceptos en la empresa y los conceptos incluidos en el lenguaje de configuración.

Por ejemplo, en un sistema ERP que se vendió a una universidad, debía definirse el concepto de cliente. Esto causó grandes dificultades cuando se configuró el sistema. Sin embargo, las universidades tienen múltiples tipos de clientes, como estudiantes, agencias que patrocinan investigaciones, instituciones que proveen fondos educativos, etcétera, cada uno de los cuales posee diferentes características. Ninguno de ellos es realmente comparable a la noción de un cliente comercial (esto es, una persona o empresa que adquiere productos o servicios). Una discordancia seria entre el modelo empresarial usado por el sistema y el del comprador del sistema hace altamente probable que el sistema ERP no cumpla con las necesidades reales del comprador (Scott, 1999).

Tanto los productos de dominio específico COTS como los sistemas ERP por lo general requieren de una extensa configuración para adaptarlos a los requerimientos de cada organización donde se instalan. Esta configuración puede implicar:

1. Seleccionar la funcionalidad requerida del sistema (por ejemplo, al decidir qué módulos deben incluirse).
2. Establecer un modelo de datos que defina cómo se estructurarán los datos de la organización en la base de datos del sistema.
3. Definir las reglas empresariales que se aplican a dichos datos.
4. Definir las interacciones esperadas con sistemas externos.
5. Diseñar los formatos de entrada y los reportes de salida generados por el sistema.
6. Diseñar nuevos procesos empresariales que se conformen al modelo de proceso subyacente apoyado por el sistema.
7. Establecer parámetros que definan cómo se implementará el sistema en su plataforma subyacente.

Una vez completados los escenarios de configuración, el sistema de solución COTS está listo para las pruebas. Las pruebas son un gran problema cuando los sistemas se configuran en vez de programarse con un lenguaje convencional. Puesto que dichos sistemas se construyen mediante una plataforma fiable, las fallas y caídas evidentes del sistema son relativamente excepcionales. Más bien, los problemas con frecuencia son sutiles y se relacionan con las interacciones entre los procesos operacionales y la configuración del sistema. Esto sólo puede ser detectable por parte de los usuarios finales y, por lo tanto, tal vez pase inadvertido durante el proceso de pruebas del sistema. Además, no pueden usarse las pruebas automatizadas de unidad, apoyadas por los frameworks de pruebas como JUnit. Es improbable que el sistema subyacente soporte algún tipo de automatización de pruebas, y es posible que no haya especificación completa del sistema que pueda usarse para derivar pruebas del sistema.

16.4.2 Sistemas COTS integrados

Los sistemas COTS integrados son aplicaciones que incluyen dos o más productos COTS o, en ocasiones, sistemas de aplicación heredados. Usted podrá usar este enfoque cuando no existan sistemas COTS individuales que cumplan todas sus necesidades, o cuando quiera integrar un nuevo producto COTS con los sistemas que ya utiliza. Los productos COTS pueden interactuar a lo largo de sus interfaces de programación de aplicación (API, por las siglas de *Application Programming Interfaces*) o interfaces de servicio si están definidas. Alternativamente, pueden componerse al conectar la salida de un sistema a la entrada de otro o al actualizar las bases de datos usadas por las aplicaciones COTS.

Para desarrollar sistemas usando productos COTS, debe hacer algunas elecciones de diseño:

1. *¿Cuáles productos COTS ofrecen funcionalidad más adecuada?* Por lo general, habrá varios productos COTS disponibles, que pueden combinarse en diferentes formas. Si aún no tiene experiencia con un producto COTS, tal vez le resulte difícil decidir cuál producto es el más adecuado.
2. *¿Cómo se intercambiarán los datos?* Por lo general, diferentes productos usan estructuras y formatos de datos únicos. Usted tendrá que escribir adaptadores que conviertan de una representación a otra. Dichos adaptadores son sistemas a tiempo de ejecución que operan junto con los productos COTS.
3. *¿Qué características de un producto se usarán realmente?* Los productos COTS tal vez incluyan más funcionalidad de la necesaria, y la funcionalidad puede duplicarse a través de diferentes productos. Usted tendrá que decidir cuáles características y en cuál producto son las más adecuadas para sus requerimientos. Si es posible, también debe negar acceso a la funcionalidad no utilizada, porque esto podría interferir con la operación normal del sistema. La falla del primer vuelo del cohete Ariane 5 (Nuseibeh, 1997) fue consecuencia de un problema en un sistema de navegación inercial del sistema Ariane 4 que se reutilizó. Sin embargo, la funcionalidad que falló no se requería en realidad en el Ariane 5.

Considere el siguiente escenario como una ilustración de la integración COTS. Una gran organización pretende desarrollar un sistema de compras que permita al personal realizar pedidos desde su escritorio. Al introducir este sistema en la organización, la compañía estima que puede ahorrar cinco millones de dólares al año. Al centralizar las compras, el nuevo sistema de adquisición puede garantizar que los pedidos siempre se realicen desde proveedores que ofrecen los mejores precios, y que se reduzcan los costos de papeleo asociados con los pedidos. Como sucede con los sistemas manuales, el sistema implica elegir los bienes disponibles de un proveedor, planear un pedido, aprobarlo, enviar la solicitud a un proveedor, recibir los bienes y confirmar el pago.

La compañía tiene un sistema de pedidos heredado que se utiliza en una oficina central de compras. Este software de procesamiento de pedidos está integrado con un sistema existente de facturación y entrega. Para crear el nuevo sistema de pedidos, el sistema heredado se integra con una plataforma de comercio electrónico basada en Web y un sistema de correo electrónico que maneje las comunicaciones con los usuarios.

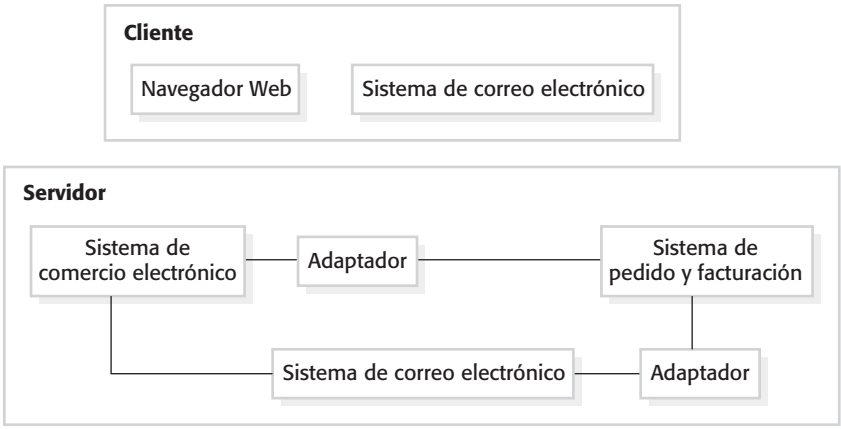


Figura 16.13 Sistema de procuración COTS integrado

En la figura 16.13 se muestra la estructura final del sistema de compras, construida usando COTS.

Este sistema de compras se basa en cliente-servidor y, en el lado del cliente, se usa navegación Web y software de correo electrónico estándar. En el servidor, la plataforma de comercio electrónico debe integrarse con el sistema de pedidos existente a través de un adaptador. El sistema de comercio electrónico tiene su propio formato para los pedidos, confirmaciones de entrega, etcétera, y éstos se convierten en el formato que usa el sistema de pedidos. El sistema de comercio electrónico utiliza el sistema de correo electrónico para enviar notificaciones a los usuarios, pero el sistema de pedidos nunca se diseñó para ello. Por lo tanto, debe escribirse otro adaptador para convertir las notificaciones del sistema de pedidos en mensajes de correo electrónico.

Es posible ahorrar meses, a veces años, de esfuerzo de implementación, y el tiempo de desarrollo e implementación de un sistema puede reducirse drásticamente usando un enfoque COTS integrado. El sistema de compras descrito anteriormente se implementó y desplegó en una compañía muy grande en nueve meses, en vez de los tres años que se requerirían, de acuerdo con las estimaciones, para desarrollar el sistema en Java.

La integración COTS puede simplificarse al usar un enfoque orientado a servicios. En esencia, un enfoque orientado a servicios significa permitir el acceso a la funcionalidad del sistema de aplicación mediante una interfaz de servicio estándar, con un servicio para cada unidad discreta de funcionalidad. Algunas aplicaciones pueden ofrecer una interfaz de servicio, pero, en ocasiones, ésta debe implementarse mediante el integrador de sistema. En esencia, debe programarse una capa (*wrapper*) que oculte la aplicación y ofrezca servicios claros en el exterior (figura 16.14). Este enfoque es valioso en particular para los sistemas heredados que deben integrarse con sistemas de aplicación más recientes.

En principio, la integración de los productos COTS es igual que la integración de cualquier otro componente. Deben entenderse las interfaces del sistema y usarlas de forma exclusiva para comunicarse con el software; habrá que negociar requerimientos específicos contra desarrollo rápido y de reutilización, y diseñar una arquitectura de sistema que permita a los sistemas COTS operar en conjunto.

Sin embargo, el hecho de que tales productos por lo regular sean sistemas grandes por derecho propio, y que se vendan con frecuencia como sistemas independientes, entraña

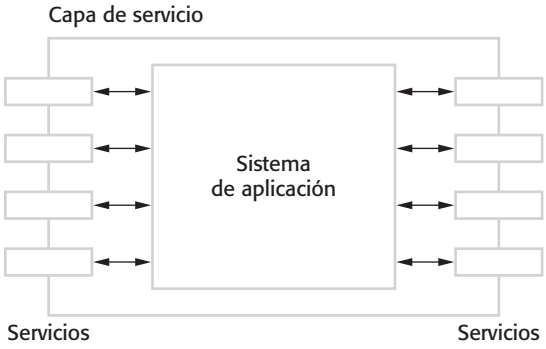


Figura 16.14 Capas de la aplicación

problemas adicionales. Boehm y Abts (1999) examinan cuatro problemas importantes de la integración de sistemas COTS:

1. *Falta de control sobre la funcionalidad y el rendimiento* Aunque la interfaz publicada de un producto indique que ofrece las facilidades requeridas, éstas tal vez no se implementen de manera adecuada o se desempeñen mínimamente. El producto podría tener operaciones ocultas que interfieran con su uso en una situación específica. Corregir dichos problemas puede ser una prioridad para el integrador de productos COTS, pero tal vez no sea una preocupación real para el proveedor del producto. Es posible que los usuarios simplemente tengan que encontrar una solución a los problemas si quieren reutilizar el producto COTS.

2. *Problemas con la interoperabilidad del sistema COTS* A veces es difícil hacer que los productos COTS funcionen en conjunto, porque cada producto establece sus propias suposiciones sobre cómo se utilizará. Garlan y sus colaboradores (1995), al reportar su experiencia para tratar de integrar cuatro productos COTS, descubrieron que tres de estos productos se basaban en eventos, pero cada uno usaba un modelo diferente de eventos. Cada sistema supuso que tenía acceso exclusivo a la cola de eventos. En consecuencia, la integración fue muy difícil. El proyecto requirió cinco veces más de esfuerzo que el que se pronosticó originalmente. La fecha de entrega se extendió a dos años en lugar de los seis meses previstos. En un análisis retrospectivo de su trabajo 10 años después, Garlan y sus colaboradores (2009) concluyeron que no se habían resuelto los problemas de integración descubiertos. Torchiano y Morisio (2004) encontraron que la falta de cumplimiento con los estándares en algunos productos COTS significaba que la integración era más difícil que lo esperado.

3. *Ningún control sobre la evolución del sistema* En respuesta a presiones del mercado, los proveedores de productos COTS toman sus propias decisiones acerca de cambios al sistema. En particular para productos PC, se generan a menudo nuevas versiones, aunque tal vez no sean compatibles con todas las versiones anteriores. Las nuevas versiones pueden tener funcionalidad adicional no deseada, y las versiones anteriores podrían quedar fuera de disposición y no recibir servicios de apoyo.

4. *Soporte de los proveedores COTS* El nivel de soporte disponible de los proveedores COTS varía ampliamente. El servicio de soporte del proveedor es específicamente importante cuando surgen problemas, pues los desarrolladores no tienen acceso al código fuente ni a la documentación detallada del sistema. Aunque quizá

los proveedores se comprometan a brindar servicios de asesoría, tanto el mercado cambiante como las circunstancias económicas podrían dificultar el cumplimiento de este compromiso. Por ejemplo, un proveedor de sistemas COTS tal vez decida discontinuar un producto debido a una demanda limitada, o quizás otra firma tome el control de la compañía proveedora y no quiera dar servicios de apoyo a los clientes que adquirieron ciertos productos.

Boehm y Abts suponen que, en muchos casos, el costo de mantenimiento y evolución del sistema es mayor para los sistemas COTS integrados. Todas las dificultades anteriores son problemas del ciclo de vida que no sólo afectan el desarrollo inicial del sistema. Cuanto mayor sea el número de personas implicadas en el mantenimiento del sistema que se distancie de los desarrolladores originales del sistema, más probable será que surjan dificultades reales con los productos COTS integrados.

PUNTOS CLAVE

- En la actualidad, la mayoría de los nuevos sistemas de software empresariales se desarrollan con la reutilización de conocimiento y código de sistemas aplicados anteriormente.
- Existen muchas diferentes formas de reutilizar el software. Esto varía desde la reutilización de clases y métodos en las librerías hasta la reutilización de sistemas completos de aplicación.
- Las ventajas de la reutilización de software son costos más bajos, desarrollo de software más rápido y menores riesgos. Además, aumenta la confiabilidad del sistema. Los especialistas pueden trabajar de manera más efectiva al concentrar su experiencia en el diseño de componentes de reutilización.
- Los frameworks (estructuras) de aplicación son colecciones de objetos concretos y abstractos que se diseñan para reutilizarse a través de la especialización y la adición de nuevos objetos. Por lo general, incorporan buena práctica de diseño mediante patrones de diseño.
- Las líneas de productos de software son aplicaciones relacionadas que se desarrollan a partir de una o más aplicaciones base. Un sistema genérico se adapta y especializa para satisfacer requerimientos específicos de funcionalidad, plataforma objetivo o configuración operativa.
- La reutilización de productos COTS se ocupa de la reutilización de sistemas comerciales a gran escala. Éstos ofrecen amplia funcionalidad, y es posible que su reutilización reduzca radicalmente los costos y el tiempo de desarrollo. Pueden diseñarse sistemas al configurar un solo producto COTS genérico o al integrar dos o más productos COTS.
- Los sistemas de Planeación de Recursos Empresariales son ejemplos a gran escala de reutilización de COTS. Se crea una instancia de un sistema ERP al configurar un sistema genérico con información acerca de los procesos y las reglas empresariales del cliente.
- Los problemas potenciales con la reutilización basada en COTS incluyen falta de control sobre la funcionalidad y el rendimiento, falta de control sobre la evolución del sistema, la necesidad de servicios de soporte de proveedores externos, y dificultades para garantizar que los sistemas puedan operar de manera conjunta.

LECTURAS SUGERIDAS

Reuse-based Software Engineering. Un análisis completo de los diferentes enfoques para la reutilización de software. Los autores examinan los conflictos de reutilización técnica y administración de los procesos de reutilización. (H. Mili, A. Mili, S. Yacoub y E. Addy, John Wiley & Sons, 2002.)

"Overlooked Aspects of COTS-Based Development". Un interesante artículo que examina una encuesta de desarrolladores que usan un enfoque basado en COTS y los problemas encontrados. (M. Torchiano y M. Morisio, *IEEE Software*, **21** (2), marzo-abril de 2004.) http://dx.doi.org/10.1109/MS.2004.1270770.

"Construction by Configuration: A New Challenge for Software Engineering". Se trata de un artículo que escribió el autor, en el que analiza los problemas y dificultades de construir una nueva aplicación al configurar sistemas existentes. (I. Sommerville, *Proc. 19th Australian Software Engineering Conference*, 2008.) http://dx.doi.org/10.1109/ASWEC.2008.75.

"Architectural Mismatch: Why Reuse Is Still So Hard". Este artículo revisa un ensayo anterior que estudia los problemas de reutilización y la integración de algunos sistemas COTS. Los autores concluyeron que, aunque se ha realizado cierto progreso, aún hay problemas en las suposiciones conflictivas que hicieron los diseñadores de los sistemas individuales. (D. Garlan et al., *IEEE Software*, **26** (4), julio-agosto de 2009.) http://dx.doi.org//10.1109/MS.2009.86.

EJERCICIOS

16.1. ¿Cuáles son los principales factores técnicos y no técnicos que impiden la reutilización de software? ¿Usted reutiliza mucho software? Si no es así, ¿por qué?

16.2. Sugiera por qué los ahorros en costo de reutilizar software existente no son proporcionales al tamaño de los componentes que se reutilizan.

16.3. Describa cuatro circunstancias donde desaconseje la reutilización de software.

16.4. Explique qué entiende por "inversión de control" en los frameworks de aplicación. Describa por qué este enfoque podría causar problemas si se integran dos sistemas separados que se crearon originalmente usando el mismo framework de aplicación.

16.5. Con el ejemplo del sistema de estación meteorológica descrito en los capítulos 1 y 7, sugiera una arquitectura de línea de productos para una familia de aplicaciones que se ocupen de la monitorización remota y la recolección de datos. Debe presentar su arquitectura como un modelo en capas, que muestre los componentes que puedan incluirse en cada nivel.

16.6. La mayoría del software de escritorio, como el software de procesamiento de textos, puede configurarse en algunas formas diferentes. Examine el software que usa regularmente y mencione las opciones de configuración para dicho software. Sugiera las dificultades que podrían enfrentar los usuarios para configurar el software. Si usa Microsoft Office u Open Office, son buenos ejemplos para usar en este ejercicio.

16.7. ¿Por qué numerosas compañías grandes eligieron sistemas ERP como la base para su sistema de información organizacional? ¿Qué problemas podrían surgir al implementar a gran escala un sistema ERP en una organización?

16.8. Identifique seis posibles riesgos que puedan surgir cuando los sistemas se construyen usando COTS. ¿Qué acciones recomendaría emprender a una compañía para reducir esos riesgos?

16.9. Exponga por qué se necesitan generalmente adaptadores cuando se construyen sistemas mediante la integración de productos COTS. Describa tres problemas prácticos que pudieran surgir al escribir software adaptador para vincular dos productos de aplicación COTS.

16.10. La reutilización de software plantea algunos conflictos de derechos de autor y propiedad intelectual. Si un cliente paga a un contratista de software para desarrollar un sistema, ¿quién tiene el derecho de reutilizar el código desarrollado? ¿El contratista de software tiene el derecho a usar dicho código como base para un componente genérico? ¿Qué mecanismos de pago se podrían usar para compensar a los proveedores de componentes reutilizables? Analice estos temas y otros conflictos éticos asociados con la reutilización de software.

REFERENCIAS

Baker, T. (2002). "Lessons Learned Integrating COTS into Systems". *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems*), Orlando, Fla:: Springer, 21–30.

Balk, L. D. y Kedia, A. (2000). "PPT: A COTS Integration Case Study". *Proc. Int. Conf. on Software Eng*., Limerick, Irlanda: ACM Press, 42–9.

Baumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. y Zullighoven, H. (1997). "Framework Development for Large Systems". *Comm*. ACM, **40** (10), 52–9.

Boehm, B. y Abts, C. (1999). "COTS Integration: Plug and Pray?" *IEEE Computer*, **32** (1), 135–38.

Brownsword, L. y Morris, E. (2003). "The Good News about COTS". http://www.sei.cmu.edu/news-at-sei/features/2003/1q03/feature-1-1q03.htm

Cusamano, M. (1989). "The Software Factory: A Historical Interpretation". *IEEE Software*, **6** (2), 23–30.

Fayad, M. E. y Schmidt, D. C. (1997). "Object-oriented Application Frameworks". *Comm*. ACM, **40** (10), 32–38.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.

Garlan, D., Allen, R. y Ockerbloom, J. (1995). "Architectural Mismatch: Why Reuse is so Hard". *IEEE Software*, **12** (6), 17–26.

Garlan, D., Allen, R. y Ockerbloom, J. (2009). "Architectural Mismatch: Why Reuse is Still so Hard". *IEEE Software*, **26** (4), 66–9.

Griss, M. L. y Wosser, M. (1995). "Making reuse work at Hewlett-Packard". *IEEE Software*, **12** (1), 105–7.

Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Calif.: O'Reilly and Associates.

Jacobson, I., Griss, M. y Jonsson, P. (1997). *Software Reuse*. Reading, Mass.: Addison-Wesley.

Matsumoto, Y. (1984). "Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels". *IEEE. Trans. on Software Engineering*, **SE-10** (5), 502–12.

McIlroy, M. D. (1968). "Mass-produced software components". *Proc. NATO Conf. on Software Eng.*, Garmisch, Alemania: Springer-Verlag.

Nuseibeh, B. (1997). "Ariane 5: Who Dunnit?" *IEEE Software*, **14** (3), 15–6.

O'Leary, D. E. (2000). *Enterprise Resource Planning Systems: Systems, Life Cycle, Electronic Commerce and Risk*. Cambridge, UK: Cambridge University Press.

Pfarr, T. y Reis, J. E. (2002). "The Integration of COTS/GOTS within NASA's HST Command and Control System". *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, Fla.: Springer, 209–21.

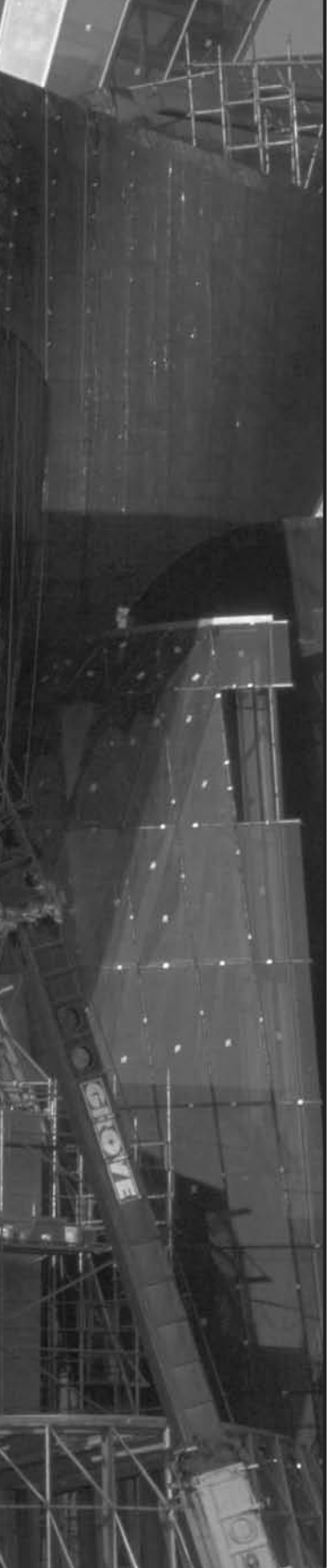
Schmidt, D. C. (1997). "Applying design patterns and frameworks to develop object-oriented communications software". En *Handbook of Programming Languages, Vol. 1.* (ed.). Nueva York: Macmillan Computer Publishing.

Schmidt, D. C., Gokhale, A. y Natarajan, B. (2004). "Leveraging Application Frameworks". *ACM Queue*, **2** (5 (julio/agosto)), 66–75.

Scott, J. E. (1999). "The FoxMeyer Drug's Bankruptcy: Was it a Failure of ERP". Proc. *Association for Information Systems 5th Americas Conf. on Information Systems*, Milwaukee, WI.

Torchiano, M. y Morisio, M. (2004). "Overlooked Aspects of COTS-Based Development". *IEEE Software*, **21** (2), 88–93.

Tracz, W. (2001). "COTS Myths and Other Lessons Learned in *Component-Based Software Development*". En *Component-based Software Engineering*. Heineman, G. T. y Councill, W. T. (ed.). Boston: Addison-Wesley, 99–112.



17

Ingeniería de software basada en componentes

Objetivos

El objetivo de este capítulo es describir un enfoque de la reutilización de software basado en la composición de componentes estandarizados de reutilización. Al estudiar este capítulo:

- sabrá que la ingeniería de software basada en componentes se ocupa del desarrollo de componentes estandarizados con base en un modelo de componentes, y la organización de éstos en sistemas de aplicación;
- entenderá qué es un componente y un modelo de componentes;
- conocerá las principales actividades en el proceso CBSE para reutilización y el proceso CBSE con reutilización;
- comprenderá algunas de las dificultades que surgen durante el proceso de composición de componentes.

Contenido

17.1 Componentes y modelos de componentes
17.2 Procesos CBSE
17.3 Composición de componentes

Como se explicó en el capítulo 16, numerosos y nuevos sistemas empresariales se desarrollan ahora al configurar sistemas comerciales COTS. Sin embargo, cuando una compañía no puede usar un sistema COTS porque no cubre sus requerimientos, el software que se requiere debe desarrollarse especialmente. Para el software personalizado, la ingeniería de software basada en componentes es una forma efectiva orientada a la reutilización para desarrollar nuevos sistemas empresariales.

La ingeniería de software basada en componentes (CBSE, por las siglas de *Component-Based Software Engineering*) surgió a finales de la década de 1990 como un enfoque al desarrollo de sistemas de software basado en la reutilización de componentes de software. Su creación fue motivada por la frustración de los diseñadores al percatarse de que el desarrollo orientado a objetos no conducía a una reutilización extensa, como se había sugerido originalmente. Las clases de objetos individuales eran muy detalladas y específicas y con frecuencia tenían que acotarse con una aplicación al momento de compilar. Para usarlas, se debe tener conocimiento detallado de las clases, y por lo general esto significa tener el código fuente del componente. Por consiguiente, era casi imposible vender o distribuir objetos como componentes de reutilización individuales.

Los componentes son abstracciones de alto nivel en comparación con los objetos y se definen mediante sus interfaces. Por lo general, son más grandes que los objetos individuales y todos los detalles de implementación se ocultan a otros componentes. La CBSE es el proceso de definir, implementar e integrar o componer los componentes independientes e imprecisos en los sistemas. Se ha constituido en un importante enfoque de desarrollo de software porque los sistemas de software son cada vez más amplios y complejos. Los clientes demandan un software más confiable que se entregue e implemente más rápidamente. La única forma de enfrentar la complejidad y entregar un mejor software con mayor rapidez es reutilizar en lugar de implementar una vez más los componentes de software.

Los fundamentos de la ingeniería de software basada en componentes son:

1. Componentes independientes que se especifican por completo mediante sus interfaces. Debe existir una separación clara entre la interfaz del componente y su implementación. Esto significa que la implementación de un componente puede sustituirse por otra, sin cambiar otras partes del sistema.

2. Estándares de componentes que facilitan la integración de éstos. Tales estándares se incrustan en un modelo de componentes. Definen, a un nivel mínimo, cómo deben especificarse las interfaces de componentes y cómo se comunican estos últimos. Algunos modelos van más allá y definen las interfaces que deben implementarse por todos los componentes integrantes. Si los componentes se conforman a los estándares, entonces su ejecución es independiente de su lenguaje de programación. Los componentes escritos en diferentes lenguajes pueden integrarse en el mismo sistema.

3. Middleware que brinda soporte de software para integración de componentes. Para hacer que componentes independientes distribuidos trabajen en conjunto, es necesario soporte de middleware que maneje las comunicaciones de componentes. El middleware para soporte de componentes maneja eficientemente los conflictos de bajo nivel y permite enfocarse en problemas relacionados con la aplicación. Además, el middleware para soporte de componentes puede brindar apoyo para la asignación de recursos, la gestión de transacciones, la seguridad y concurrencia.

4. Un proceso de desarrollo que se engrana con la ingeniería de software basada en componentes. Usted necesita un proceso de desarrollo que permita la evolución de

Problemas con CBSE

La CBSE es ahora un enfoque de mantenimiento a la ingeniería de software: es una buena forma de construir sistemas. Sin embargo, cuando se usa como enfoque para la reutilización, los problemas se relacionan con la fiabilidad y certificación de los componentes, los compromisos de requerimientos y la predicción de las propiedades de los componentes, en especial al integrarse con otros.

http://www.SoftwareEngineering-9.com/Web/CBSE/problems.html

requerimientos, dependiendo de la funcionalidad de los componentes disponibles. En la sección 17.2 se estudian los procesos de desarrollo CBSE.

El desarrollo basado en componentes implica una buena práctica de ingeniería de software. Tiene sentido diseñar un sistema mediante componentes, incluso si usted debe desarrollarlos en vez de reutilizarlos. En la base de la CBSE existen firmes principios de diseño que apoyan la construcción de software comprensible y mantenible:

1. Los componentes son independientes, de manera que sus ejecuciones no interfieren entre sí. Se ocultan los detalles de la implementación. La implementación de componentes puede cambiar sin afectar al resto del sistema.
2. Los componentes se comunican a través de interfaces bien definidas. Si dichas interfaces se mantienen, es posible sustituir un componente por otro, lo que ofrece funcionalidad adicional o mayor.
3. Las infraestructuras de componentes ofrecen varios servicios estándar que pueden usarse en sistemas de aplicación. Esto reduce la cantidad de código nuevo que debe desarrollarse.

La motivación inicial para la CBSE fue la necesidad de brindar apoyo tanto a la ingeniería de software de reutilización como a la distribuida. Un componente se considera como un elemento de un sistema de software al que se podría acceder, mediante un mecanismo llamado procedimiento remoto, por parte de otros componentes que se ejecutan en computadoras independientes. Cada sistema que reutiliza un componente debe incorporar su propia copia de dicho componente. Esta idea de componente se extiende a la noción de objetos distribuidos, como se define en modelos de sistemas distribuidos, tales como la especificación CORBA (Pope, 1997). Para apoyar esta visión de componente, se han desarrollado muchos y diferentes protocolos y estándares, como el Enterprise Java Beans (EJB) de Sun, COM y .NET de Microsoft, y CCM de CORBA (Lau y Wang, 2007).

En la práctica, estos estándares múltiples obstaculizan la aceptación de CBSE. Era imposible que componentes desarrollados mediante diferentes enfoques funcionaran juntos. Los componentes diseñados para distintas plataformas, tales como .NET o J2EE, no pueden interoperar. Más aún, los estándares y protocolos propuestos se consideraban complejos y difíciles de entender. Esto también se presentaba como una barrera para su adopción.

En respuesta a tales problemas, se desarrolló la noción de componente como un servicio, y se propusieron estándares para apoyar la ingeniería de software orientada a servicios.

La diferencia más significativa entre un componente como servicio y la noción original de componente es que los servicios son entidades independientes externas a un programa que los usa. Cuando se construye un sistema orientado al servicio, se hace referencia al servicio externo en vez de incluir en su sistema una copia de dicho servicio.

Por lo tanto, la ingeniería de software orientada al servicio, que se estudia en el capítulo 19, es un tipo de ingeniería de software basada en componentes. Utiliza una noción más simple de componente que el propuesto originalmente en CBSE. Los estándares impulsaron esto desde el comienzo. En situaciones en que no es práctica la reutilización basada en COTS, la CBSE orientada al servicio se convierte en el enfoque dominante para el desarrollo de sistemas empresariales.

17.1 Componentes y modelos de componentes

En la comunidad CBSE existe un acuerdo general de que un componente es una unidad de software independiente que puede organizarse con otros componentes para crear un sistema de software. Sin embargo, más allá de eso, hay quienes proponen definiciones variables de un componente de software. Council y Heineman (2001) definen un componente como:

> *Un elemento de software que se conforma a un modelo de componentes estándar y puede desplegarse y componerse independientemente sin modificación, de acuerdo con un estándar de composición.*

En esencia, esta definición se basa en estándares, de manera que una unidad de software conformada a dichos estándares es un componente. No obstante, Szyperski (2002) no menciona estándares en su definición de componente; en vez de ello, se enfoca en características clave de los componentes:

> *Un componente de software es una unidad de composición con interfaces especificadas contractualmente y sólo con dependencias de contexto explícitas. Un componente de software puede implementarse de manera independiente y está sujeto a composición por terceras partes.*

Ambas definiciones se basan en la noción de componente como elemento que se incluye en un sistema, más que en un servicio al que hace referencia el sistema. Con todo, también son compatibles con la idea de un servicio como un componente.

Szyperski establece además que un componente no tiene estado observable externo. Esto significa que las copias de los componentes son indistinguibles. No obstante, algunos modelos de componentes, tales como el modelo Enterprise Java Beans, permiten componentes de estado, así que no corresponden a la definición de Szyperski. Aunque los componentes sin estado son más simples de usar, existen algunos sistemas en que los componentes de estado son más convenientes y reducen la complejidad del sistema.

Lo que tienen en común las definiciones anteriores es que concuerdan en que los componentes son independientes, y los consideran la unidad fundamental de composición en un sistema. La visión del autor es que puede obtenerse una mejor definición de

Característica del componente	Descripción
Estandarizado	Estandarización de componentes significa que un componente utilizado durante un proceso CBSE debe ajustarse a un modelo de componentes estándar. Este modelo puede definir interfaces de componentes, metadatos de componentes, documentación, composición e implementación.
Independiente	Un componente debe ser independiente; debe ser factible componerlo e implementarlo sin usar otros componentes específicos. En situaciones en que el componente necesita brindar servicios externos, esto debería plantearse claramente en una especificación de interfaz de "requiere".
Componible	Para que un componente sea componible, todas las interacciones externas deben tener lugar mediante interfaces definidas públicamente. Además, debe permitir acceso externo a información acerca de sí mismo, así como a sus métodos y atributos.
Implementable	Para que sea implementable, un componente debe estar autocontenido. Debe ser capaz de ejecutarse como entidad independiente en una plataforma de componente que permita una implementación del modelo de componentes. Por lo general, esto significa que el componente es binario y no tiene que compilarse antes de su implementación. Si un componente se implementa como servicio, no tiene que implementarse por parte de un usuario de un componente. En vez de ello, se implementa por parte del proveedor del servicio.
Documentado	Los componentes deben implementarse por completo, para que los usuarios potenciales puedan decidir si los componentes cumplen o no sus necesidades. Debe especificarse la sintaxis y, de manera ideal, la semántica de todas las interfaces de componente.

Figura 17.1 Características de los componentes

componente al combinar estas propuestas. La figura 17.1 muestra lo que el autor considera como las características esenciales de un componente, como se usa en CBSE.

Una forma útil de pensar en un componente es como un proveedor de uno o más servicios. Cuando un sistema necesita un servicio, llama a un componente que brinde 0programación que se usó para desarrollarlo. Por ejemplo, un componente en un sistema de biblioteca puede ofrecer un servicio de búsqueda que permita a los usuarios examinar diferentes catálogos de la biblioteca. Un componente que convierte de un formato gráfico a otro (por ejemplo, TIFF a JPEG) ofrece un servicio de conversión de datos, etcétera.

Visualizar un componente como un proveedor de servicio pone de relieve dos características críticas de un componente de reutilización:

1. El componente es una entidad ejecutable independiente definida mediante sus interfaces. Para usarlo no se necesita conocimiento alguno de su código fuente. Puede hacerse referencia a él como un servicio externo o incluirse directamente en un programa.
2. Los servicios ofrecidos por un componente se ponen a disposición mediante una interfaz, y todas las interacciones por dicha interfaz. La interfaz del componente se expresa en términos de operaciones parametrizadas y nunca se expone su estado interno.

Componente y objetos

Los componentes se implementan con frecuencia en lenguajes orientados a objetos y, en algunos casos, el acceso a la interfaz "proporciona" de un componente se realiza a través de solicitudes de método. Sin embargo, los componentes y las clases de objetos no son lo mismo. A diferencia de las clases de objetos, los componentes se implementan de manera independiente, no son tipos definidos, sino independientes del lenguaje y se basan en un modelo de componentes estándar.

http://www.SoftwareEngineering-9.com/Web/CBSE/objects.html

Los componentes tienen dos interfaces relacionadas, como se muestra en la figura 17.2. Dichas interfaces reflejan los servicios que proveen los componentes y los servicios que el componente requiere para ejecutarse correctamente:

- La interfaz "proporciona" define los servicios que ofrece el componente. En esencia, esta interfaz es el componente API. Define los métodos que puede solicitar el usuario del componente. En un diagrama de componentes UML, la interfaz "proporciona" para un componente se indica mediante un círculo al final de una línea desde el icono del componente.
- La interfaz "requiere" especifica qué servicios deben ofrecer otros componentes en el sistema para que un componente opere correctamente. Si no están disponibles, entonces el componente no funcionará. Esto no compromete la independencia o el carácter implementable de un componente, porque la interfaz "requiere" no define cómo deben proporcionarse dichos servicios. En el UML, el símbolo para una interfaz "requiere" es un semicírculo al final de una línea desde el icono del componente. Observe que los iconos de las interfaces "proporciona" y "requiere" pueden encajar como una articulación de rótula.

Para ilustrar estas interfaces, la figura 17.3 muestra un modelo de componente que se diseñó para recopilar e intercalar información desde un arreglo de sensores. Se ejecuta de manera autónoma para recopilar datos durante un cierto tiempo y, a petición, proporciona datos intercalados a un componente que lo solicite. La interfaz "proporciona" incluye métodos para agregar, remover, iniciar, detener y probar los sensores. El método report regresa los datos del sensor que se recopilaron, y el método listAll brinda información de los sensores unidos. Aunque esto no se muestra aquí, dichos métodos tienen parámetros asociados que especifican los identificadores del sensor, así como sus ubicaciones, etcétera.

La interfaz "requiere" se usa para conectar el componente a los sensores. Supone que los sensores tienen una interfaz de datos, a los que se accede a través de sensorData, y

Figura 17.2 Interfaces de componentes

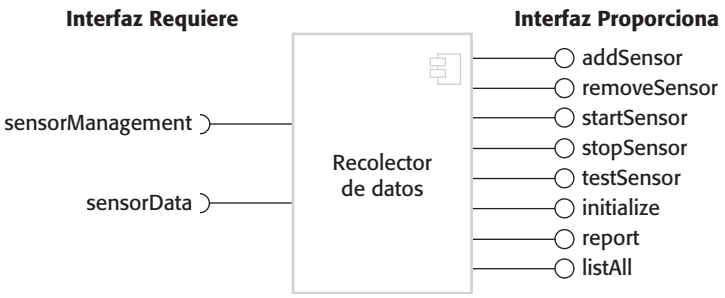


Figura 17.3 Modelo de componente recopilador de datos

una interfaz de gestión, a la que se accede a través de sensorManagement. Esta interfaz se diseñó para conectarse a diferentes tipos de sensor, así que no incluye operaciones de sensor específicas, tales como Test, provideReading, etcétera. En vez de ello, los comandos que usa un tipo específico de sensor se incrustan en una cadena, que es un parámetro para las operaciones en la interfaz "requiere". Los componentes de adaptador analizan gramaticalmente (*parse*) esta cadena y traducen los comandos incrustados en la interfaz de control específica de cada tipo de sensor. Más adelante en este capítulo se estudia el uso de los adaptadores, y se muestra cómo los componentes recolectores de datos se vinculan con un sensor (figura 17.12).

Una diferencia crucial entre un componente como servicio externo y un componente como elemento de programa es que los servicios son entidades independientes por completo. No tienen una interfaz "requiere". Diferentes programas pueden usar dichos servicios sin necesidad de implementar soporte adicional requerido por el servicio.

17.1.1 Modelos de componentes

Un modelo de componentes es una definición de estándares para implementación, documentación y despliegue de componentes. Estos estándares se establecen con la finalidad de que los desarrolladores de componentes se aseguren de que éstos pueden interoperar. También funcionan para proveedores de infraestructuras de ejecución de componentes que ofrecen middleware para apoyar la ejecución de componentes. Se han propuesto muchos modelos de componentes, pero ahora los modelos más importantes son el modelo WebServices, el modelo Enterprise Java Beans (EJB) de Sun, y el modelo .NET de Microsoft (Lau y Wang, 2007).

Weinreich y Sametinger (2001) analizan los elementos básicos de un modelo ideal de componentes. En la figura 17.4 se resumen esos elementos de modelo. Este diagrama muestra que los elementos de un modelo de componentes definen las interfaces de componentes, la información que necesita usar el componente en un programa y cómo debe implementarse un componente:

1. *Interfaces* Los componentes se definen al especificar sus interfaces. El modelo de componentes especifica cómo deben definirse las interfaces y los elementos, tales como los nombres de operación, los parámetros y las excepciones que deben incluirse en la definición de la interfaz. El modelo también debe especificar el lenguaje usado para definir las interfaces de componentes. Para servicios Web, éste es

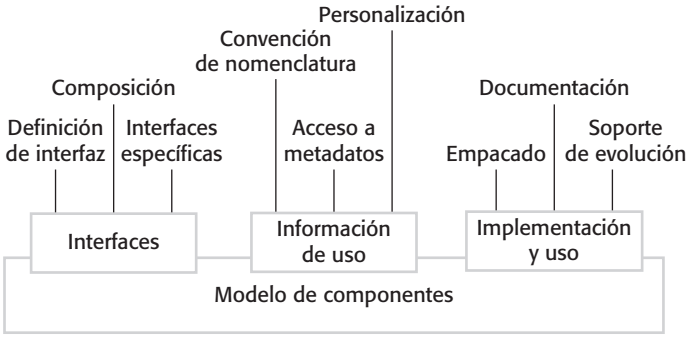


Figura 17.4 Elementos básicos de un modelo de componentes

WSDL, que se estudia en el capítulo 19; EJB es específico de Java, de manera que se usa Java como el lenguaje de definición de interfaz; en .NET, las interfaces se definen con el Common Intermediate Language (CIL, lenguaje intermedio común). Algunos modelos de componentes requieren interfaces específicas que deben definirse por un componente. Se usan para componer el componente con la infraestructura de modelo de componentes, que ofrece servicios estandarizados, tales como seguridad y gestión de transacción.

2. *Uso* Para que los componentes se distribuyan y se acceda a ellos de manera remota, deben tener un nombre único asociado. Éste debe ser totalmente único, por ejemplo, en EJB se genera un nombre jerárquico con la raíz basada en un nombre de dominio de Internet. Los servicios tienen un URI único (Uniform Resource Identifier, esto es, un identificador de recursos uniforme).

Los metadatos de componente son datos acerca del componente en sí, tales como información acerca de sus interfaces y atributos. Los metadatos son importantes porque permiten a los usuarios del componente determinar qué servicios se proporcionan y requieren. Las implementaciones de modelos de componentes por lo general incluyen formas específicas (tales como el uso de una interfaz de reflexión en Java) para acceder a los metadatos de este componente.

Los componentes son entidades genéricas y, cuando se implementan, deben configurarse para ajustarse en un sistema de aplicación. Por ejemplo, se podría configurar el componente recolector de datos (figura 17.3) al definir el número máximo de sensores en un arreglo. Por lo tanto, el modelo de componentes puede especificar cómo pueden personalizarse los componentes binarios para un entorno de implementación particular.

3. *Implementación* El modelo de componentes incluye una especificación de cómo deben empacarse los componentes para su implementación como entidades ejecutables independientes. Puesto que los componentes son entidades independientes, deben empacarse con todo el software de soporte que no proporcione la infraestructura de componente, o que no esté definido en una interfaz "requiere". La información de implementación incluye información sobre el contenido de un paquete y su organización binaria.

Inevitablemente, conforme surjan nuevos requerimientos, los componentes deberán cambiarse o sustituirse. Por consiguiente, el modelo de componentes puede incluir

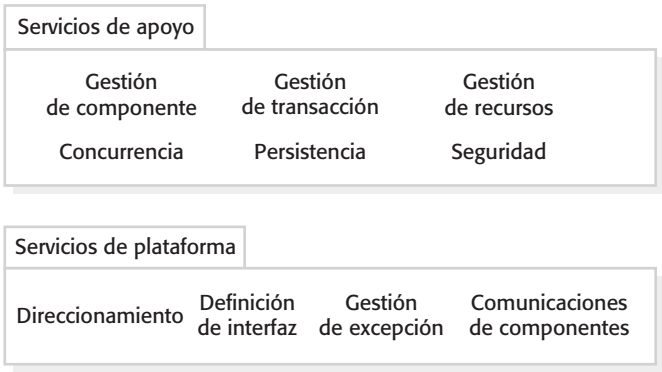


Figura 17.5 Servicios middleware definidos en un modelo de componentes

reglas que rigen cuándo y cómo se permite la sustitución de componentes. Finalmente, el modelo de componentes puede definir la documentación de componentes que deba producirse. Esto se usa para encontrar el componente y decidir si es adecuado.

Para componentes que se implementen como unidades de programa y no como servicios externos, el modelo de componentes establece los servicios a ofrecer por parte del middleware que apoye los componentes en ejecución. Weinreich y Sametinger (2001) usan la analogía de un sistema operativo para explicar los modelos de componentes. Un sistema operativo brinda un conjunto de servicios genéricos que pueden usar las aplicaciones. La implementación de un modelo de componentes ofrece servicios compartidos comparables para los componentes. La figura 17.5 muestra algunos de los servicios que puede ofrecer una implementación de un modelo de componentes.

Los servicios brindados por una implementación de modelo de componentes se dividen en dos categorías:

1. Servicios de plataforma, los cuales permiten a los componentes comunicarse e interoperar en un entorno distribuido. Se trata de servicios fundamentales que deben estar disponibles en todos los sistemas basados en componentes.
2. Servicios de apoyo, que son servicios comunes que probablemente requieran muchos componentes diversos. Por ejemplo, numerosos componentes requieren autenticación para garantizar que el usuario de los servicios del componente está autorizado. Tiene sentido ofrecer un conjunto estándar de servicios middleware para uso de todos los componentes. Esto reduce los costos del desarrollo de componentes y permite evitar las incompatibilidades potenciales de los componentes.

El middleware implementa los servicios de componentes y ofrece interfaces a dichos servicios. Para usar los servicios ofrecidos por la infraestructura de un modelo de componentes, se puede considerar a los componentes como implementados en un "contenedor". Un contenedor es una implementación de los servicios de apoyo más una definición de las interfaces que debe proporcionar un componente para integrarlo con el contenedor. Incluir el componente en el contenedor significa que el componente puede ingresar a los servicios de apoyo, y el contenedor puede acceder a las interfaces de componente. Cuando se usan, otros componentes no acceden directamente a las interfaces del componente; en vez de

ello, se accede a éstas a través de una interfaz de contenedor que recurre al código para acceder a la interfaz del componente embebido.

Los contenedores son grandes y complejos, y cuando un componente se implementa en un contenedor se consigue acceso a todos los servicios middleware. Sin embargo, los componentes simples tal vez no necesiten todas las instalaciones que ofrece el middleware de apoyo. Por lo tanto, es un poco diferente el enfoque que se toma en los servicios Web para el suministro de un servicio común. En el caso de los servicios Web se han definido estándares para servicios comunes como la gestión de transacciones y la seguridad, y dichos estándares se implementan como librerías de programa. Si usted implementa un componente de servicio, sólo utiliza los servicios comunes que necesita.

17.2 Procesos CBSE

Los procesos CBSE son procesos de software que brindan soporte a la ingeniería de software basada en componentes. Toman en cuenta las posibilidades de reutilización y las diferentes actividades de proceso implicadas en el desarrollo y uso de componentes reutilizables. La figura 17.6 (Kotonya, 2003) ilustra un panorama de los procesos en CBSE. Al nivel más alto, existen dos tipos de procesos CBSE:

1. *Desarrollo para reutilización* Este proceso se ocupa del desarrollo de componentes o servicios que se reutilizarán en otras aplicaciones. Por lo regular, implica la generalización de los componentes existentes.
2. *Desarrollo con reutilización* Éste es el proceso para desarrollar nuevas aplicaciones usando los componentes y servicios existentes.

Dichos procesos tienen diferentes objetivos y, por consiguiente, incluyen distintas actividades. En el proceso de desarrollo para reutilización, el objetivo es producir uno o más componentes reutilizables. Usted conoce los componentes con los que trabajará y tiene acceso a su código fuente para generalizarlos. En el desarrollo con reutilización, no sabe cuáles componentes están disponibles, así que necesita descubrir dichos componentes y diseñar un sistema para utilizarlos de la manera más efectiva. No puede tener acceso al código fuente del componente.

En la figura 17.6 se observa que los procesos básicos de CBSE con y para reutilización tienen procesos de soporte que se ocupan de la adquisición, gestión y certificación de componentes:

1. Adquisición de componentes es el proceso de adquirir componentes para reutilización o desarrollo en un componente reutilizable. Puede implicar el acceso a componentes o servicios desarrollados localmente, o encontrar dichos componentes en una fuente externa.
2. La gestión de componentes se ocupa de la gestión de los componentes de reutilización de una compañía, asegurándose de que estén adecuadamente catalogados, almacenados y dispuestos para reutilizarse.

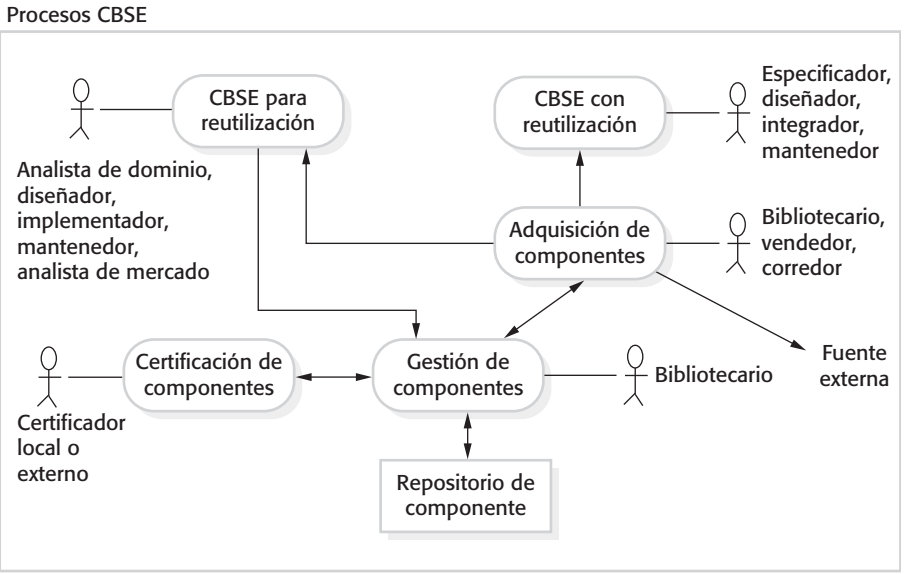


Figura 17.6 Procesos CBSE

3. Certificación de componentes es el proceso de comprobar un componente y asegurarse de que cumple su especificación.

Los componentes que conserva una organización pueden almacenarse en un repositorio de componentes que incluya tanto los componentes como la información de su uso.

17.2.1 CBSE para reutilización

La CBSE para reutilización es el proceso de desarrollar componentes reutilizables y ponerlos a disposición para reutilizarlos a través de un sistema de gestión de componentes. La visión de los primeros defensores de la CBSE (Szyperski, 2002) fue que se desarrollaría un floreciente mercado de componentes. Habría proveedores especializados de componentes y vendedores que organizarían la venta de componentes de diferentes desarrolladores. Los desarrolladores de software comprarían componentes para incluir en un sistema o pagarían por servicios conforme los utilizaran. Sin embargo, esta visión no se ha materializado. Existen relativamente pocos proveedores de componentes, y la compra de componentes es poco común. Al momento de escribir este texto, el mercado de servicios también está poco desarrollado, aunque hay predicciones de que se expandirá significativamente durante los próximos años.

En consecuencia, es más probable que la CBSE para reutilización tenga lugar dentro de una organización que realizó un compromiso con la ingeniería de software dirigida por la reutilización. En esas circunstancias, se desea aprovechar los activos de software que se desarrollaron en diferentes partes de la compañía. Sin embargo, tales componentes desarrollados internamente por lo general no son reutilizables sin cambios. Con frecuencia incluyen características e interfaces específicas de aplicación que es improbable que se requieran en otros programas donde se reutilice el componente.

Para elaborar componentes de reutilización, deben adaptarse y extenderse componentes específicos de aplicación para crear versiones más genéricas y, por lo tanto, más reutilizables. Evidentemente, esta adaptación tiene un costo asociado. Así que primero habrá que decidir si es probable que un componente se reutilice y, segundo, si los ahorros de costo de la futura reutilización justifican los costos de hacer reutilizable al componente.

Para responder la primera de estas preguntas, se debe decidir si el componente implementa o no una o más abstracciones de dominio estables. Las abstracciones de dominio estables son elementos fundamentales del dominio de aplicación que cambian lentamente. Por ejemplo, en un sistema bancario, las abstracciones de dominio pueden incluir cuentas, poseedores de cuentas y enunciados. En un sistema de administración de hospitales, las abstracciones de dominio pueden incluir pacientes, tratamientos y enfermeros. En ocasiones, a tales abstracciones de dominio se les llama "objetos empresariales". Si el componente es una implementación de una abstracción de dominio o un grupo de objetos empresariales relacionados comúnmente usados, tal vez puedan reutilizarse.

Para responder la pregunta acerca de la efectividad en términos de costo, habrá que valorar los costos de los cambios que se requieren para hacer reutilizable al componente. Éstos son los costos de la documentación y validación del componente, y los asociados con el hecho de hacerlo más genérico. Los cambios que se pueden hacer a un componente para volverlo más reutilizable incluyen:

- eliminar métodos específicos de aplicación;
- cambiar los nombres para hacerlos más generales;
- agregar métodos para brindar cobertura funcional más completa;
- hacer manejadores de excepción consistentes para todos los métodos;
- adicionar una interfaz de "configuración" para permitir la adaptación de los componentes a diferentes situaciones de uso;
- integrar los componentes requeridos para aumentar la independencia.

El problema del manejo de excepción es particularmente difícil. Los componentes no deben manejar las excepciones por sí mismos, porque cada aplicación tendrá sus propios requerimientos para manejo de excepción. En vez de ello, el componente debe definir qué excepciones pueden surgir y éstas deben publicarse como parte de la interfaz. Por ejemplo, un componente simple que implemente una estructura de datos en pila debe detectar y publicar excepciones de desbordamiento (*overflow*) de pilas y cuando se quiere extraer de la pila vacía (*underflow*). Sin embargo, en la práctica, existen dos problemas con esto:

1. Publicar todas las excepciones conduce a interfaces infladas que son difíciles de entender. Esto podría alejar a usuarios potenciales del componente.
2. La ejecución del componente puede depender del manejo de excepciones locales, y cambiar esto tal vez tenga serias implicaciones para la funcionalidad del componente.

Mili y sus colaboradores (2002) discuten formas de estimar los costos de hacer reutilizable un componente y los rendimientos de dicha inversión. Los beneficios de la reutilización antes de volver a desarrollar un componente no son simplemente ganancias

en términos de productividad. También existen ganancias de calidad, porque un componente de reutilización debe ser más confiable, y hay ganancias de tiempo de salida al mercado. Se trata de un aumento en los rendimientos que se acrecientan al implementar el software más rápidamente. Mili y sus colaboradores muestran varias fórmulas para estimar dichas ganancias, como el modelo COCOMO que se estudia en el capítulo 23 (Boehm *et al*., 2000). Sin embargo, los parámetros de tales fórmulas son difíciles de evaluar con precisión, y las fórmulas deben adaptarse a circunstancias locales, lo que las hace más difíciles de usar. Es posible que pocos administradores de proyecto de software usen estos modelos para estimar el rendimiento sobre la inversión a partir de la reutilización de componentes.

Desde luego, si un componente es susceptible de reutilización, o no, depende de su dominio de aplicación y su funcionalidad. Conforme se agrega generalidad a un componente, aumenta la probabilidad de su reutilización. No obstante, esto por lo regular significa que el componente tiene más operaciones y más complejidades, las cuales lo hacen difícil de entender y emplear.

Sin embargo, esto es un intercambio inevitable entre la reutilización y el uso de un componente. Para hacer reutilizable un componente, usted deberá proporcionar un conjunto de interfaces genéricas con operaciones que incluyan todas las formas que el componente podría usar. Hacer un componente utilizable significa ofrecer una interfaz simple y mínima, que sea fácil de entender. La reutilización agrega complejidad y, por eso, se dificulta el hecho de entender un componente. Por consiguiente, es más difícil decidir cuándo y cómo reutilizar dicho componente. En consecuencia, cuando diseñe un componente de reutilización, debe encontrar un compromiso entre la generalidad y comprensibilidad.

Una fuente potencial de componentes está constituida por los sistemas heredados existentes. Como se estudió en el capítulo 9, se trata de sistemas que cumplen una importante función empresarial, pero que están escritos con tecnologías de software obsoletas. Por ello, tal vez sea difícil usarlos con sistemas nuevos. Sin embargo, si estos sistemas antiguos se convierten a componentes, su funcionalidad puede reutilizarse en nuevas aplicaciones.

Desde luego, estos sistemas heredados no tienen normalmente interfaces "requiere" y "proporciona" bien definidas. Para hacer reutilizables dichos componentes, debe crear una envoltura (*wrapper*) que defina las interfaces de componente. La envoltura oculta la complejidad del código subyacente y ofrece una interfaz para que los componentes externos accedan a los servicios que se brindan. Aunque esta envoltura es una pieza de software bastante compleja, con frecuencia el costo de desarrollarlo es mucho menor que el costo de volver a implementar el sistema heredado. En el capítulo 19 se examina con más detalle este enfoque, y se explica cómo puede accederse a las características del sistema heredado a través de servicios.

Una vez que se ha desarrollado y probado un componente o servicio de reutilización, entonces debe gestionarse para una reutilización en el futuro. La gestión implica decidir cómo clasificar el componente de forma que pueda descubrirse, hacer al componente disponible ya sea en un repositorio o como servicio, mantener información acerca de su uso, y hacer un seguimiento de las diferentes versiones del componente. Si el componente es de fuente abierta, se puede hacer disponible en un repositorio público como Sourceforge. Si se pretende utilizarlo en una compañía, entonces se puede usar un sistema de depósito interno.

Una compañía con un programa de reutilización puede realizar alguna forma de certificación de componente antes de que éste se encuentre disponible para su reutilización.

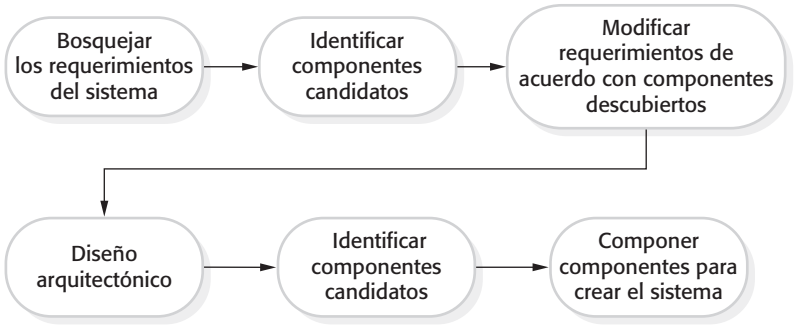


Figura 17.7 CBSE con reutilización

La certificación significa que alguien, aparte del desarrollador, verifica la calidad del componente. Se prueba el componente y se certifica que alcanza un estándar de calidad aceptable, antes de ponerlo a disposición para su reutilización. Sin embargo, éste puede ser un proceso costoso y muchas compañías simplemente dejan las pruebas y la comprobación de la calidad en manos de los desarrolladores del componente.

17.2.2 CBSE con reutilización

La reutilización exitosa de componentes requiere un proceso de desarrollo ajustado a CBSE. La CBSE con un proceso de reutilización debe incluir actividades que encuentren e integren componentes reutilizables. La estructura de tal proceso se trató en el capítulo 2, y la figura 17.7 muestra las principales actividades dentro de ese proceso. Algunas de las actividades dentro de este proceso, como el descubrimiento inicial de los requerimientos de usuario, se realizan en la misma forma que en otros procesos de software. Sin embargo, las diferencias esenciales entre CBSE con reutilización y procesos de software para desarrollo de software original son:

1. Los requerimientos del usuario inicialmente se desarrollan en bosquejos y no en detalle, y se alienta a las partes interesadas a ser tan flexibles como sea posible para definir sus requerimientos. Los requerimientos que son demasiado específicos limitan el número de componentes que pueden satisfacer dichos requerimientos. Sin embargo, a diferencia del desarrollo incremental, se necesita un conjunto completo de requerimientos para poder identificar tantos componentes como sea posible para reutilización.
2. Los requerimientos se afinan y modifican oportunamente durante el proceso, dependiendo de los componentes disponibles. Si los requerimientos del usuario no pueden cumplirse a partir de los componentes disponibles, se deberán analizar los requerimientos relacionados que puedan cumplirse. Los usuarios tal vez tengan voluntad de cambiar su mentalidad si esto significa entrega de sistema a menor costo o más rápidamente.
3. Después de diseñar la arquitectura del sistema, hay una actividad adicional de búsqueda de componentes y clarificación de diseño. Algunos componentes aparentemente utilizables quizá resulten inadecuados o no funcionen como es debido con otros componentes elegidos. Aunque no se ilustra en la figura 17.7, esto implica que pueden ser necesarios posteriores cambios de requerimientos.

Figura 17.8 Proceso de identificación de componentes

4. El desarrollo es un proceso de composición en que se integran los componentes descubiertos. Esto implica integrar los componentes con la infraestructura del modelo de componentes y, con frecuencia, desarrollar adaptadores que reconcilien las interfaces de componentes incompatibles. Desde luego, también puede requerirse funcionalidad adicional por encima de la que ofrecen los componentes de reutilización.

La etapa de diseño arquitectónico es particularmente importante. Jacobson y sus colaboradores (1997) descubrieron que definir una arquitectura robusta es crucial para tener éxito en la reutilización. Durante la actividad de diseño arquitectónico, se puede elegir un modelo de componentes y una plataforma de implementación. Sin embargo, muchas compañías tienen una plataforma de desarrollo estándar (por ejemplo, .NET), así que el modelo de componentes está predeterminado. Como se estudió en el capítulo 6, en esta etapa se establece también la organización de alto nivel del sistema y se toman decisiones acerca de la distribución y el control del sistema.

Una actividad que es única para el proceso CBSE es identificar los componentes o servicios candidatos para reutilización. Esto implica algunas actividades específicas, como se muestra en la figura 17.8. Inicialmente, el enfoque debe estar en la búsqueda y selección. Es necesario cerciorarse de que hay componentes disponibles para cumplir los requerimientos. Desde luego, se debe hacer una comprobación inicial de que el componente es adecuado, aunque es posible que no se requieran pruebas detalladas. En la última etapa, después de diseñada la arquitectura del sistema, debe dedicarse más tiempo a la validación de componentes. Hay que estar seguros de que los componentes identificados son realmente adecuados para su aplicación; si no, entonces habrá que repetir los procesos de búsqueda y selección.

El primer paso en la identificación de los componentes es buscar aquellos que estén disponibles localmente o con proveedores confiables. Como se dijo en la sección anterior, existen relativamente pocos vendedores de componentes, y por eso usted tiene más probabilidades de buscar los componentes desarrollados en su propia compañía. Las compañías de desarrollo de software pueden construir su propia base de datos de componentes de reutilización sin los riesgos inherentes de usar componentes de proveedores externos. Alternativamente, es posible buscar librerías de código disponibles en la Web, como Sourceforge o Google Code, para saber si está disponible código fuente para el componente que se necesita. Si usted busca servicios, en ese caso están disponibles varios motores de búsqueda Web especializados que pueden descubrir servicios Web públicos.

Una vez que el proceso de búsqueda permite identificar los posibles componentes, se deben seleccionar componentes candidatos para su valoración. En algunos casos, ésta será una labor sencilla. Los componentes en la lista implementarán directamente los requerimientos del usuario y no habrá componentes competidores que coincidan con estos requerimientos. Sin embargo, en otros casos, el proceso de selección es mucho más complejo. No existirá un mapa claro de requerimientos en los componentes y habrá que integrar muchos componentes para cumplir un requerimiento específico o un conjunto de

La falla del lanzador Ariane 5

Mientras se desarrollaba el cohete espacial Ariane 5, los diseñadores decidieron reutilizar el software de referencia inercial que se desempeñó con éxito en el Ariane 4. El software de referencia inercial mantiene la estabilidad del cohete. Decidieron reutilizarlo sin cambios (como se haría con los componentes), aun cuando incluía funcionalidad adicional que no se requería en el Ariane 5.

En el primer lanzamiento del Ariane 5 falló el software de navegación inercial y no pudo controlarse el cohete. Los controladores en tierra instruyeron al lanzador a autodestruirse; así, el cohete y su carga fueron destruidos. La causa del problema fue una excepción no considerada cuando una conversión de un número de punto fijo a entero dio por resultado un desbordamiento numérico. Esto hizo que el sistema en tiempo de ejecución desactivara el sistema de referencia inercial, de manera que no pudo mantenerse la estabilidad del lanzador. La falla nunca ocurrió en el Ariane 4 porque tenía motores menos poderosos, y el valor que se convirtió no podía ser suficientemente grande para que la conversión se desbordara.

La falla ocurrió en un código que no se requería en el Ariane 5. Las pruebas de validación para el software de reutilización se basaron en los requerimientos del Ariane 5. Puesto que no había requerimientos para la función que falló, no se desarrollaron pruebas. En consecuencia, el problema con el software nunca se descubrió durante las pruebas de simulación del lanzamiento.

Figura 17.9 Ejemplo de falla de validación con software reutilizado

requerimientos. En consecuencia, se debe decidir cuáles composiciones ofrecen la mejor cobertura de los requerimientos.

Una vez seleccionados los componentes para su posible inclusión en un sistema, se deben validar para comprobar que se comportan como se espera. La extensión de la validación requerida depende de la fuente de los componentes. Si usted usa un componente que haya desarrollado una fuente conocida y confiable, puede decidir que no es necesaria la prueba de componentes. Usted simplemente analiza los componentes cuando se integran con otros. Por otra parte, si utiliza un componente de otras fuentes desconocidas, siempre deberá comprobar y probar dicho componente antes de incluirlo en su sistema.

La validación de componentes implica desarrollar un conjunto de casos de prueba para un componente (o, posiblemente, extender los casos de prueba suministrados con dicho componente) y desarrollar un conjunto de pruebas para ejecutar pruebas de componente. El principal problema con la validación de componentes es que la especificación de componentes tal vez no esté suficientemente detallada para permitirle desarrollar un conjunto completo de pruebas de componentes. Por lo general, los componentes se especifican de manera informal, y su única documentación formal es la de su especificación de interfaz. Ésta quizá no incluya suficiente información para desarrollar un conjunto completo de pruebas que lo convencerían de que la interfaz anunciada del componente es la que requiere.

Además de probar que un componente para reutilización logra lo que se requiere, es posible que se tenga que verificar también que el componente no incluya algún código o una funcionalidad maliciosos e innecesarios. Los desarrolladores profesionales pocas veces usan componentes de fuentes no confiables, en especial si esas fuentes no proporcionan código fuente. Por lo tanto, el problema de código malicioso no surge habitualmente. Sin embargo, los componentes con frecuencia pueden contener funcionalidad innecesaria y se debe comprobar que esta funcionalidad no interfiera con el uso del componente.

El problema con la funcionalidad innecesaria es que puede activarse por el componente en sí. Esto podría volver lento al componente, hacer que produzca resultados inesperados o, en algunos casos, provocar fallas graves al sistema. La figura 17.9 resume una situación en la que la funcionalidad innecesaria en un sistema de reutilización causó una falla catastrófica del software.

El problema en el lanzador Ariane 5 surgió porque las suposiciones acerca del software para el Ariane 4 eran inválidas para el Ariane 5. Éste es un problema general con los componentes de reutilización. Originalmente se implementan para un entorno de aplicación y, como es natural, se incrustan suposiciones acerca de ese entorno. Tales conjeturas se documentan pocas veces, así que, cuando se reutiliza el componente, es imposible efectuar pruebas para comprobar si las suposiciones todavía son válidas. Si usted reutiliza un componente en un entorno distinto, es posible que no descubra las suposiciones ambientales incrustadas sino hasta que use el componente en un sistema operativo.

17.3 Composición de componentes

La composición de componentes es el proceso de integrar componentes uno con otro y con "código pegamento" especialmente escrito para crear un sistema u otro componente. Existen muchas formas diferentes en las que se pueden componer componentes, como se muestra en la figura 17.10. De izquierda a derecha, dichos diagramas ilustran la composición secuencial, la composición jerárquica y la composición aditiva. En el siguiente análisis suponga que compone dos componentes (A y B) para crear uno nuevo:

1. La composición secuencial es la situación (a) en la figura 17.10. Usted crea un nuevo componente a partir de dos componentes existentes al llamar en secuencia a los componentes existentes. Puede considerar a la composición como una composición de las "interfaces proporciona". Esto es, se llaman los servicios ofrecidos por el componente A y luego los resultados emitidos por A se usan en la llamada a los servicios que ofrece el componente B. Los componentes no se llaman mutuamente en composición secuencial. Se requiere algún código pegamento adicional para llamar los servicios del componente en el orden correcto y asegurar que los resultados entregados por el componente A sean compatibles con las entradas esperadas por el componente B. La interfaz "proporciona" de la composición depende de la funcionalidad combinada de A y B, pero generalmente no será una composición de sus "interfaces proporciona". Este tipo de composición puede usarse con componentes que son elementos de programa o con componentes que son servicios.
2. La composición jerárquica es la situación (b) en la figura 17.10. Este tipo de composición ocurre cuando un componente llama directamente a los servicios que ofrece otro componente. El componente llamado proporciona los servicios que requiere el componente que llama. Por lo tanto, la interfaz "proporciona" del componente llamado debe ser compatible con la interfaz "requiere" del componente que llama. El componente A llama al componente B directamente y, si sus interfaces coinciden, tal vez no haya necesidad de un código adicional. Sin embargo, si existe una discordancia entre la interfaz "requiere" de A y la interfaz "proporciona" de B, entonces puede requerirse algún código de conversión. Como los servicios no tienen una interfaz "requiere", este modo de composición no se usa cuando los componentes se implementan como servicios Web.
3. La composición aditiva corresponde a la situación (c) en la figura 17.10. Esto ocurre cuando dos o más componentes se juntan (se suman) para crear un nuevo

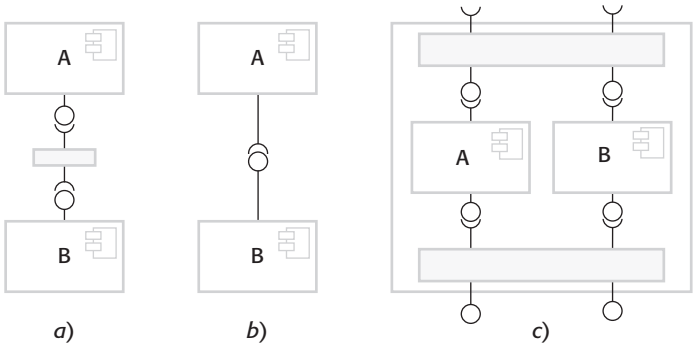


Figura 17.10 Tipos de composición de componente

componente, lo que combina su funcionalidad. La interfaz "proporciona" y la interfaz "requiere" del nuevo componente es una combinación de las correspondientes interfaces en los componentes A y B. Los componentes se llaman por separado mediante la interfaz externa del componente compuesto. A y B no son dependientes y no se llaman mutuamente. Este tipo de composición puede usarse con componentes que son unidades de programa o con componentes que son servicios.

Usted puede usar todas las formas de composición de componentes cuando crea un sistema. En todos los casos, tal vez tenga que escribir "código pegamento" que vincule los componentes. Por ejemplo, para composición secuencial, la salida del componente A se convierte por lo general en la entrada al componente B. Necesitará enunciados intermedios que llamen al componente A, recolecten el resultado y luego llamen al componente B con dicho resultado como parámetro. Cuando un componente llama a otro, tal vez necesite introducir un componente intermedio que asegure que la interfaz "proporciona" y la interfaz "requiere" sean compatibles.

Cuando escriba nuevos componentes especialmente para composición, deberá diseñar las interfaces de dichos componentes de manera que sean compatibles con otros componentes en el sistema. Por consiguiente, puede componer fácilmente dichos componentes en una sola unidad. No obstante, cuando los componentes se desarrollan de manera independiente para su reutilización, con frecuencia usted se enfrentará con incompatibilidades de interfaz. Esto significa que las interfaces de los componentes que desea componer no son iguales. Es posible que ocurran tres tipos de incompatibilidades:

1. *Incompatibilidad de parámetro* Las operaciones en cada lado de la interfaz tienen el mismo nombre, pero sus tipos de parámetro o el número de parámetros son diferentes.
2. *Incompatibilidad de operación* Los nombres de las operaciones en las interfaces "proporciona" y "requiere" son diferentes.
3. *Operación incompleta* La interfaz "proporciona" de un componente es un subconjunto de la interfaz "requiere" de otro componente o viceversa.

En todos los casos, el problema de la incompatibilidad se resuelve al escribir un adaptador que reconcilie las interfaces de los dos componentes a reutilizar. Un componente adaptador convierte una interfaz a otra. La forma precisa del adaptador depende del tipo

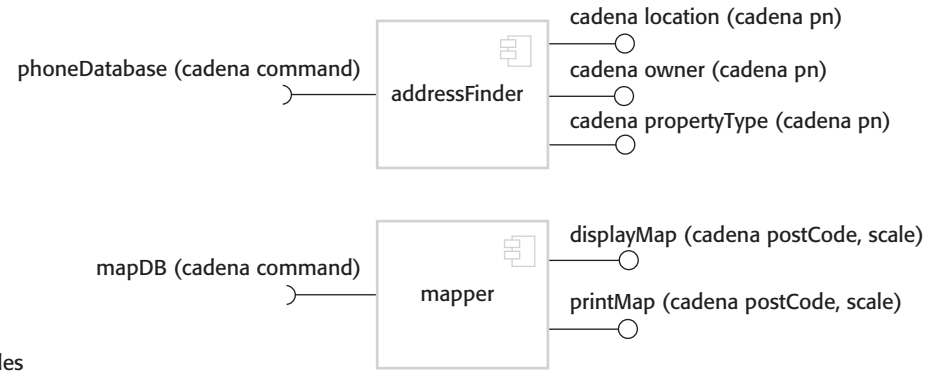


Figura 17.11
Componentes con interfaces incompatibles

de composición. En ocasiones, como en el siguiente ejemplo, el adaptador toma un resultado de un componente y lo convierte en una forma en que puede usarse como entrada a otro. En otros casos, el adaptador puede llamarse por el componente A como un proxy para el componente B. Esta situación ocurre si A quiere llamar a B, pero los detalles de la interfaz "requiere" de A no coinciden con los detalles de la interfaz "proporciona" de B. El adaptador reconcilia dichas diferencias al convertir sus parámetros de entrada desde A en los parámetros de entrada requeridos por B. Entonces llama a B para entregar los servicios requeridos por A.

Para ilustrar los adaptadores, considere los dos componentes que se muestran en la figura 17.11, cuyas interfaces son incompatibles. Esto puede ser parte de un sistema usado por los servicios de emergencia. Cuando el operador de emergencia recibe una llamada, el número telefónico se ingresa al componente addressFinder para localizar la dirección. Entonces, al usar el componente mapper, el operador imprime un mapa para mandarlo al vehículo que acude a la emergencia. De hecho, los componentes tendrían interfaces más complejas que las mostradas aquí, pero la versión simplificada ilustra el concepto de adaptador.

El primer componente, addressFinder, encuentra la dirección que coincide con un número telefónico. También puede regresar al dueño de la propiedad asociada con el número telefónico y el tipo de propiedad. El componente mapper toma un código postal (en Estados Unidos, un código ZIP estándar con los cuatro dígitos adicionales que identifican la ubicación de la propiedad) y muestra o imprime un mapa de las calles del área que rodean dicho código a una escala específica.

Tales componentes se pueden componer, en principio, porque la ubicación de la propiedad incluye un código postal o ZIP. Sin embargo, se debe escribir un componente adaptador llamado postCodeStripper que toma los datos de ubicación del addressFinder y consigue el código postal. Entonces este código postal se usa como entrada a mapper, y el mapa de las calles se muestra a una escala de 1:10,000. El siguiente código, que es un ejemplo de composición secuencial, ilustra la secuencia de llamadas que se requieren para implementar esto:

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

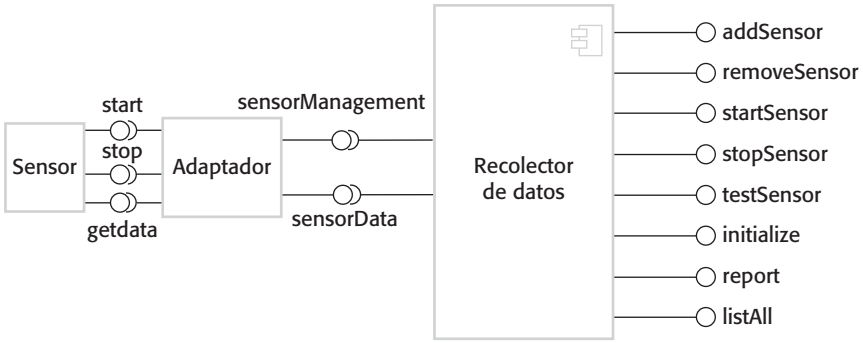


Figura 17.12
Adaptador que vincula un recolector de datos y un sensor

Otro caso en el que puede usarse un componente adaptador es el de la composición jerárquica, donde un componente quiere usar otro, pero existe una incompatibilidad entre la interfaz "proporciona" y la interfaz "requiere" de los componentes en la composición. En la figura 17.12 se ilustra el uso de un adaptador que vincula un recolector de datos y un componente sensor. Esto podría usarse en la implementación de un sistema de estación meteorológica a campo abierto, como se estudió en el capítulo 7.

Los componentes sensor y recolector de datos se componen usando un adaptador que reconcilia la interfaz "requiere" del componente de recolección de datos con la interfaz "proporciona" del componente sensor. El componente recolector de datos se diseñó con una interfaz "requiere" genérica que brinda soporte a la recolección de datos del sensor y la gestión del sensor. Para cada una de dichas operaciones, el parámetro es una cadena de texto que representa los comandos de sensor específicos. Por ejemplo, para emitir un comando de recolección (collect), diría sensorData("collect"). Como se muestra en la figura 17.12, el sensor en sí tiene operaciones separadas como start, stop y getdata.

El adaptador analiza gramaticalmente la cadena de entrada, identifica el comando (por ejemplo, collect) y luego llama a Sensor.getdata para recolectar el valor del sensor. Luego, regresa el resultado (como una cadena de caracteres) al componente recolector de datos. Este estilo de interfaz significa que el recolector de datos puede interactuar con diferentes tipos de sensor. Un adaptador separado, que convierte los comandos del sensor de Recolector de datos a la interfaz de sensor real, se implementa para cada tipo de sensor.

El análisis anterior de composición supone que usted puede decir, a partir de la documentación del componente, si las interfaces son compatibles o no. Desde luego, la definición de interfaz incluye el nombre de la operación y los tipos de parámetro, de manera que se puede hacer cierta valoración de la compatibilidad a partir de esto. Sin embargo, usted depende de la documentación del componente para decidir si las interfaces son semánticamente compatibles.

Para ilustrar este problema, considere la composición que se ilustra en la figura 17.13. Dichos componentes se usan para implementar un sistema que descargue imágenes de una cámara digital y las almacene en una fototeca. El usuario del sistema puede dar información adicional para describir y catalogar la fotografía. Para evitar confusión, aquí no se muestran todos los métodos de interfaz. En vez de ello, simplemente se

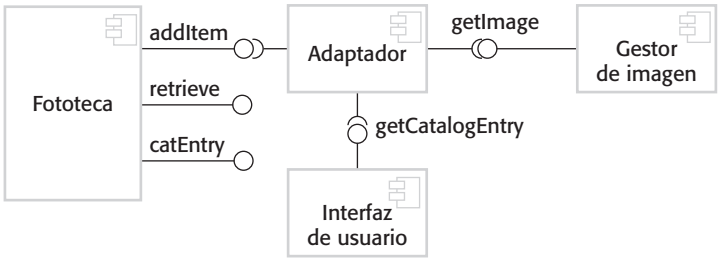


Figura 17.13 Composición de fototeca

indican los métodos que se necesitan para ejemplificar el problema de la documentación de componentes. Los métodos en la interfaz de Fototeca son:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry
photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid);
```

Suponga que la documentación para el método addItem en Fototeca es:

Este método agrega una fotografía a la fototeca y asocia el identificador de la fotografía, y cataloga el descriptor con la fotografía.

Esta descripción explica qué hace el componente; sin embargo, considere las siguientes preguntas:

- ¿Qué sucede si el identificador de fotografía ya está asociado con una fotografía en la fototeca?
- ¿El descriptor de fotografía está asociado con la entrada de catálogo, al igual que la fotografía? Esto es, si borra la fotografía, ¿también borra la información del catálogo?

No hay suficiente información en la descripción informal de addItem para responder dichas preguntas. Desde luego, es posible agregar más información a la descripción en lenguaje natural del método, pero, en general, la mejor forma de resolver ambigüedades es usar un lenguaje formal para describir la interfaz. La especificación que se muestra en la figura 17.14 es parte de la descripción de la interfaz de Fototeca que agrega información a la descripción informal.

La especificación en la figura 17.14 usa precondiciones y post-condiciones que se definen en una notación con base en el lenguaje de restricción de objeto (OCL), que es parte del UML (Warmer y Kleppe, 2003). OCL está diseñado para describir restricciones en modelos de objetos UML; permite expresar predicados que deben ser verdaderos siempre, que deben ser verdaderos antes de ejecutar un método, y qué deben ser verdaderos

```
— La palabra clave del contexto menciona el componente al que se aplican las
  condiciones

context addItem

— Las precondiciones especifican qué debe ser verdadero antes de ejecutar addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

— Las post-condiciones especifican qué es verdadero después de la ejecución
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:    PhotoLibrary.retrieve(pid) <>null ;

post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre

        PhotoLibrary.libSize() = libSize()@pre—1
```

Figura 17.14 Descripción OCL de la interfaz Fototeca

después de ejecutar un método. Se trata de invariantes, precondiciones y post-condiciones. Para acceder al valor de una variable antes de una operación, agregue @pre después de su nombre. Por lo tanto, al usar la edad como ejemplo:

```
age = age@pre + 1
```

Este enunciado significa que el valor de la edad después de una operación es uno más del que era antes de dicha operación.

Los enfoques basados en OCL se usan cada vez más para agregar información semántica a modelos UML, y las descripciones OCL pueden usarse para derivar generadores de código en ingeniería dirigida por modelo. El enfoque general se derivó del enfoque Diseño por Contrato, de Meyer (Meyer, 1992), en el que las interfaces y obligaciones de los objetos en comunicación se especifican formalmente y se refuerzan por el sistema al tiempo de ejecución. Meyer sugiere que usar Diseño por Contrato es esencial si usted debe desarrollar componentes confiables (Meyer, 2003).

La figura 17.14 incluye una especificación para los métodos addItem y delete en Fototeca. El método a especificar se indica mediante el contexto de palabra clave, y las precondiciones y post-condiciones mediante las palabras clave pre y post. Las precondiciones para addItem afirman que:

1. No debe haber una fotografía en la fototeca con el mismo identificador que la fotografía a ingresar.
2. Debe existir la fototeca: suponga que crear una fototeca agrega un solo ítem en ella, de manera que el tamaño de una fototeca siempre es mayor que cero.

3. Las post-condiciones para addItem afirman que:

   El tamaño de la fototeca aumentó por 1 (de manera que únicamente se realizó una sola entrada).

   Si usted recupera usando el mismo identificador, entonces recupera la fotografía que agregó.

   Si busca el catálogo usando dicho identificador, recupera la entrada catálogo que realizó.

La especificación delete brinda más información. La precondición afirma que, para borrar un ítem, éste debe estar en la fototeca y, después del borrado, la fotografía ya no podrá recuperarse y el tamaño de la fototeca se reducirá en 1. Sin embargo, delete no borra la entrada de catálogo: todavía es posible recuperarla después de borrar la fotografía. La razón de esto es que tal vez usted quiera mantener información en el catálogo acerca de por qué borró una fotografía, su nueva ubicación, etcétera.

Cuando usted crea un sistema al componer componentes, puede descubrir que hay conflictos potenciales entre requerimientos funcionales y no funcionales, o entre la necesidad de entregar un sistema tan rápidamente como sea posible y la necesidad de crear un sistema que puede evolucionar conforme cambian los requerimientos. Las decisiones donde puede tomar en cuenta negociaciones son:

1. ¿Qué composición es más efectiva para entregar los requerimientos funcionales del sistema?
2. ¿Qué composición facilitará la adaptación del componente cuando cambien los requerimientos?
3. ¿Cuáles serán las propiedades emergentes del sistema compuesto? Se trata de propiedades como rendimiento y confiabilidad. Sólo podrá valorarlos una vez que se implemente el sistema completo.

Por desgracia, existen muchas situaciones donde las soluciones a los problemas de composición podrían estar en conflicto. Por ejemplo, considere una situación como la que se ilustra en la figura 17.15, donde puede crear un sistema mediante dos composiciones alternativas. El sistema es una colección de datos y un sistema de reporte donde se recopilan datos de diferentes fuentes, se almacenan en una base de datos y luego se generan diferentes informes que resumen dichos datos.

Aquí, existe un conflicto potencial entre adaptabilidad y rendimiento. La composición *a*) es más adaptable, pero la composición *b*) tal vez es más rápida y más fiable. Las ventajas de la composición *a*) son que los reportes y la gestión de datos están separados, de manera que hay más flexibilidad para un cambio en el futuro. El sistema de gestión de datos podría sustituirse y, si se requieren reportes que el actual componente de reporte no puede generar, también es posible sustituir dicho componente sin tener que cambiar el componente de gestión de datos.

En la composición *b*) se usa un componente de base de datos con instalaciones de reporte incorporadas (por ejemplo, Microsoft Access). La ventaja clave de la composición *b*) es que existen menos componentes, de manera que será una implementación más

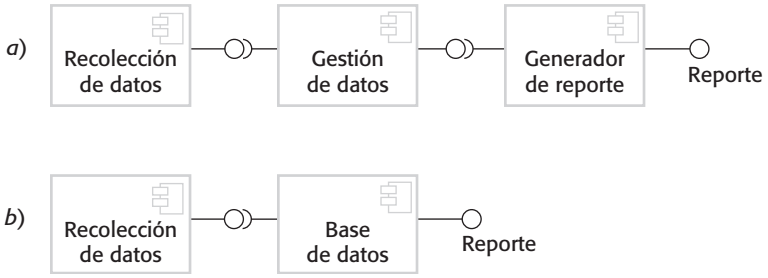


Figura 17.15 Componentes de recolección de datos y generación de reporte

rápida porque no hay cargas de comunicación del componente. Más aún, las reglas de integridad de datos que se aplican a la base de datos también se aplicarán a los reportes. Dichos reportes no podrán combinar datos en formas incorrectas. En la composición *a*), no hay tales restricciones, de manera que podrían ocurrir errores en los reportes.

En general, un buen principio de composición a seguir es el principio de separación de asuntos. Esto es, debe tratar de diseñar su sistema de tal forma que cada componente tenga un papel claramente definido y que, de manera ideal, dichos roles no se traslapen. Sin embargo, quizá sea menos costoso comprar un componente multifuncional en vez de dos o tres componentes separados. Más aún, podría haber sanciones en términos de la confiabilidad o el rendimiento cuando se usan múltiples componentes.

PUNTOS CLAVE

- La ingeniería de software basada en componentes es un enfoque fundado en la reutilización para definir, implementar y componer componentes independientes imprecisos en los sistemas.
- Un componente es una unidad de software cuya funcionalidad y dependencias están completamente definidas por un conjunto de interfaces públicas. Los componentes pueden componerse con algunos otros, sin conocimiento de su implementación y pueden aplicarse como una unidad ejecutable.
- Los componentes pueden implementarse como unidades de programa que se incluyen en un sistema o como servicios externos a los que se hace referencia dentro del sistema.
- Un modelo de componentes define un conjunto de estándares para componentes, incluidos estándares de interfaz, estándares de uso y estándares de implementación. La implementación del modelo de componentes brinda un conjunto de servicios comunes que pueden usarse por parte de todos los componentes.
- Durante el proceso CBSE, usted tiene que entremezclar los procesos de ingeniería de requerimientos y del diseño del sistema. Debe negociar requerimientos deseables contra los servicios que están disponibles a partir de componentes de reutilización existentes.
- La composición de componentes es el proceso de “organizar” componentes para crear un sistema. Los tipos de composición incluyen composición secuencial, composición jerárquica y composición aditiva.

- Cuando se componen componentes de reutilización que no se escribieron para la aplicación que usted desea, tal vez sea necesario escribir adaptadores o "código pegamento" para reconciliar las interfaces de los diferentes componentes.
- Cuando elija composiciones, deberá considerar la funcionalidad requerida del sistema, los requerimientos no funcionales y la facilidad con la que puede sustituirse un componente cuando cambie el sistema.

LECTURAS SUGERIDAS

Component-based Software Engineering: Putting the Pieces Together. Este libro es una colección de ensayos de varios autores acerca de diferentes aspectos de la CBSE. Como todas las colecciones, es una mezcla de temas, pero tiene mejor cobertura de los conflictos generales de la ingeniería de software con componentes que el libro de Szyperski. (G. T. Heineman y W. T. Councill, Addison-Wesley, 2001.)

Component Software: Beyond Object-Oriented Programming, 2nd ed. Esta edición actualizada del primer libro acerca de CBSE trata conflictos técnicos y no técnicos en CBSE. Contiene más detalles sobre tecnologías específicas que el libro de Heineman y Councill, e incluye un profundo análisis de los conflictos de mercado. (C. Szyperski, Addison-Wesley, 2002.)

"Specification, Implementation and Deployment of Components". Una buena introducción a los fundamentos de CBSE. El mismo conflicto del *CACM* incluye artículos acerca de componentes y el desarrollo basado en componentes. (I. Crnkovic, B. Hnich, T. Jonsson y Z. Kiziltan, *Comm. ACM,* **45** (10), octubre de 2002.) http://dx.doi.org/10.1145/570907.570928.

"Software Component Models". Éste es un análisis amplio de los modelos de componentes comerciales y de investigación que clasifican dichos modelos y explican las diferencias entre ellos. (K-K. Lau y Z. Wang, *IEEE Transactions on Software Engineering,* **33** (10), octubre de 2007.) http://dx.doi.org/10.1109/TSE.2007.70726.

EJERCICIOS

17.1. ¿Por qué es importante que todas las interacciones de los componentes se definan mediante interfaces "requiere" y "proporciona"?

17.2. El principio de independencia de componentes significa que debe ser posible sustituir un componente con otro que se implemente en una forma completamente diferente. Usando un ejemplo, explique cómo tal sustitución de componentes podría tener consecuencias indeseables y conducir a una falla del sistema.

17.3. ¿Cuáles son las diferencias fundamentales entre componentes como elementos de programa y componentes como servicios?

17.4. ¿Por qué es importante que los componentes se basen en un modelo de componentes estándar?

17.5. Con un ejemplo de un componente que implemente un tipo de datos abstracto, como una pila o una lista, demuestre por qué por lo general es necesario extender y adaptar componentes para su reutilización.

17.6. Explique por qué es difícil validar un componente de reutilización sin el código fuente del componente. ¿En qué formas la especificación formal de componente simplificaría los problemas de la validación?

17.7. Diseñe la interfaz "proporciona" y la interfaz "requiere" de un componente de reutilización que pueda usarse para representar a un paciente en el MHC-PMS.

17.8. Con ejemplos, ilustre los diferentes tipos de adaptador necesarios para soportar composición secuencial, composición jerárquica y composición aditiva.

17.9. Diseñe las interfaces de los componentes que puedan usarse en un sistema para una sala de control de emergencias. Debe diseñar interfaces para un componente de registro de llamadas que registre las llamadas realizadas, y un componente de descubrimiento de vehículo que, a partir de un código postal (*zip code*) y un tipo de incidente, encuentre el vehículo adecuado más cercano para enviarlo al lugar del incidente.

17.10. Se ha sugerido que debe establecerse una autoridad de certificación independiente. Los proveedores enviarían sus componentes a esta autoridad, la cual validaría que el componente es confiable. ¿Cuáles serían las ventajas y desventajas de tal autoridad de certificación?

REFERENCIAS

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. y Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ.: Prentice Hall.

Councill, W. T. y Heineman, G. T. (2001). "Definition of a Software Component and its Elements". En *Component-based Software Engineering*. Heineman, G. T. y Councill, W. T. (ed.). Boston: Addison-Wesley, 5–20.

Jacobson, I., Griss, M. y Jonsson, P. (1997). *Software Reuse*. Reading, Mass.: Addison-Wesley.

Kotonya, G. (2003). "The CBSE Process: Issues and Future Visions". *Proc. 2nd CBSEnet workshop,* Budapest, Hungría.

Lau, K.-K. y Wang, Z. (2007). "Software Component Models". *IEEE Trans. on Software Eng.*, **33** (10), 709–24.

Meyer, B. (1992). "Design by Contract". *IEEE Computer,* **25** (10), 40–51.

Meyer, B. (2003). "The Grand Challenge of Trusted Components". *ICSE 25: Int. Conf. on Software Eng.,* Portland, Oregon: IEEE Press.

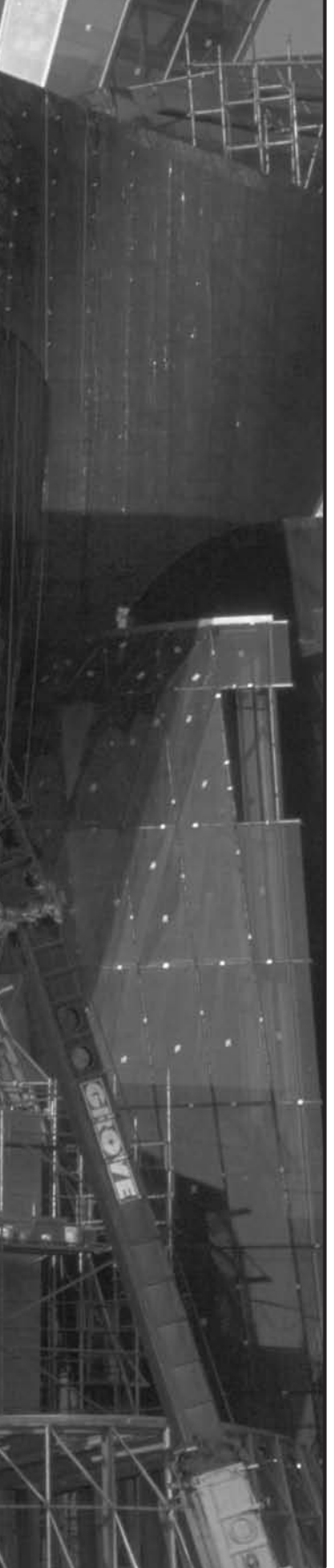
Mili, H., Mili, A., Yacoub, S. y Addy, E. (2002). *Reuse-based Software Engineering*. Nueva York: John Wiley & Sons.

Pope, A. (1997). *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Szyperski, C. (2002). *Component Software: Beyond Object-oriented Programming, 2nd ed*. Harlow, UK: Addison-Wesley.

Warmer, J. y Kleppe, A. (2003). *The Object Constraint Language: Getting your models ready for MDA.* Boston: Addison-Wesley.

Weinreich, R. y Sametinger, J. (2001). "Component Models and Component Services: Concepts and Principles". En *Component-Based Software Engineering*. Heineman, G. T. y Councill, W. T. (ed.). Boston: Addison-Wesley, 33–48.



18

Ingeniería de software distribuido

Objetivos

El objetivo de este capítulo es introducirlo a la ingeniería de sistemas distribuidos y las arquitecturas de sistemas distribuidos. Al estudiar este capítulo:

- conocerá los conflictos clave que debe considerar al diseñar e implementar los sistemas de software distribuidos;
- comprenderá el modelo de cómputo cliente-servidor y la arquitectura en capas de los sistemas cliente-servidor;
- se introducirá a los patrones de uso común para las arquitecturas de sistemas distribuidos y reconocerá los tipos de sistema más aplicables para cada arquitectura;
- entenderá la noción de software como servicio y proporcionará acceso basado en Web a sistemas de aplicación de implementación remota.

Contenido

18.1 Conflictos de los sistemas distribuidos
18.2 Computación cliente-servidor
18.3 Patrones arquitectónicos para sistemas distribuidos
18.4 Software como servicio

Prácticamente todos los grandes sistemas basados en computadora son ahora sistemas distribuidos. Un sistema distribuido es aquel que implica numerosas computadoras, en contraste con los sistemas centralizados en que todos los componentes de sistema se ejecutan en una sola computadora. Tanenbaum y Van Steen (2007) definen un sistema distribuido como:

> *. . .una colección de computadoras independientes que aparecen al usuario como un solo sistema coherente.*

Es evidente que la ingeniería de los sistemas distribuidos tiene mucho en común con la ingeniería de cualquier otro software. Sin embargo, existen conflictos específicos que deben considerarse al diseñar este tipo de sistema. Éstos surgen porque los componentes del sistema pueden estar en ejecución en computadoras con administración independiente y éstas se comunican a través de una red.

Coulouris y sus colaboradores (2005) identifican las siguientes ventajas de usar un enfoque distribuido para el desarrollo de sistemas:

1. *Compartición de recursos* Un sistema distribuido permite compartir los recursos de hardware y software, tales como discos, impresoras, archivos y compiladores, que se asocian con computadoras en una red.
2. *Apertura* Los sistemas distribuidos, por lo general, son sistemas abiertos, lo cual significa que están diseñados en torno a protocolos estándar que permiten la combinación de equipo y software de diferentes proveedores.
3. *Concurrencia* En un sistema distribuido, grandes procesos pueden ejecutarse al mismo tiempo en computadoras independientes en red. Dichos procesos pueden (pero no es necesario) comunicarse uno con el otro durante su operación normal.
4. *Escalabilidad* Al menos en principio, los sistemas distribuidos son escalables cuando las capacidades del sistema pueden aumentarse al agregar nuevos recursos para enfrentar nuevas demandas del sistema. En la práctica, la red que vincula las computadoras individuales en el sistema puede limitar la escalabilidad del sistema.
5. *Tolerancia a fallas* La disponibilidad de muchas computadoras y el potencial de reproducir información significa que los sistemas distribuidos pueden tolerar algunas fallas de hardware y software (véase el capítulo 13). En la mayoría de los sistemas distribuidos, puede darse un servicio degradado al ocurrir fallas; la pérdida completa de servicio sólo sucede cuando hay una falla de red.

Para sistemas organizacionales a gran escala, estas ventajas representan que los sistemas distribuidos han sustituido, en especial, a los sistemas heredados mainframe desarrollados en la década de 1990. Sin embargo, existen numerosos sistemas de aplicación para computadora personal (por ejemplo, sistemas de edición de fotografía), que no son distribuidos y que ejecutan en un solo sistema de cómputo. La mayoría de los sistemas embebidos también son sistemas de procesador individual.

Los sistemas distribuidos son inherentemente más complejos que los sistemas centralizados. Esto los hace más difíciles de diseñar, implementar y poner a prueba. Es más complicado entender las propiedades emergentes de los sistemas distribuidos debido a la complejidad de las interacciones entre los componentes y la infraestructura del sistema.

Por ejemplo, en vez de que el rendimiento del sistema dependa de la velocidad de ejecución de un procesador, depende del ancho de banda de la red, la carga de la red y la velocidad de todas las computadoras que forman parte del sistema. Trasladar los recursos de una parte del sistema a otra puede afectar significativamente el rendimiento del sistema.

Más aún, como saben todos los usuarios de la WWW, los sistemas distribuidos son impredecibles en su respuesta. El tiempo de respuesta depende de la carga total del sistema, su arquitectura y la carga de la red. Puesto que todo esto puede cambiar a corto plazo, el tiempo para responder a la petición de un usuario puede variar drásticamente de una petición a otra.

El desarrollo más importante que ha afectado a los sistemas de software distribuidos en los años anteriores es el enfoque orientado a servicios. Gran parte de este capítulo se enfoca en los conflictos generales de los sistemas distribuidos, pero en la sección 18.4 se trata la noción de aplicaciones implementadas como servicios. Esto complementa el material del capítulo 19, que se enfoca en los servicios como componentes en una arquitectura orientada a servicios, y en los conflictos más generales de la ingeniería de software orientada a servicios.

18.1 Conflictos de los sistemas distribuidos

Como se explicó en la introducción de este capítulo, los sistemas distribuidos son más complejos que los sistemas que se ejecutan en un solo procesador. Esta complejidad surge porque es prácticamente imposible tener un modelo descendente de control de dichos sistemas. Los nodos en el sistema que entregan funcionalidad con frecuencia son sistemas independientes sin una sola autoridad encargada de ellos. La red que conecta dichos nodos es un sistema de gestión independiente. Es un sistema complejo por derecho propio y no puede controlarse por los propietarios de los sistemas que usan la red. Por lo tanto, existe una imprevisibilidad inherente en la operación de los sistemas distribuidos que debe considerar el diseñador del sistema.

Algunos de los conflictos de diseño más importantes que deben considerarse en la ingeniería de sistemas distribuidos son:

1. *Transparencia* ¿En qué medida el sistema distribuido debe aparecer al usuario como un solo sistema? ¿Cuándo es útil para los usuarios entender que el sistema es distribuido?
2. *Apertura* ¿Un sistema debe diseñarse usando protocolos estándar que soporten interoperabilidad o deben usarse protocolos más especializados que restrinjan la libertad del diseñador?
3. *Escalabilidad* ¿Cómo puede construirse el sistema para que sea escalable? Esto es, ¿cómo podría diseñarse un sistema global para que su capacidad se pueda aumentar en respuesta a demandas crecientes hechas sobre el sistema?
4. *Seguridad* ¿Cómo pueden definirse e implementarse políticas de seguridad útiles que se apliquen a través de un conjunto de sistemas administrados de manera independiente?

CORBA (Common Object Request Broker Architecture), Arquitectura Común de Intermediario de Peticiones de Objetos

CORBA es una especificación bien conocida para un sistema middleware que el Object Management Group desarrolló en la década de 1990. Estaba previsto como un estándar abierto que permitiría el desarrollo de middleware para soporte de comunicaciones y ejecución de componentes distribuidos, además de proporcionar un conjunto de servicios estándar que pudieran usar dichos componentes.

Se produjeron varias implementaciones de CORBA, pero el sistema nunca logró masa crítica. Los usuarios prefirieron sistemas propietarios o se trasladaron a arquitecturas orientadas a servicios.

http://www.SoftwareEngineering-9.com/Web/DistribSys/Corba.html

5. *Calidad de servicio* ¿Cómo debe especificarse la calidad del servicio que se entrega a los usuarios del sistema y cómo debe implementarse el sistema para entregar una calidad de servicio aceptable para todos los usuarios?
6. *Gestión de fallas* ¿Cómo pueden detectarse las fallas del sistema, contenerse (de modo que tengan efectos mínimos sobre otros componentes del sistema) y repararse?

En un mundo ideal, el hecho de que un sistema sea distribuido sería transparente para los usuarios. Esto significa que los usuarios verían el sistema como un solo sistema cuyo comportamiento no resulta afectado por la forma en que se distribuye el sistema. En la práctica, esto es imposible de lograr. Es improbable el control central de un sistema distribuido y, como resultado, las computadoras individuales en un sistema pueden comportarse de manera diferente en distintos momentos. Más aún, las demoras de red son ineludibles, puesto que el hecho de que las señales viajen a través de la red siempre tiene una duración finita de tiempo. La duración de dichas demoras depende de la ubicación de los recursos en el sistema, la calidad de conexión de la red del usuario y la carga de la red.

El enfoque de diseño para lograr transparencia depende de la creación de abstracciones de los recursos en un sistema distribuido, de modo que la realización física de dichos recursos puede cambiar sin tener que realizar variaciones en el sistema de aplicación. El middleware (estudiado en la sección 18.1.2) se usa para trazar un mapa de los recursos lógicos referidos por un programa sobre los recursos físicos reales, y gestionar las interacciones entre dichos recursos.

En la práctica, es imposible hacer un sistema por completo transparente y los usuarios, por lo general, están al tanto de que se enfrentan con un sistema distribuido. Por consiguiente, usted puede decidir que es mejor exponer la distribución para los usuarios. Así, ellos podrán prepararse para algunas de las consecuencias de la distribución, como las demoras de red, fallas del nodo remoto, etcétera.

Los sistemas distribuidos abiertos son sistemas que se construyen de acuerdo con estándares generalmente aceptados. Esto significa que los componentes de cualquier proveedor pueden integrarse en el sistema e interoperar con los otros componentes del sistema. A nivel de red, los sistemas abiertos se conforman de acuerdo con los protocolos de Internet, pero a nivel de componente la apertura no es aún universal. La apertura implica que los componentes del sistema pueden desarrollarse de manera independiente en cualquier lenguaje de programación y, si se ajustan a las normas, funcionarán con otros componentes.

El estándar CORBA (Pope, 1997) desarrollado en la década de 1990 tenía la intención de conseguir esto, aunque nunca logró una cantidad suficiente que lo adoptaran. En vez de ello, muchas compañías eligieron desarrollar sistemas utilizando estándares propietarios para componentes de compañías como Sun y Microsoft. Ello proporcionó mejores implementaciones y soporte de software, y a largo plazo mejor soporte para protocolos industriales.

Los estándares de servicio Web (que se estudian en el capítulo 19) para arquitecturas orientadas a servicios se desarrollaron para ser estándares abiertos. Sin embargo, existe significativa resistencia a dichos estándares debido a su percibida ineficiencia. Algunos desarrolladores de sistemas basados en servicio optaron por los llamados protocolos RESTful, ya que tienen una sobrecarga inferior a los protocolos de servicio Web.

La escalabilidad de un sistema refleja su disponibilidad para entregar una alta calidad de servicio conforme aumentan las demandas al sistema. Neuman (1994) identifica tres dimensiones de la escalabilidad:

1. *Tamaño* Debe ser posible agregar más recursos a un sistema para enfrentar el creciente número de usuarios.
2. *Distribución* Debe ser posible dispersar geográficamente los componentes de un sistema sin reducir su rendimiento.
3. *Manejabilidad* Debe ser posible administrar un sistema conforme aumenta en tamaño, incluso si las partes del sistema se ubican en organizaciones independientes.

En términos de tamaño, hay una distinción entre expandir (*scaling up*) y ampliar (*scaling out*). Lo primero significa sustituir recursos en el sistema con recursos más poderosos. Por ejemplo, es posible expandir o aumentar la memoria de un servidor de 16 GB a 64 GB. En cambio, ampliar significa agregar recursos adicionales al sistema (por ejemplo, un servidor Web adicional para trabajar junto a un servidor existente). Ampliar es a menudo más efectivo en costo que expandir, aun cuando, por lo general, representa que el sistema debe diseñarse para que sea posible el procesamiento concurrente.

En la parte 2 de este libro se analizaron los conflictos generales de la seguridad y los conflictos de la ingeniería de seguridad. Sin embargo, cuando se distribuye un sistema, el número de formas en que éste puede ser atacado aumenta considerablemente, en comparación con los sistemas centralizados. Si una parte del sistema es atacada con éxito, entonces el atacante podrá usar esto como una "puerta trasera" en otras partes del sistema.

Los tipos de ataques contra los que debe defenderse un sistema distribuido son los siguientes:

1. Intercepción, en que un atacante intercepta las comunicaciones entre las partes del sistema, para que haya poca confidencialidad.
2. Interrupción, que sucede cuando los servicios del sistema son atacados y no pueden entregarse como se esperaba. Los ataques de negación de servicio implican el bombardeo de un nodo con peticiones de servicio ilegítimas, de modo que no se puede hacer frente a peticiones legítimas.
3. Modificación, que se presenta cuando el atacante cambia los datos o servicios del sistema.
4. Fabricación, que sucede cuando un atacante genera información que no debe existir y luego la usa para conseguir ciertos privilegios. Por ejemplo, un atacante puede generar una falsa entrada de contraseña y usarla para obtener el acceso a un sistema.

La principal dificultad en los sistemas distribuidos es el establecimiento de una política de seguridad que pueda aplicarse de manera fiable a todos los componentes del sistema. Como se refirió en el capítulo 11, un conjunto de políticas de seguridad establece el nivel de seguridad que debe alcanzar un sistema. Los mecanismos de seguridad, tales como encriptación y autenticación, se usan para reforzar la política de seguridad. Las dificultades en un sistema distribuido surgen debido a que diferentes organizaciones pueden poseer partes del sistema. Tales organizaciones podrían tener políticas y mecanismos de seguridad incompatibles entre sí. Tal vez sea necesario hacer compromisos de seguridad para permitir al sistema trabajar en conjunto.

La calidad del servicio (QoS, por las siglas de *Quality of Service*) ofrecida por un sistema distribuido refleja la capacidad del sistema para entregar sus servicios de manera confiable y con un tiempo de respuesta y rendimiento total que sean aceptables para sus usuarios. De manera ideal, los requerimientos QoS deben especificarse por adelantado y el sistema debe diseñarse y configurarse para entregar dicha QoS. Por desgracia, esto no siempre es posible, por dos razones:

1. Tal vez no sea efectivo en términos de costos diseñar y configurar el sistema para que entregue una QoS bajo carga pico. Esto podría implicar poner a disposición recursos que no se usan durante gran parte del tiempo. Uno de los principales argumentos para la "computación en nube" es que enfrenta parcialmente este problema. Al usar una nube, es fácil agregar recursos conforme aumente la demanda.
2. Los parámetros QoS pueden ser contradictorios entre sí. Por ejemplo, creciente fiabilidad puede significar reducción en el rendimiento total, puesto que se introducen procedimientos de comprobación para garantizar que son válidas todas las entradas del sistema.

La QoS es particularmente importante cuando el sistema enfrenta datos críticos en el tiempo, tales como difusión de sonido o video. Ante estas circunstancias, si la QoS cae por abajo de un valor umbral, entonces el sonido o el video puede quedar tan degradado que es imposible de entender. Los sistemas que se enfrentan con el sonido y el video deben incluir negociación y componentes de gestión QoS. Éstos deberían evaluar los requerimientos QoS frente a los recursos disponibles y, si son insuficientes, negociar más recursos o una reducción de la QoS establecida como meta.

En un sistema distribuido, es inevitable que ocurran fallas, así que el sistema debe diseñarse para ser resistente a dichas fallas. Las fallas son tan omnipresentes, que una definición poco seria de un sistema distribuido sugerida por Leslie Lamport, destacado investigador de los sistemas distribuidos, es la siguiente:

> *Usted sabe que tiene un sistema distribuido cuando la caída de un sistema de la que nunca escuchó impide que usted realice cualquier trabajo.*

La gestión de fallas en la operación implica la aplicación de técnicas de tolerancia a fallas que se estudiaron en el capítulo 13. En consecuencia, los sistemas distribuidos deben incluir mecanismos para descubrir si falló un componente del sistema, deben seguir entregando tantos servicios como sea posible a pesar de dicha falla y, en la medida de lo posible, recuperarse automáticamente de la falla.

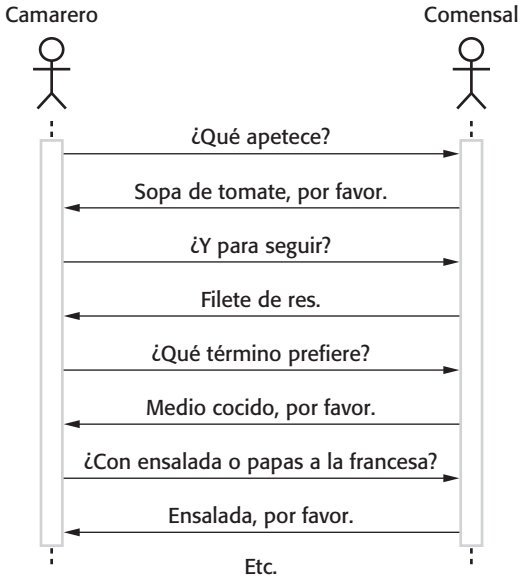


Figura 18.1 Interacción procedimental entre un comensal y un camarero

18.1.1 Modelos de interacción

Existen dos tipos fundamentales de interacción que pueden tener lugar entre las computadoras en un sistema de cómputo distribuido: interacción procedimental e interacción basada en mensajes. La primera implica una computadora que solicita un servicio conocido ofrecido por alguna otra computadora y (por lo general) espera la entrega de dicho servicio. La interacción basada en mensajes implica que la computadora "emisora" defina en un mensaje la información acerca de lo que requiere, el cual se envía entonces a otra computadora. Por lo general, los mensajes transmiten más información en una sola interacción que la solicitud de un procedimiento a otra máquina.

Para ilustrar la diferencia entre interacción procedimental y basada en mensajes, considere una situación donde usted solicita los alimentos que desea en un restaurante. Cuando tiene una conversación con el camarero, participa en una serie de interacciones sincrónicas procedimentales que definen su pedido. Usted hace una solicitud; el camarero reconoce dicha solicitud; usted hace otra solicitud, que es reconocida; y así sucesivamente. Esto es comparable a los componentes que interactúan en un sistema de software donde un componente solicita métodos de otros componentes. El camarero toma nota de su pedido junto con las órdenes de los demás comensales. Entonces comunica el pedido, que incluye detalles de todo lo que solicitó, al personal de cocina para que prepare la comida. En esencia, el camarero transmite un mensaje al personal de la cocina que define la comida a preparar. Ésta es una interacción basada en mensajes.

Esto se ilustra en la figura 18.1, la cual muestra el proceso de solicitud sincrónico como una serie de solicitudes, y en la figura 18.2, donde se refiere un hipotético mensaje XML que define una orden expedida por la mesa de tres personas. Es clara la diferencia entre estas formas de intercambio de información. El camarero toma la orden como una serie de interacciones, en que cada interacción define parte de la orden. Sin embargo,

```
<starter>
   <dish name = “soup” type = “tomato” />
   <dish name = “soup” type = “fish” />
   <dish name = “pigeon salad” />
</starter>
<main course>
   <dish name = “steak” type = “sirloin” cooking = “medium” />
   <dish name = “steak” type = “fillet” cooking = “rare” />
   <dish name = “sea bass”>
</main>
<accompaniment>
   <dish name = “french fries” portions = “2” />
   <dish name = “salad” portions = “1” />
</accompaniment>
```

Figura 18.2 Interacción basada en mensajes entre un camarero y el personal de cocina

el camarero tiene una sola interacción con la cocina, donde el mensaje define la orden completa.

La comunicación procedimental en un sistema distribuido se implementa con frecuencia usando solicitudes de procedimiento remoto (RPC, por las siglas de *Remote Procedure Calls*). En RPC, un componente solicita otro componente como si fuera un procedimiento o método local. El middleware en el sistema intercepta esta solicitud y la transmite a un componente remoto. Éste realiza la computación requerida y, por medio del middleware, regresa el resultado al componente solicitante. En Java, las solicitudes de método remoto (RMI, por las siglas de *Remote Method Invocations*) son comparables con RPC, aunque no idénticas. El framework RMI maneja la petición de métodos remotos en un programa Java.

RPC requiere un “stub” (talonario) para que el procedimiento solicitado sea accesible en la computadora que inicia la solicitud. Se solicita el stub y éste traduce los parámetros del procedimiento en una representación estándar para transmisión al procedimiento remoto. A través del middleware, envía entonces la petición de ejecución al procedimiento remoto. Éste usa funciones de librería para convertir los parámetros en el formato requerido, realiza los cálculos y luego comunica los resultados requeridos vía el “stub” que representa al que solicita.

La interacción basada en mensajes implica normalmente un componente que crea un mensaje, el cual detalla los servicios requeridos por otro componente. Mediante el sistema middleware, éste se envía al componente de recepción. El receptor analiza el mensaje, efectúa los cálculos y diseña un mensaje para el componente emisor con los resultados requeridos. Entonces esto se transmite al middleware para que, a la vez, lo transmita al componente emisor.

Un problema con el enfoque RPC de interacción es que tanto el solicitante como el solicitado tienen que estar disponibles al momento de la comunicación, y deben saber cómo se refieren el uno al otro. En esencia, una RPC tiene los mismos requerimientos que una solicitud de procedimiento o método local. En contraste, en un enfoque basado en mensajes, puede tolerarse la indisponibilidad, pues el mensaje simplemente permanece en una cola hasta que el receptor esté disponible. Más aún, no es necesario que emisario y receptor estén al tanto uno del otro. Ellos simplemente se comunican con el middleware, que es el responsable de garantizar que los mensajes se transmitan al sistema adecuado.

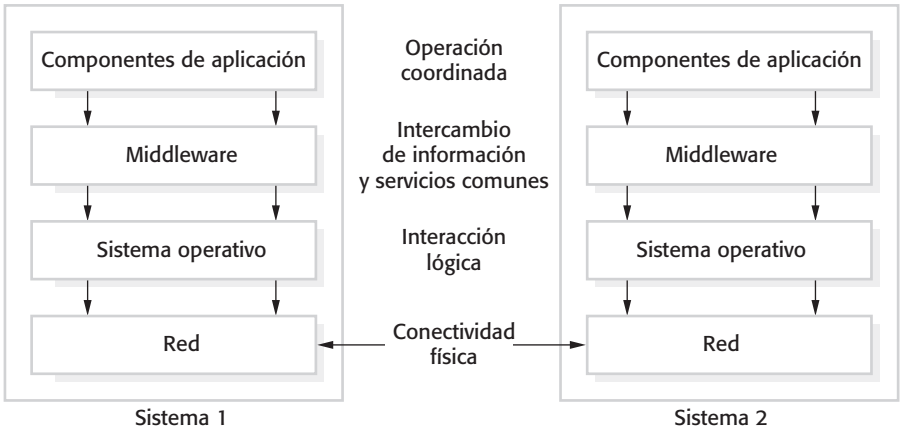


Figura 18.3
Middleware en un sistema distribuido

18.1.2 Middleware

Los componentes en un sistema distribuido pueden implementarse en distintos lenguajes de programación y ejecutarse en tipos de procesador diferentes por completo. Modelos de datos, representación de información y protocolos para comunicación pueden ser todos distintos. Por lo tanto, un sistema distribuido requiere software que pueda gestionar esas diversas partes, y garantizar que puedan comunicarse e intercambiar datos.

El término "middleware" se usa para referirse a este software: se encuentra en el centro, entre los componentes distribuidos del sistema. Esto se ilustra en la figura 18.3, la cual muestra que el middleware es una capa entre el sistema operativo y los programas de aplicación. Por lo general, el middleware se implementa como un conjunto de librerías, que se instalan en cada computadora distribuida, además de un sistema de tiempo en operación para gestionar las comunicaciones.

Bernstein (1996) describe tipos de middleware que están disponibles para soportar computación distribuida. El middleware es software de propósito general que se compra comúnmente en tiendas en lugar de que desarrolladores de aplicación los escriban especialmente. Los ejemplos de middleware incluyen software para gestionar comunicaciones con bases de datos, gestores de transacciones, convertidores de datos y controladores de comunicación.

En un sistema distribuido, el middleware por lo general brinda dos distintos tipos de soporte:

1. Soporte de interacción, en el que el middleware coordina las interacciones entre diferentes componentes del sistema. El middleware proporciona transparencia de ubicación en cuanto a que no es necesario que los componentes conozcan las ubicaciones físicas de los otros componentes. También puede soportar conversión de parámetros si se usan diferentes lenguajes de programación para la implementación de componentes, detección de eventos y comunicación, etcétera.
2. La provisión de servicios comunes, en la que el middleware proporciona implementaciones de reutilización de servicios que pueden requerir varios componentes en el sistema distribuido. Al usar dichos servicios comunes, los componentes pueden interoperar fácilmente y ofrecer a los usuarios servicios en una forma consistente.

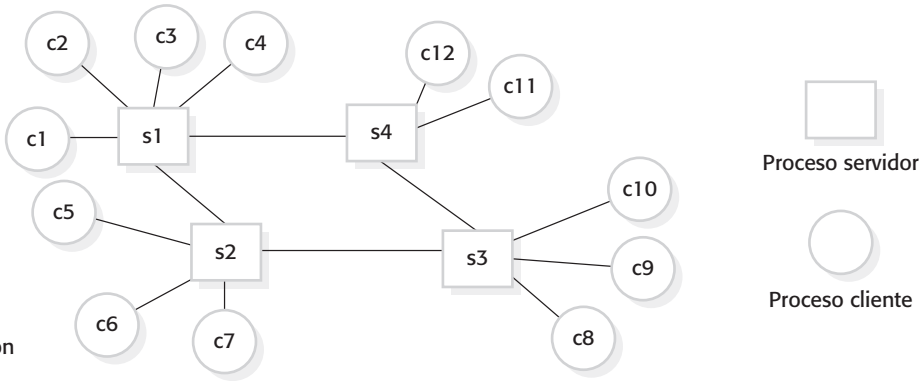


Figura 18.4 Interacción cliente-servidor

En la sección 18.1.1 se presentaron ejemplos del soporte a la interacción que puede ofrecer el middleware. Este último se usa para soportar procedimientos remotos y solicitudes de métodos remotos, intercambio de mensajes, etcétera.

Los servicios comunes son aquellos que pueden requerir diferentes componentes con independencia de la funcionalidad de dichos componentes. Como se describió en el capítulo 17, pueden incluir servicios de seguridad (autenticación y autorización), servicios de notificación y nomenclatura, y servicios de gestión de transacción, etcétera. Usted puede considerar que estos servicios comunes los brinda un contenedor middleware. Entonces usted implementa su componente en dicho contenedor y puede acceder y usar dichos servicios comunes.

18.2 Computación cliente-servidor

Los sistemas distribuidos a los que se accede por Internet se organizan normalmente como sistemas cliente-servidor. En un sistema cliente-servidor, el usuario interactúa con un programa que se ejecuta en su computadora local (por ejemplo, un navegador Web o una aplicación basada en telefonía). Éste interactúa con otro programa que se ejecuta en una computadora remota (por ejemplo, un servidor Web). La computadora remota proporciona servicios, como acceso a páginas Web, que están disponibles a clientes externos. Este modelo cliente-servidor, como se estudió en el capítulo 6, es un modelo arquitectónico muy general de una aplicación. No está restringido a aplicaciones distribuidas a través de varias máquinas. También se puede usar como un modelo de interacción lógica donde cliente y servidor operan en la misma computadora.

En una arquitectura cliente-servidor, una aplicación se modela como un conjunto de servicios que proporcionan los servidores. Los clientes pueden acceder a dichos servicios y presentar resultados a usuarios finales (Orfali y Harkey, 1998). Los clientes deben estar al tanto de los servidores que están disponibles, pero no conocen la existencia de otros clientes. Clientes y servidores son procesos separados, como se muestra en la figura 18.4. Ésta ilustra una situación donde existen cuatro servidores (s1-s4), que entregan diferentes servicios. Cada servicio tiene un conjunto de clientes asociados que acceden a dichos servicios.

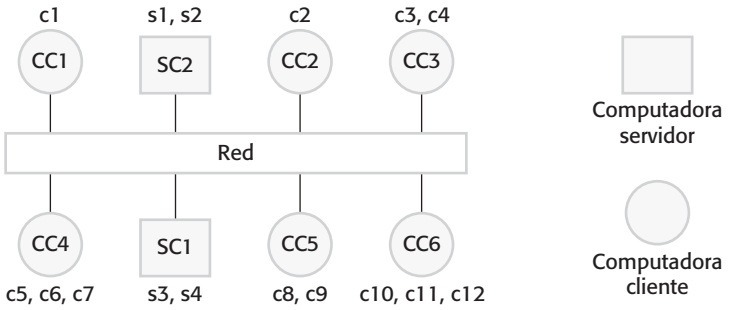


Figura 18.5 Mapeo de clientes y servidores para computadoras en red

La figura 18.4 muestra procesos cliente y servidor en vez de procesadores. Es normal que muchos procesos cliente se ejecuten en un solo procesador. Por ejemplo, en su PC, usted puede ejecutar un cliente de correo que descargue mensajes de un servidor de correo remoto. También puede ejecutar un navegador Web que interactúe con un servidor Web remoto y un cliente de impresión que envíe documentos a una impresora remota. La figura 18.5 ilustra la situación donde los 12 clientes lógicos que se muestran en la figura 18.4 operan en seis computadoras. Los cuatro procesos servidor se mapean sobre dos computadoras servidor físicas.

Varios procesos servidor diferentes pueden ejecutar en el mismo procesador, pero, con frecuencia, los servidores se implementan como sistemas multiprocesador en los que una instancia independiente del proceso servidor se ejecuta en cada máquina. El software de balanceo de carga distribuye las peticiones de servicio de los clientes a diferentes servidores, de modo que cada servidor realiza la misma cantidad de trabajo. Esto permite el manejo de mayor volumen de transacciones con los clientes, sin degradar la respuesta a clientes individuales.

Los sistemas cliente-servidor dependen de que exista una separación clara entre la presentación de información y los cálculos que crea y procesa esa información. En consecuencia, se debe diseñar la arquitectura de los sistemas distribuidos cliente-servidor para que se estructuren en varias capas lógicas, con interfaces claras entre dichas capas. Esto permite que cada capa se distribuya en diferentes computadoras. La figura 18.6 ilustra este modelo y muestra una aplicación estructurada en cuatro capas:

- Una capa de presentación que se ocupa de presentar la información al usuario y gestionar todas las interacciones de usuario;
- Una capa de gestión de datos que gestiona los datos que pasan hacia y desde el cliente. Esta capa puede implementar comprobaciones en los datos, generar páginas Web, etcétera;
- Una capa de procesamiento de aplicación que se ocupa de implementar la lógica de la aplicación y, de este modo, proporciona la funcionalidad requerida a los usuarios finales;
- Una capa de base de datos que almacena los datos y ofrece servicios de gestión de transacción, etcétera.

La siguiente sección explica cómo diferentes arquitecturas cliente-servidor distribuyen dichas capas lógicas en diversas formas. El modelo cliente-servidor también destaca la noción de software como servicio (SaaS, por las siglas de *Software as a Service*), una manera cada vez más importante de implementar software y acceder a él a través de Internet. Esto se estudia en la sección 18.4.

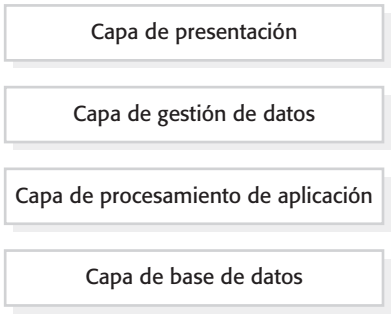


Figura 18.6 Modelo arquitectónico en capas para una aplicación cliente-servidor

18.3 Patrones arquitectónicos para sistemas distribuidos

Como se explicó en la introducción de este capítulo, los diseñadores de sistemas distribuidos deben organizar sus diseños de sistema para encontrar un equilibrio entre rendimiento, confiabilidad, seguridad y manejabilidad del sistema. No hay un modelo universal de organización de sistemas adecuado a todas las circunstancias, así que han surgido varios estilos arquitectónicos distribuidos. Cuando diseñe una aplicación distribuida, deberá elegir un estilo arquitectónico que soporte los requerimientos no funcionales críticos de su sistema.

En esta sección se describen cinco estilos arquitectónicos:

1. Arquitectura maestro-esclavo, que se usa en sistemas de tiempo real en los que se requiere garantía de tiempos de respuesta de interacción.
2. Arquitectura cliente-servidor de dos niveles, que se usa para sistemas cliente-servidor simple, y en situaciones donde es importante centralizar el sistema por razones de seguridad. En tales casos, la comunicación entre el cliente y el servidor por lo general está encriptada.
3. Arquitectura cliente-servidor multinivel, que se usa cuando existe un enorme volumen de transacciones a procesar por el servidor.
4. Arquitectura de componentes distribuidos, que se usa cuando es necesario combinar los recursos de diferentes sistemas y bases de datos, o como un modelo de implementación para sistemas cliente-servidor multinivel.
5. Arquitectura *peer-to-peer* (entre pares o punto a punto, o par a par), que se usa cuando los clientes intercambian de manera local la información almacenada, y el papel del servidor es presentar a los clientes entre sí. También puede usarse cuando se deba elaborar un amplio número de cálculos independientes.

18.3.1 Arquitecturas maestro-esclavo

Las arquitecturas maestro-esclavo para sistemas distribuidos se usan comúnmente en sistemas de tiempo real donde puede haber procesadores separados asociados con la

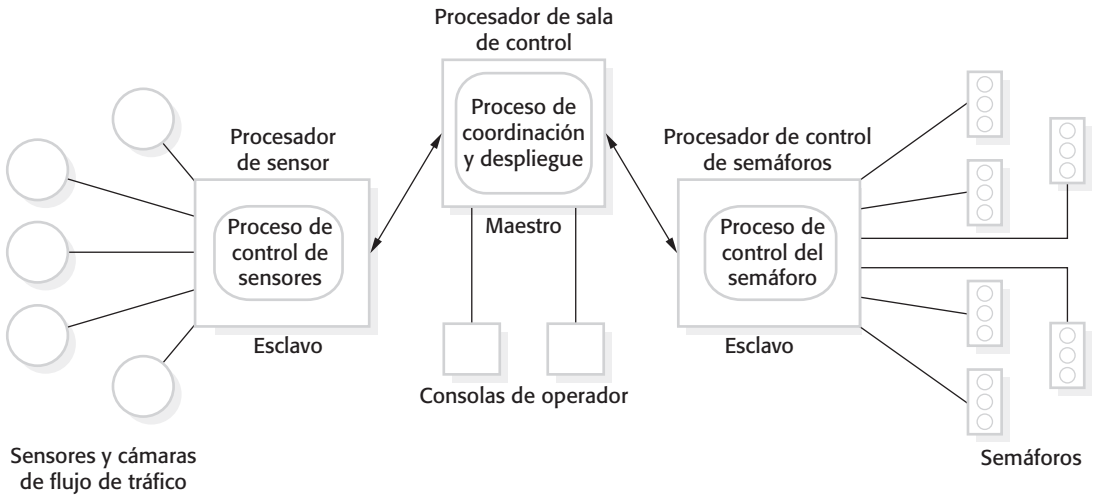


Figura 18.7 Sistema de gestión de tráfico con una arquitectura maestro-esclavo

adquisición de datos del entorno del sistema, procesamiento de datos y computación y actuador de gestión. Los actuadores, como se estudia en el capítulo 20, son dispositivos controlados mediante el sistema de software que actúan para cambiar el entorno del sistema. Por ejemplo, un actuador puede controlar una válvula y cambiar su estado de "abierto" a "cerrado". El proceso "maestro" es por lo general el responsable de la computación, la coordinación y las comunicaciones, y controla los procesos "esclavos". Estos procesos esclavos se dedican a acciones específicas, tales como la adquisición de datos de un arreglo de sensores.

La figura 18.7 ilustra este modelo arquitectónico. Es un modelo de un sistema de control de tráfico en una ciudad y tiene tres procesos lógicos que se ejecutan en procesadores independientes. El proceso maestro es el proceso de sala de control, que se comunica con procesos esclavo independientes que son responsables de recolectar datos de tráfico y administrar la operación de los semáforos.

Un conjunto de sensores distribuidos recolecta información acerca del flujo del tránsito. El proceso de control de sensores consulta los sensores periódicamente para capturar la información del flujo del tránsito y coteja esta información para mayor procesamiento. El procesador de sensores en sí se consulta habitualmente para obtener información por parte del proceso maestro, que se ocupa de desplegar el estatus de tránsito a los operadores, calcular las secuencias de los semáforos y aceptar los comandos del operador para modificar dichas secuencias. El sistema de sala de control envía comandos a un proceso de control de semáforos que los convierte en señales para controlar el hardware de los semáforos. El sistema maestro de sala de control está organizado como un sistema cliente-servidor, y los procesos cliente se ejecutan en las consolas del operador.

Este modelo maestro-esclavo de sistema distribuido se usa en situaciones en que es posible predecir el procesamiento distribuido que se requiere, y en que el procesamiento puede asignarse fácilmente a procesadores esclavos. Esta situación es común en los sistemas en tiempo real, en los que es importante cumplir con los plazos de procesamiento. Los procesadores esclavos pueden usarse para operaciones de cómputo intensivo, como el procesamiento de señales y la administración de equipo controlado por el sistema.

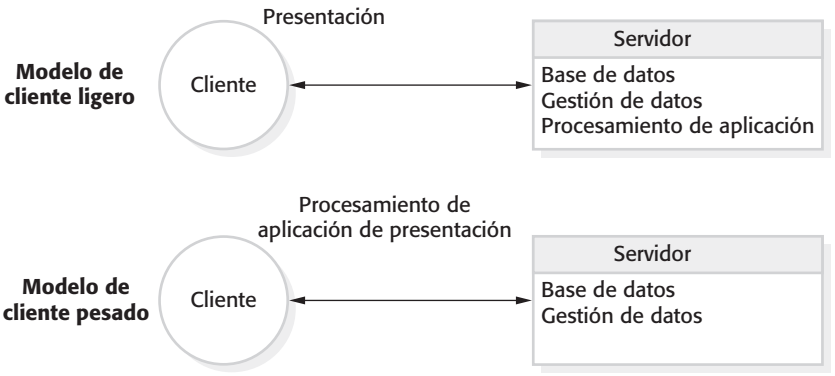


Figura 18.8 Modelos arquitectónicos de cliente pesado y cliente ligero

18.3.2 Arquitecturas cliente-servidor de dos niveles

En la sección 18.2 se explicó la forma general de los sistemas cliente-servidor en los que parte del sistema de aplicación se ejecuta en la computadora del usuario (el cliente) y parte se ejecuta en una computadora remota (el servidor). También se presentó un modelo de aplicación en capas (figura 18.6) en el que las diferentes capas del sistema pueden ejecutarse en diferentes computadoras.

Una arquitectura cliente-servidor de dos niveles es la forma más simple de arquitectura cliente-servidor. El sistema se implementa como un solo servidor lógico más un número indefinido de clientes que usan dicho servidor. Esto se ejemplifica en la figura 18.8, que indica dos formas de este modelo arquitectónico:

1. Un modelo de cliente ligero, en que la capa de presentación se implementa en el cliente, y todas las otras capas (gestión de datos, procesamiento de la aplicación y bases de datos) se implementan en un servidor. El software cliente puede ser un programa escrito especialmente en el cliente para manejar la presentación. Sin embargo, se usa con más frecuencia un navegador Web en la computadora cliente para presentar los datos.
2. Un modelo de cliente pesado, en que parte o todo el procesamiento de la aplicación se realiza en el cliente. Las funciones de gestión de datos y de base de datos se implementan en el servidor.

La ventaja del modelo de cliente ligero consiste en que es simple de manejar por los clientes. Esto representa un gran conflicto si existe un considerable número de clientes, pues puede ser difícil y costoso instalar un nuevo software en todos ellos. Si se usa un navegador Web como el cliente, no hay necesidad de instalar software alguno.

No obstante, la desventaja del enfoque de cliente ligero es que puede colocarse una fuerte carga de procesamiento tanto en el servidor como en la red. El servidor es responsable de toda la computación y esto puede conducir a la generación de tráfico de red significativo entre el cliente y el servidor. Por lo tanto, implementar un sistema que use este modelo puede requerir inversión adicional en capacidad de red y servidor. Sin embargo, los navegadores pueden realizar parte del procesamiento local al ejecutar scripts (por ejemplo, Javascript) en la página Web a la que se ingresa mediante el navegador.

El modelo de cliente pesado utiliza el poder de procesamiento disponible en la computadora que ejecuta el software cliente, y distribuye parte o todo el procesamiento de la aplicación y la presentación al cliente. En esencia, el servidor es un servidor de transacción

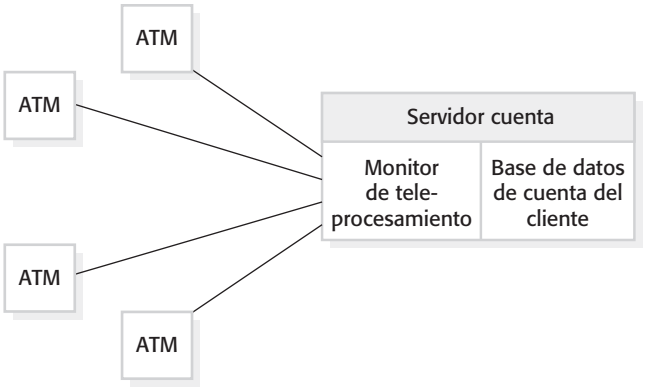


Figura 18.9 Arquitectura de cliente pesado para un sistema de cajero automático

que gestiona todas las transacciones de la base de datos. La gestión de datos es directa, pues no hay necesidad de gestionar la interacción entre el cliente y el sistema de procesamiento de la aplicación. Desde luego, el problema con el modelo de cliente pesado es que requiere gestión de sistema adicional para implementar y mantener el software en la computadora cliente.

Un ejemplo de una situación en la que se usa una arquitectura de cliente pesado es un sistema de cajero automático, que entrega efectivo y otros servicios bancarios a los usuarios. El cajero es la computadora cliente y el servidor es, por lo general, un mainframe que opera la base de datos de cuentas de los clientes. Una computadora mainframe es una máquina poderosamente diseñada para procesamiento de transacciones. Por consiguiente, es capaz de manejar el gran volumen de las transacciones generadas por los cajeros automáticos, otros sistemas de cajeros y banca en línea. El software en el cajero realiza mucho procesamiento relacionado con el cliente asociado con una transacción.

La figura 18.9 muestra una versión simplificada de la organización del sistema del cajero automático. Observe que los cajeros no se conectan directamente a la base de datos cliente, sino que, en vez de ello, se conecta a un monitor de teleprocesamiento. Un monitor de teleprocesamiento (TP) es un sistema middleware que organiza las comunicaciones con clientes remotos y pone en serie las transacciones del cliente para su procesamiento de la base de datos. Esto garantiza que las transacciones sean independientes y no interfieran unas con otras. Usar transacciones seriales significa que el sistema puede recuperarse de fallas sin corromper los datos del sistema.

Mientras que un modelo de cliente pesado distribuye el procesamiento más efectivamente que un modelo de cliente ligero, es más compleja la gestión del sistema. La funcionalidad de la aplicación se dispersa a través de muchas computadoras. Cuando el software de aplicación debe cambiarse, esto implica la reinstalación en cada computadora cliente. Esto podría representar un costo mayor si existen cientos de clientes en el sistema. Quizá se deba diseñar el sistema para soportar actualizaciones de software remoto y tal vez sea necesario desactivar todos los servicios del sistema hasta sustituir el software cliente.

18.3.3 Arquitecturas cliente-servidor multinivel

El problema fundamental con un enfoque cliente-servidor de dos niveles es que las capas lógicas del sistema (presentación, procesamiento de aplicación, gestión de datos y base de datos) deben mapearse en dos sistemas de cómputo: el cliente y el servidor. Esto puede conducir a problemas con la escalabilidad y el rendimiento si se elige el modelo de

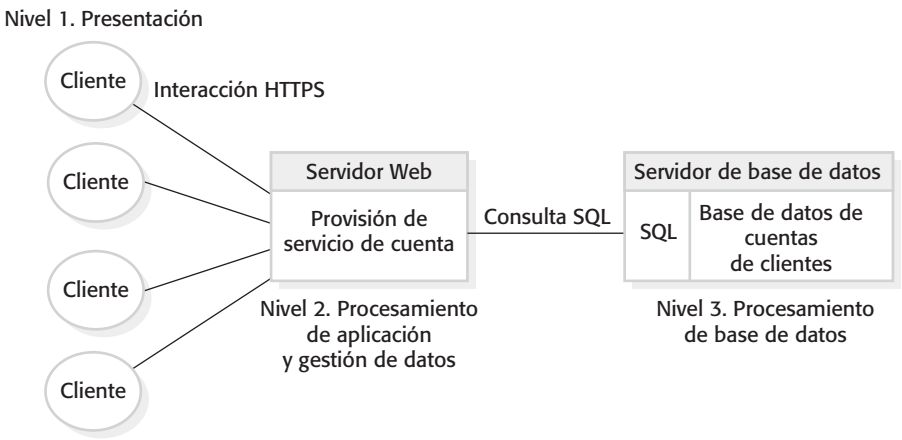


Figura 18.10
Arquitectura de tres niveles para un sistema de banca por Internet

cliente ligero, o problemas de gestión de sistema si se usa el modelo de cliente pesado. Para evitar algunos de estos problemas, se puede usar una arquitectura "cliente-servidor multinivel". En esta arquitectura, las diferentes capas del sistema (esto es, presentación, gestión de datos, procesamiento de aplicación y base de datos) son procesos separados que pueden ejecutarse en diferentes procesadores.

Un sistema de banca por Internet (figura 18.10) es un ejemplo de una arquitectura cliente-servidor multinivel, donde existen tres niveles en el sistema. La base de datos de clientes del banco (por lo general ubicada en una computadora mainframe, como se expuso anteriormente) brinda servicios de base de datos. Un servidor Web ofrece servicios de gestión de datos, tales como generación de páginas Web y algunos servicios de aplicación. Los servicios de aplicación, tales como las instalaciones para transferir efectivo, generar estados de cuenta, pagar facturas y así sucesivamente, se implementan en el servidor Web y como scripts que se ejecutan por el cliente. La propia computadora del cliente con un navegador es el cliente. Este sistema es escalable porque es relativamente sencillo agregar servicios (ampliar) conforme aumenta el número de clientes.

En este caso, el uso de una arquitectura de tres niveles permite optimizar la transferencia de información entre el servidor Web y el servidor de base de datos. Las comunicaciones entre dichos sistemas pueden usar rápidos protocolos de intercambio de datos de bajo nivel. Para manejar la recuperación de información de la base de datos, se usa middleware eficiente que soporta consultas de base de datos en lenguaje de consulta estructurado (SQL, por las siglas de *Structured Query Language*).

El modelo cliente-servidor de tres niveles puede extenderse a una variante multinivel, donde se agregan servidores adicionales al sistema. Esto podría implicar el uso de un servidor Web para gestión de datos y servidores separados para procesamiento de aplicación y servicios de base de datos. Los sistemas multinivel también pueden usarse cuando las aplicaciones necesitan tener acceso y utilizar datos de diferentes bases de datos. En este caso, tal vez se requiera agregar un servidor de integración al sistema. El servidor de integración recolecta los datos distribuidos y los presenta al servidor de aplicación como si fuera de una sola base de datos. Como se estudiará en la siguiente sección, las arquitecturas de componentes distribuidos pueden usarse para implementar sistemas cliente-servidor multinivel.

Los sistemas cliente-servidor multinivel que distribuyen el procesamiento de aplicación a través de varios servidores son inherentemente más escalables que las arquitecturas de dos niveles. El procesamiento de aplicación con frecuencia es la parte más volátil del

Arquitectura	Aplicaciones
Arquitectura cliente-servidor de dos niveles con clientes ligeros	Aplicaciones de sistema heredado que se usan cuando no es práctica la separación de procesamiento de aplicación y gestión de datos. Los clientes pueden tener acceso a ellos como servicios, tal como se refiere en la sección 18.4. Aplicaciones de cómputo intensivo, como los compiladores con poca o ninguna gestión de datos. Aplicaciones intensivas en datos (navegación y consulta) con procesamiento de aplicación no intensivo. Navegar la Web es el ejemplo más común de una situación donde se emplea esta arquitectura.
Arquitectura cliente-servidor de dos niveles con clientes pesados	Aplicaciones donde el procesamiento de aplicación lo proporciona software comercial (por ejemplo, Microsoft Excel) en el cliente. Aplicaciones en que se requiere procesamiento de datos de cómputo intensivo (por ejemplo, visualización de datos). Aplicaciones móviles en las que no puede garantizarse la conectividad a Internet. Por lo tanto, es posible algún procesamiento local usando información en caché de la base de datos.
Arquitectura cliente-servidor multinivel	Aplicaciones grandes con cientos o miles de clientes. Aplicaciones en que tanto los datos como la aplicación son volátiles. Aplicaciones en que se integran datos de fuentes múltiples.

Figura 18.11 Uso de patrones arquitectónicos cliente-servidor

sistema y puede actualizarse fácilmente, ya que se ubica centralmente. El procesamiento, en algunos casos, puede distribuirse entre la lógica de la aplicación y los servidores de gestión de datos; por lo tanto, conduce a una respuesta más rápida a las solicitudes del cliente.

Los diseñadores de arquitecturas cliente-servidor deben tomar en cuenta varios factores cuando elijan la arquitectura de distribución más adecuada. En la figura 18.11 se describen las situaciones en las que es probable que sean adecuadas las arquitecturas cliente-servidor discutidas aquí.

18.3.4 Arquitecturas de componentes distribuidos

Al organizar el procesamiento en capas, como se muestra en la figura 18.6, cada capa de un sistema puede implementarse como un servidor lógico separado. Este modelo funciona bien para muchos tipos de aplicación. Sin embargo, limita la flexibilidad de los diseñadores del sistema en cuanto a que deben decidir cuáles servicios incluir en cada capa. No obstante, en la práctica, no siempre es claro si un servicio es de gestión de datos, uno de aplicación o uno de base de datos. Los diseñadores también deben planear la escalabilidad y así ofrecer ciertos medios para que los servidores se repliquen conforme se agregan más clientes al sistema.

Un enfoque más general al diseño de sistemas distribuidos es diseñar el sistema como un conjunto de servicios, sin tratar de asignar dichos servicios a capas en el sistema. Cada servicio, o grupo de servicios relacionados, se implementa usando un componente separado. En una arquitectura de componentes distribuida (figura 18.12), el sistema está organizado como un conjunto de componentes u objetos en interacción. Dichos componentes proporcionan una interfaz a un conjunto de servicios que ofrecen. Otros componentes

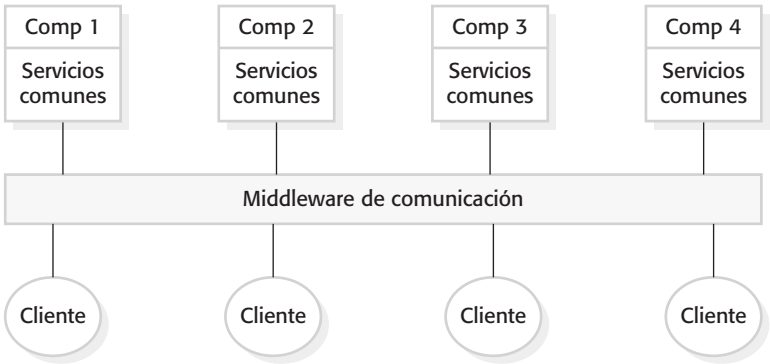


Figura 18.12 Arquitectura de componentes distribuida

solicitan dichos servicios a través de middleware, usando solicitudes de procedimiento remoto o llamadas a métodos.

Los sistemas de componentes distribuidos dependen del middleware, que gestiona las interacciones de componentes, reconcilia las diferencias entre tipos de parámetros transmitidos entre componentes, y ofrece un conjunto de servicios comunes que pueden usar los componentes de la aplicación. CORBA (Orfali *et al*., 1997) fue un ejemplo temprano de tal middleware, pero ahora no se usa ampliamente. Fue sustituido sobre todo por software propietario como Enterprise Java Beans (EJB) o .NET.

Los beneficios de usar un modelo de componentes distribuidos para implementar sistemas distribuidos son los siguientes:

1. Permite al diseñador del sistema demorar las decisiones acerca de dónde y cómo deben proporcionarse los servicios. Los componentes que brindan servicios pueden ejecutarse en cualquier nodo de la red. No hay necesidad de decidir por adelantado si un servicio es parte de una capa de gestión de datos, de una capa de aplicación, etcétera.
2. Es una arquitectura de sistema muy abierta que permite adicionar nuevos recursos conforme se requiera. Pueden agregarse fácilmente nuevos servicios de sistema sin grandes perturbaciones al sistema existente.
3. El sistema es flexible y escalable. Pueden añadirse nuevos componentes o componentes replicados a medida que aumente la carga del sistema, sin perturbar otras partes de éste.
4. Es posible, según se requiera, reconfigurar dinámicamente el sistema con componentes que migran a través de la red. Esto puede ser importante donde existan patrones fluctuantes de demanda de servicios. Un componente de prestación de servicios puede migrar hacia el mismo procesador que solicitó el servicio como un objeto, por lo que mejora el rendimiento del sistema.

Una arquitectura de componentes distribuidos puede usarse como un modelo lógico que permita estructurar y organizar el sistema. En este caso, piense en cómo ofrecer funcionalidad a la aplicación exclusivamente en términos de servicios y combinaciones de servicios. Entonces calcule cómo ofrecer dichos servicios mediante un conjunto de componentes distribuidos. Por ejemplo, en una aplicación de ventas, puede haber

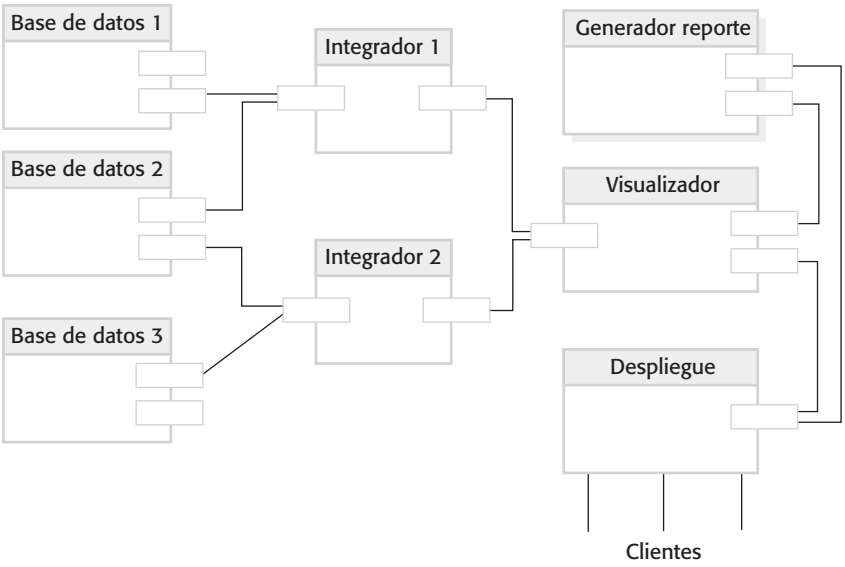


Figura 18.13
Arquitectura de componentes distribuida para un sistema de minería de datos

componentes de aplicación que se ocupen del control de almacenes, comunicaciones con los clientes, pedidos, etcétera.

Los sistemas de minería de datos son un buen ejemplo de un tipo de sistema en el que una arquitectura de componentes distribuidos es el mejor patrón arquitectónico a usar. Un sistema de minería de datos busca relaciones entre los datos que se almacenan en algunas bases de datos (figura 18.13). Los sistemas de minería de datos por lo general obtienen información de varias bases de datos separadas, realizan procesamiento intensivo de cómputo y despliegan sus resultados en forma gráfica.

Un ejemplo de tal aplicación de minería de datos puede ser un sistema para un negocio que vende alimentos y libros. El departamento de marketing quiere encontrar relaciones entre las compras de alimentos y libros de un cliente. Por ejemplo, una proporción relativamente alta de personas que compran pizzas quizá también compren novelas policiacas. Con este conocimiento, la empresa puede dirigirse específicamente a clientes que realicen compras de alimentos específicas con información acerca de nuevas novelas cuando se publican.

En este ejemplo, cada base de datos de ventas puede encapsularse como un componente distribuido con una interfaz que proporcione acceso de sólo lectura a sus datos. Los componentes integrador se dedican cada uno a tipos específicos de relaciones, y recolectan información de todas las bases de datos para tratar de deducir las relaciones. Podría haber un componente integrador que se ocupe de las variaciones estacionales de los bienes vendidos, y otro que trate con las relaciones entre diferentes tipos de bienes.

Los componentes visualizador interactúan con los componentes integrador para producir una visualización de un reporte acerca de las relaciones que se descubrieron. Debido a los grandes volúmenes de datos que se manejan, los componentes visualizador con frecuencia presentan sus resultados gráficamente. Para concluir, un componente despliegue puede ser el responsable de la entrega de los modelos gráficos a los clientes para su presentación final.

Una arquitectura de componentes distribuidos, en lugar de una arquitectura en capas, es adecuada para este tipo de aplicaciones, porque es posible agregar nuevas bases de

datos al sistema sin causar grandes perturbaciones. A cada nueva base de datos se accede simplemente al agregar otro componente distribuido. Los componentes de acceso a la base de datos proporcionan una interfaz simplificada que controla el acceso a los datos. Las bases de datos a las que se ingresa pueden residir en diferentes máquinas. La arquitectura facilita también extraer nuevos tipos de relaciones al agregar nuevos componentes integrador.

Las arquitecturas de componentes distribuidos enfrentan dos grandes desventajas:

1. Son más complejas de diseñar que los sistemas cliente-servidor. Los sistemas cliente-servidor multinivel parecen ser una forma bastante intuitiva de pensar en los sistemas. Ellos reflejan muchas transacciones humanas donde las personas solicitan y reciben servicios de otras personas que se especializan en ofrecer dichos servicios. En contraste, para las personas, las arquitecturas de componentes distribuidos son más difíciles de visualizar y entender.
2. El middleware estandarizado para sistemas de componentes distribuidos nunca se ha aceptado por la comunidad. En su lugar, distintos proveedores, como Microsoft y Sun, han desarrollado un middleware diferente e incompatible. El middleware incrementa la complejidad general en los componentes de los sistemas distribuidos.

Como resultado de estos problemas, las arquitecturas orientadas a servicios (que se estudian en el capítulo 19) sustituyen a las arquitecturas de componentes distribuidos en muchas situaciones. Sin embargo, los sistemas de componentes distribuidos tienen beneficios de rendimiento sobre los sistemas orientados a servicios. Las comunicaciones RPC, por lo general, son más rápidas que la interacción basada en mensajes que se usa en los sistemas orientados a servicios. Por lo tanto, las arquitecturas basadas en componentes son más adecuadas para sistemas de alto rendimiento global en los que un gran número de transacciones debe procesarse rápidamente.

18.3.5 Arquitecturas entre pares (peer-to-peer)

El modelo cliente-servidor de computación que se analizó en las secciones anteriores del capítulo hace una clara distinción entre servidores, que son los proveedores de servicios, y clientes, que son los receptores de los servicios. Este modelo conduce regularmente a una distribución desigual de la carga en el sistema, en el que los servidores realizan más trabajo que los clientes. Esto puede conducir a que las organizaciones gasten mucho en capacidad de servidor mientras que existe una capacidad de procesamiento no utilizada en los cientos o miles de PC que se usan para acceder a los servidores del sistema.

Los sistemas entre pares (o punto a punto, p2p) son sistemas descentralizados en los que los cálculos pueden realizarse en cualquier nodo de la red. Al menos en principio, no se hacen distinciones entre clientes y servidores. En las aplicaciones entre pares, el sistema global está diseñado para sacar ventaja del poder computacional y de almacenamiento disponible a través de una red de computadoras potencialmente enorme. Los estándares y protocolos que permiten las comunicaciones a través de los nodos se embeben en la aplicación en sí y cada nodo debe ejecutar una copia de dicha aplicación.

Las tecnologías par a par se han usado principalmente para sistemas personales más que empresariales (Oram, 2001). Por ejemplo, los sistemas para compartir archivos basa-

dos en los protocolos Gnutella y BitTorrent se usan para intercambiar archivos en las PC de los usuarios. Los sistemas de mensajería instantánea como ICQ y Jabber proporcionan comunicaciones directas entre usuarios sin un servidor intermediario. SETI@home es un proyecto de hace mucho tiempo que procesa datos de radiotelescopios en computadoras domésticas para buscar indicios de vida extraterrestre. Freenet es una base de datos descentralizada diseñada para facilitar la publicación de información de manera anónima, y dificultar que las autoridades supriman esta información. Los servicios telefónicos voz sobre IP (*Voice over IP*, VOIP), como Skype, dependen de la comunicación par a par entre los participantes en la llamada o conferencia telefónica.

Sin embargo, los sistemas par a par se usan también en las empresas para aprovechar el poder en sus redes de computadoras (McDougall, 2000). Intel y Boeing implementaron sistemas p2p para aplicaciones de cómputo intensivo. Esto saca ventaja de la capacidad de procesamiento no utilizada de computadoras locales. En lugar de comprar un costoso hardware de alto rendimiento, los cálculos de ingeniería pueden ejecutarse durante la noche cuando no se utilizan las computadoras de escritorio. Las empresas también usan de manera extensa los sistemas p2p comerciales, tales como los sistemas de mensajería y VOIP.

Es adecuado usar un modelo arquitectónico punto a punto para un sistema en dos circunstancias:

1. Donde el sistema es de cómputo intensivo y es posible separar el procesamiento requerido en un gran número de cálculos independientes. Por ejemplo, un sistema punto a punto que soporta el descubrimiento computacional de medicamentos distribuye los cálculos que buscan tratamientos potenciales de cáncer al analizar un enorme número de moléculas para ver si tienen las características requeridas para suprimir el crecimiento de tumores cancerosos. Cada molécula puede considerarse por separado, de modo que no hay necesidad de que los pares en el sistema se comuniquen.
2. Donde el sistema principalmente implica el intercambio de información entre computadoras individuales en una red y no hay necesidad de que esta información se almacene o gestione de manera centralizada. Los ejemplos de tales aplicaciones incluyen sistemas para compartir archivos que permiten a los pares intercambiar archivos locales, tales como música y películas, y sistemas telefónicos que soportan comunicaciones de voz y video entre computadoras.

En principio, cada nodo en una red p2p podría estar al tanto de cualquier otro nodo. Los nodos podrían conectarse e intercambiar datos directamente con cualquier otro nodo en la red. En la práctica, desde luego, esto es imposible, pues los nodos están organizados en "localidades" con algunos nodos actuando como puentes hacia otras localidades de nodos. La figura 18.14 muestra esta arquitectura p2p descentralizada.

En una arquitectura descentralizada, los nodos en la red no son simplemente elementos funcionales, sino también conmutadores de comunicaciones que pueden dirigir datos y controlar señales de un nodo a otro. Por ejemplo, suponga que la figura 18.14 representa un sistema descentralizado de gestión de documentos. Este sistema lo utiliza un consorcio de investigadores para compartir documentos, y cada miembro del consorcio mantiene su propio almacén de documentos. Sin embargo, cuando se recupera un documento, el nodo que lo recupera también lo pone a disposición de otros nodos.

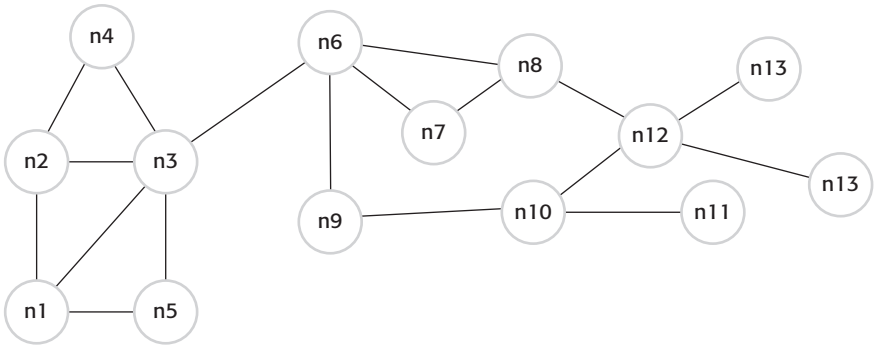


Figura 18.14
Arquitectura p2p descentralizada

Si alguien necesita un documento que se almacene en alguna otra parte de la red, ellos emiten un comando de búsqueda, que se envía a los nodos en su "localidad". Dichos nodos comprueban si tienen el documento y, si es así, lo devuelven al solicitante. Si no lo tienen, dirigen la búsqueda hacia otros nodos. Por lo tanto, si n1 emite una búsqueda para un documento que se almacena en n10, esta búsqueda se dirige a través de los nodos n3, n6 y n9 hasta n10. Cuando finalmente se encuentra el documento, el nodo que contiene el documento lo envía al nodo solicitante directamente al hacer una conexión punto a punto.

Esta arquitectura descentralizada tiene ventajas en cuanto a que es considerablemente redundante y, por consiguiente tolerante, a fallas y tolerante a los nodos que se desconectan de la red. Sin embargo, aquí las desventajas son que muchos nodos diferentes pueden procesar la misma búsqueda, y también existe significativa carga de trabajo al replicar las comunicaciones de los pares.

Un modelo arquitectónico p2p alternativo, que parte de una arquitectura p2p pura, es una arquitectura semicentralizada donde, dentro de la red, uno o más nodos actúan como servidores para facilitar las comunicaciones entre nodos. Esto reduce la cantidad de tráfico entre nodos. La figura 18.15 ilustra este modelo.

En una arquitectura semicentralizada, el papel del servidor (llamado en ocasiones superpar) es ayudar a establecer contacto entre pares en la red, o coordinar los resultados de un cálculo. Por ejemplo, si la figura 18.15 representa un sistema de mensajería instantánea, entonces los nodos de la red se comunican con el servidor (indicado mediante las líneas punteadas) para descubrir qué otros nodos están disponibles. Una vez descubiertos dichos nodos, pueden establecerse comunicaciones directas y la conexión al servidor no es necesaria. Por lo tanto, los nodos n2, n3, n5 y n6 están en comunicación directa.

En un sistema computacional p2p, donde un cómputo intensivo de procesador se distribuye a través de un gran número de nodos, es normal que algunos nodos sean superpares. Su papel es distribuir el trabajo hacia otros nodos y cotejar y verificar los resultados de la computación.

Las arquitecturas par a par permiten el uso eficiente de la capacidad a través de la red. Sin embargo, las principales preocupaciones que inhiben su uso son los conflictos de seguridad y confianza. Las comunicaciones par a par implican abrir la computadora a interacciones directas con otros pares y esto significa que dichos sistemas podrían, potencialmente, acceder a cualquiera de sus recursos. Para contrarrestar lo anterior, es necesario organizar el sistema de manera que dichos recursos estén protegidos. Si esto se hace de manera incorrecta, entonces su sistema puede ser inseguro.

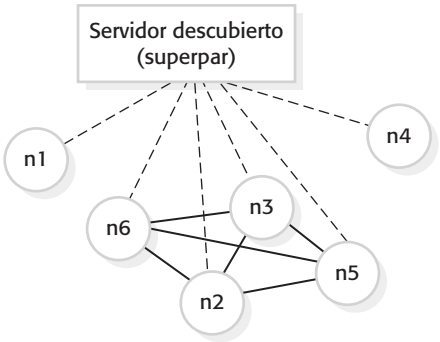


Figura 18.15
Arquitectura p2p semicentralizada

También pueden ocurrir problemas cuando los pares en una red deliberadamente se comportan en forma maliciosa. Por ejemplo, se han reportado casos donde compañías musicales que consideraban que sus derechos de autor eran violados, deliberadamente crearon "pares envenenados" disponibles. Cuando otro par descarga lo que considera que es una pieza de música, el archivo real entregado es malware que puede ser una versión deliberadamente corrompida de la música o una advertencia al usuario de violaciones al derecho de autor.

18.4 Software como servicio

En las secciones anteriores se estudiaron los modelos cliente-servidor y cómo se distribuye la funcionalidad entre el cliente y el servidor. Para implementar un sistema cliente-servidor, tal vez se tenga que instalar un programa en la computadora cliente, que se comunique con el servidor, aplique funcionalidad en el lado del cliente y gestione la interfaz de usuario. Por ejemplo, un cliente de correo, como Outlook o Mac Mail, proporciona características de gestión de correo en su computadora. Esto evita el problema de algunos sistemas de cliente ligero, en que todo el procesamiento se realiza en el servidor.

Sin embargo, los problemas de la sobrecarga del servidor pueden reducirse significativamente al usar un navegador moderno como el software cliente. Las tecnologías Web, como AJAX (Holdener, 2008), soportan gestión eficiente de presentación de página Web y computación local mediante scripts. Esto significa que un navegador puede configurarse y usarse como cliente, con significativo procesamiento local. El software de aplicación puede considerarse tal como un servicio remoto, al que puede accederse desde cualquier dispositivo capaz de ejecutar un navegador estándar. Ejemplos bien conocidos son los sistemas de correo basados en Web, como Yahoo! y Gmail, y aplicaciones de oficina, como Google Docs.

Esta noción de SaaS implica alojar el software remotamente y proporcionar acceso al mismo a través de Internet. Los elementos clave de SaaS son los siguientes:

1. El software se despliega en un servidor (o, más comúnmente, en algunos servidores) y se accede a él a través de un navegador Web. No se implementa en una computadora local.
2. El software es propiedad de un proveedor de software, quien lo administra, en lugar de las organizaciones que usan el software.

3. Los usuarios pueden pagar por el software de acuerdo con la cantidad de uso que hagan de él o mediante una suscripción anual o mensual. En ocasiones, el software es gratuito para quien lo utilice, pero entonces los usuarios deben estar de acuerdo en aceptar publicidad, que financia el servicio del software.

Para usuarios de software, el beneficio de SaaS es que los costos para administrar el software se transfieren al proveedor. El proveedor es responsable de corregir los bugs e instalar las actualizaciones de software, enfrentar los cambios a la plataforma del sistema operativo y asegurar que la capacidad del hardware pueda cumplir la demanda. Los costos de gestión de licencia del software son iguales a cero. Si alguien tiene muchas computadoras, no hay necesidad de autorizar las licencias del software para todas ellas. Si una aplicación de software sólo se usa ocasionalmente, el modelo "pago por uso" puede ser más barato que comprar una aplicación. Es posible acceder al software desde dispositivos móviles, tales como los teléfonos inteligentes (*smart phones*), desde cualquier parte del mundo.

Por supuesto, este modelo de provisión de software tiene algunas desventajas. El principal problema, posiblemente, es el costo de transferir datos al servicio remoto. La transferencia de datos se da de acuerdo con las velocidades de la red, por lo que transferir una gran cantidad de datos puede tardar mucho tiempo. También es posible que se deba pagar al proveedor del servicio de acuerdo con la cantidad transferida. Otros problemas son la falta de control sobre la evolución del software (el proveedor puede cambiar el software cuando lo desee) y problemas con la legislación y las regulaciones. Muchos países tienen leyes que regulan el almacenamiento, gestión, conservación y accesibilidad de los datos, y transferir datos a un servicio remoto puede quebrantar dichas leyes.

La noción de SaaS y arquitecturas orientadas a servicios (SOA, por las siglas de *service-oriented architectures*), que se estudian en el capítulo 19, evidentemente se relacionan, aunque no son lo mismo:

1. SaaS es una forma de proporcionar funcionalidad en un servidor remoto, con acceso de clientes mediante un navegador Web. El servidor conserva los datos y el estado del usuario durante una sesión de interacción. Por lo regular, las transacciones son largas (por ejemplo, la edición de un documento).
2. SOA es un enfoque a la estructuración de un sistema de software como un conjunto de servicios independientes, sin estado. Éstos pueden proporcionarse mediante múltiples proveedores y distribuirse. Por lo general, las transacciones son transacciones cortas donde se solicita un servicio, se hace algo y luego se devuelve el resultado.

SaaS es una forma de entregar funcionalidad de aplicación a los usuarios, mientras que SOA es una tecnología de implementación para sistemas de aplicación. La funcionalidad implementada con el uso de SOA no necesita aparecer a los usuarios como servicios. De igual modo, los servicios de usuario no tienen que implementarse usando SOA. Sin embargo, si SaaS se implementa mediante SOA, es posible que las aplicaciones usen API de servicio para acceder a la funcionalidad de otras aplicaciones. Entones pueden integrarse en sistemas más complejos. A éstos se les llama *mashups* (remezcla o aplicación híbrida) y representan otro enfoque a la reutilización de software y el desarrollo de software rápido.

Desde una perspectiva de desarrollo de software, el proceso de desarrollo de servicios tiene mucho en común con otros tipos de desarrollo de software. Sin embargo, la construcción de servicio, por lo regular, no está dirigida por los requerimientos del usuario,

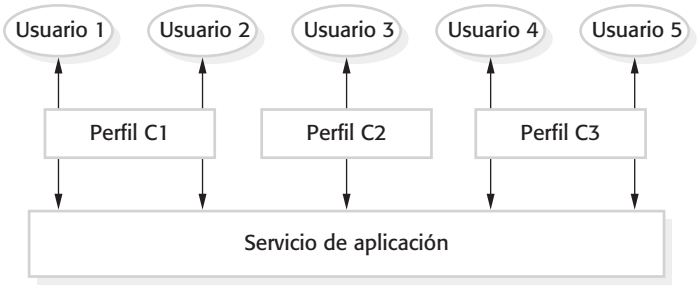


Figura 18.16
Configuración de un sistema de software ofrecido como servicio

sino por las suposiciones del proveedor del servicio acerca de lo que necesitan los usuarios. Por lo tanto, es preciso que el software pueda evolucionar rápidamente después de que el proveedor obtiene retroalimentación de los usuarios acerca de sus requerimientos. Por consiguiente, el desarrollo ágil con entrega incremental es un enfoque usado comúnmente para el software que debe implementarse como servicio.

Cuando implemente SaaS debe considerar que puede tener usuarios del software en varias organizaciones diferentes. Debe tener en cuenta tres factores:

1. *Configurabilidad* ¿Cómo configura usted el software para los requerimientos específicos de cada organización?
2. *Multitenencia* ¿Cómo presenta a cada usuario del software la impresión de que trabaja con su propia copia del sistema mientras, al mismo tiempo, hace uso eficiente de los recursos del sistema?
3. *Escalabilidad* ¿Cómo diseña el sistema de modo que pueda escalarse para alojar un número impredeciblemente grande de usuarios?

La noción de arquitecturas de línea de producto, que se expuso en el capítulo 16, es una forma de configurar el software para usuarios con requerimientos que se traslapan, aunque no son idénticos. Usted comienza con un sistema genérico y lo adapta de acuerdo con los requerimientos específicos de cada usuario.

Sin embargo, esto no funciona para SaaS, pues significaría implementar una copia diferente del servicio para cada organización que use el software. En vez de ello, es necesario diseñar la configurabilidad en el sistema y proporcionar una interfaz de configuración que permita a los usuarios especificar sus preferencias. Luego esto se usa para ajustar dinámicamente el comportamiento del software conforme se utilice. Las instalaciones de configuración pueden permitir lo siguiente:

1. Marca: A los usuarios de cada organización se les presenta una interfaz que refleja su propia organización.
2. Reglas y flujos de trabajo empresariales: Cada organización define sus propias reglas que regulan el uso del servicio y sus datos.
3. Extensiones de base de datos: Cada organización define cómo se extiende el modelo de datos del servicio genérico para cubrir sus necesidades específicas.
4. Control de acceso: Los clientes del servicio crean cuentas individuales para su personal y definen los recursos y funciones que son accesibles a cada uno de sus usuarios.

Tenencia	Clave	Nombre	Dirección
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figura 18.17 Base de datos multitenencia

La figura 18.16 ilustra esta situación. Este diagrama muestra cinco usuarios del servicio de aplicación, quienes trabajan para tres clientes diferentes del proveedor del servicio. Los usuarios interactúan con los servicios mediante un perfil de cliente que define la configuración del servicio para su empleador.

La multitenencia es una situación en la que muchos usuarios diferentes acceden al mismo sistema y la arquitectura del sistema se define para permitir que el hecho de compartir los recursos del sistema sea eficiente. Sin embargo, debe parecer a cada usuario que él tiene el uso exclusivo del sistema. La multitenencia implica diseñar el sistema de modo que exista una separación absoluta entre la funcionalidad y los datos del sistema. Por lo tanto, se debe diseñar el sistema de manera que todas las operaciones sean sin estados. Los datos deben proporcionarse por parte del cliente o estar disponibles en un sistema de almacenamiento o base de datos al que pueda accederse desde cualquier instancia del sistema. Las bases de datos relacionales no son ideales para proporcionar multitenencia, y los grandes proveedores de servicio, tales como Google, implementaron bases de datos más simples para datos de usuarios.

Un problema particular en los sistemas multitenencia es la gestión de datos. La forma más sencilla de proporcionar gestión de datos es que cada cliente tenga su propia base de datos, la cual pueden usar y configurar como desee. Sin embargo, esto requiere que el proveedor del servicio conserve numerosas instancias de base de datos diferentes (una por cliente) y las ponga a disposición de la demanda. Esto es ineficiente en términos de capacidad de servidor y aumenta el costo global del servicio.

Como alternativa, el proveedor del servicio puede usar una sola base de datos en la que diferentes usuarios estén prácticamente aislados dentro de dicha base de datos. Esto se ilustra en la figura 18.17, donde se observa que las entradas de la base de datos tienen también un "identificador de tenencia", que vincula dichas entradas a usuarios específicos. Al usar vistas de base de datos, es posible extraer las entradas para cada cliente del servicio y así presentar a los usuarios de dicho cliente una base de datos personal virtual. Esto puede extenderse para cubrir las necesidades específicas del cliente usando las características de configuración discutidas anteriormente.

La escalabilidad es la capacidad del sistema de hacer frente a un número creciente de usuarios sin reducir la QoS global que se entrega a cualquier usuario. Por lo general, cuando se considera la escalabilidad en el contexto de SaaS, se toma en cuenta la "ampliación", en lugar de la "expansión". Recuerde que "ampliación" significa agregar servidores adicionales para así incrementar el número de transacciones que pueden procesarse en paralelo. La escalabilidad es un tema complejo que aquí no puede tratarse con detalle, pero algunos lineamientos generales para implementar software escalable son:

1. Desarrolle aplicaciones en las que cada componente se implemente como un simple servicio sin estado que pueda ejecutarse en cualquier servidor. Por consiguiente,

en el curso de una sola transacción, un usuario puede interactuar con instancias del mismo servicio que operan en varios servidores diferentes.

2. Diseñe el sistema usando interacción asíncrona, para que la aplicación no tenga que esperar el resultado de una interacción (como una solicitud de lectura). Esto permite que la aplicación realice trabajo útil mientras espera que termine la interacción.
3. Gestione los recursos, tales como conexiones de red y bases de datos, como un depósito para que ningún servidor individual tenga probabilidad de agotar sus recursos.
4. Diseñe su base de datos para permitir bloqueo de grano fino. Esto es, no bloquee registros completos en la base de datos cuando esté en uso sólo parte de un registro.

La noción de SaaS es un gran cambio de paradigma para computación distribuida. En lugar de que una organización aloje aplicaciones múltiples en sus servidores, SaaS permite que diferentes proveedores proporcionen externamente dichas aplicaciones. Se está en medio de una transición de un modelo a otro y, en el futuro, es probable que esto tenga un efecto muy significativo sobre la ingeniería de los sistemas de software empresariales.

PUNTOS CLAVE

- Los beneficios de los sistemas distribuidos son que pueden escalarse para hacer frente a la demanda creciente, pueden seguir proporcionando servicios de usuario (aun si fallan algunas partes del sistema) y permitir el almacenamiento de recursos.
- Los conflictos a considerar en el diseño de los sistemas distribuidos incluyen transparencia, apertura, escalabilidad, seguridad, calidad de servicio y gestión de fallas.
- Los sistemas cliente-servidor son sistemas distribuidos en los que el sistema se estructura en capas, con la capa de presentación implementada en una computadora cliente. Los servidores ofrecen servicios de gestión de datos, aplicación y de base de datos.
- Los sistemas cliente-servidor pueden tener varios niveles, con diferentes capas del sistema distribuidas a diferentes computadoras.
- Los patrones arquitectónicos para sistemas distribuidos incluyen arquitecturas maestro-esclavo, arquitecturas cliente-servidor de dos niveles y multinivel, arquitecturas de componentes distribuidos y arquitecturas entre pares (par a par o punto a punto).
- Los sistemas de componentes distribuidos requieren que middleware maneje las comunicaciones de los componentes y que permitan que se agreguen y eliminen componentes del sistema.
- Las arquitecturas entre pares son arquitecturas descentralizadas en las que no hay distinción entre clientes y servidores. La computación se puede distribuir entre muchos sistemas en diferentes organizaciones.
- El software como servicio es una forma de implementar aplicaciones como sistemas cliente-servidor ligero, en el que el cliente es un navegador Web.

LECTURAS SUGERIDAS

"Middleware: A model for distributed systems services". Aunque un poco arcaico en algunas partes, éste es un excelente ensayo panorámico que resume el papel del middleware en sistemas distribuidos y analiza el rango de servicios middleware que pueden proporcionarse. (P. A. Bernstein, *Comm. ACM*, **39** (2), febrero de 1996.) http://dx.doi.org/10.1145/230798.230809.

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Aun cuando este libro no tiene mucha información sobre arquitecturas p2p, es una excelente introducción a la computación p2p que explica la organización y el enfoque utilizados en algunos sistemas p2p. (A. Oram (ed.), O'Reilly and Associates Inc., 2001.)

"Turning software into a service". Un buen ensayo que analiza los principios de la computación orientada a servicios. A diferencia de muchos ensayos acerca de este tema, no oculta los principios detrás de una discusión de los estándares implicados. (M. Turner, D. Budgen y P. Brereton, *IEEE Computer*, **36** (10), octubre de 2003.) http://dx.doi.org/10.1109/MC.2003.1236470.

Distributed Systems: Principles and Paradigms, 2nd edition. Un libro de texto completo que describe todos los aspectos del diseño y la implementación de sistemas distribuidos. Sin embargo, no incluye una amplia discusión del paradigma orientado a servicios. (A.S. Tanenbaum y M. Van Steen, Addison-Wesley, 2007.)

"Software as a Service; The Spark that will Change Software Engineering". Un ensayo breve que argumenta que la llegada de SaaS llevará todo el desarrollo de software a un modelo iterativo. (G. Goth, Distributed Systems Online, **9** (7), julio de 2008.) http://dx.doi.org/10.1109/MDSO.2008.21.

EJERCICIOS

18.1. ¿Qué entiende por "escalabilidad"? Discuta las diferencias entre "expansión" (*scaling up*) y "ampliación" (*scaling out*) y explique cuándo pueden usarse estos diferentes enfoques a la escalabilidad.

18.2. Explique por qué los sistemas de software distribuidos son más complejos que los sistemas de software centralizados, donde toda la funcionalidad del sistema se implementa en una sola computadora.

18.3. Use un ejemplo de una solicitud de procedimiento remoto para explicar cómo el middleware coordina la interacción de las computadoras en un sistema distribuido.

18.4. ¿Cuál es la diferencia fundamental entre un enfoque de cliente pesado y uno de cliente ligero para las arquitecturas de sistemas cliente-servidor?

18.5. Al lector se le pide diseñar un sistema seguro que requiera autenticación y autorización. El sistema debe diseñarse de forma que las comunicaciones entre partes del sistema no puedan interceptarse ni leerse por un atacante. Sugiera la arquitectura cliente-servidor más adecuada para este sistema y, argumentando razones para su respuesta, proponga cómo debe distribuirse la funcionalidad del sistema entre el cliente y el servidor.

18.6. Su cliente quiere desarrollar un sistema para información de acciones donde los operadores puedan acceder a información de compañías y evaluar varios escenarios de inversión

mediante un sistema de simulación. Cada operador usa esta simulación de una forma diferente, de acuerdo con su experiencia y el tipo de acciones en cuestión. Sugiera una arquitectura cliente-servidor para este sistema que muestre dónde se ubica la funcionalidad. Justifique el modelo cliente-servidor que haya elegido.

18.7. Con un enfoque de componentes distribuidos, proponga una arquitectura para un sistema de boletaje nacional de teatros. Los usuarios pueden verificar la disponibilidad de asientos y reservarlos en un grupo de teatros. El sistema debe soportar devoluciones de boletos de manera que las personas puedan devolver en el último minuto sus boletos para reventa a otros clientes.

18.8. Indique dos ventajas y dos desventajas de las arquitecturas entre pares descentralizada y semicentralizada.

18.9. Explique por qué implementar software como servicio puede reducir los costos de soporte TI para una compañía. ¿Qué costos adicionales pueden surgir si se usa este modelo de implementación?

18.10. Su compañía quiere abandonar el uso de aplicaciones de escritorio para acceder a la misma funcionalidad de forma remota como servicio. Identifique tres riesgos que podrían surgir y sugiera cómo pueden reducirse tales riesgos.

REFERENCIAS

Bernstein, P. A. (1996). "Middleware: A Model for Distributed System Services". *Comm. ACM,* **39** (2), 86–97.

Coulouris, G., Dollimore, J. y Kindberg, T. (2005). *Distributed Systems: Concepts and Design, 4th edition.* Harlow, UK.: Addison-Wesley.

Holdener, A. T. (2008). *Ajax: The Definitive Guide.* Sebastopol, Calif.: O'Reilly and Associates.

McDougall, P. (2000). "The Power of Peer-To-Peer". *Information Week* (28 de agosto de 2000).

Neuman, B. C. (1994). "Scale in Distributed Systems". En *Readings in Distributed Computing Systems.* Casavant, T. y Singal, M. (ed.). Los Alamitos, Calif.: IEEE Computer Society Press.

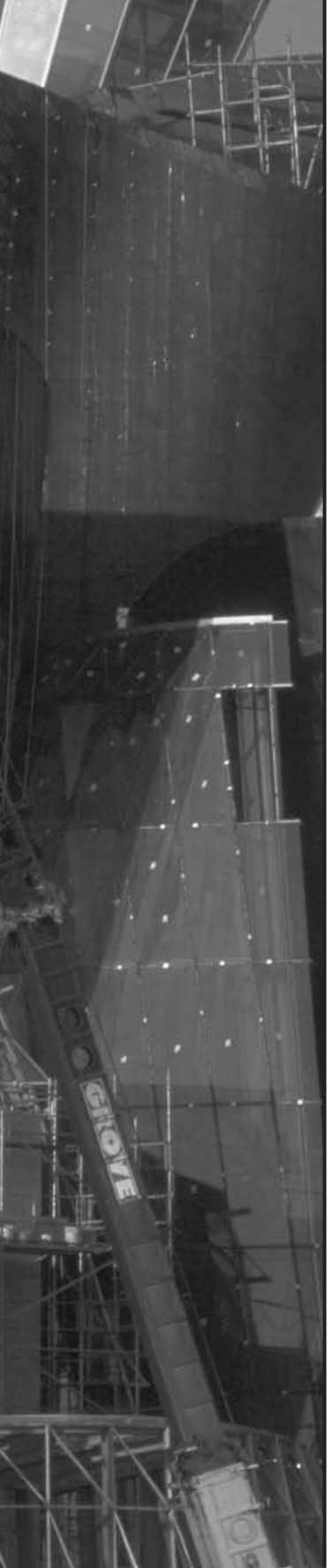
Oram, A. (2001). "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology".

Orfali, R. y Harkey, D. (1998). *Client/server Programming with Java and CORBA*. Nueva York: John Wiley & Sons.

Orfali, R., Harkey, D. y Edwards, J. (1997). *Instant CORBA*. Chichester, UK: John Wiley & Sons.

Pope, A. (1997). *The CORBA Reference Guide: Understanding the Common Request Broker Architecture*. Boston: Addison-Wesley.

Tanenbaum, A. S. y Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms, 2nd edition*. Upper Saddle River, NJ: Prentice Hall.



19

Arquitectura orientada a servicios

Objetivos

El objetivo de este capítulo es introducirlo a la arquitectura de software orientada a servicios como una forma de construir aplicaciones distribuidas mediante servicios Web. Al estudiar este capítulo:

- comprenderá las nociones básicas de un servicio Web, estándares de servicio Web y arquitectura orientada a servicios;
- conocerá el proceso de ingeniería de servicio cuya intención es producir servicios Web de reutilización;
- se introducirá al concepto de composición de servicios como un medio de desarrollo de aplicaciones orientadas a servicios;
- entenderá cómo pueden usarse los modelos de proceso empresarial como base para el diseño de sistemas orientados a servicios.

Contenido

19.1 Servicios como componentes de reutilización
19.2 Ingeniería de servicio
19.3 Desarrollo de software con servicios

El desarrollo de la Web en la década de 1990 revolucionó el intercambio de información organizacional. Las computadoras cliente podían obtener acceso a la información en servidores remotos fuera de sus organizaciones. Sin embargo, el acceso era exclusivamente a través de un navegador Web y no era práctico el acceso directo a la información por parte de otros programas. Esto significó que fueran imposibles las conexiones oportunistas entre servidores donde, por ejemplo, un programa consultaba algunos catálogos de diferentes proveedores.

Para solucionar este problema se propuso la noción de servicio Web. Al usar un servicio Web, las organizaciones que querían hacer accesible su información para otros programas podían lograrlo al definir y publicar una interfaz de servicio Web. Esta interfaz define los datos disponibles y cómo puede acceder a ellos. De manera más general, un servicio Web es una representación estándar para cierto recurso computacional o información que pueden usar otros programas. Éstos pueden ser recursos de información (como un catálogo de partes), recursos de computadora (como un procesador especializado), o recursos de almacenamiento. Por ejemplo, podría implementarse un servicio de archivo que almacenara de forma permanente y fiable datos de la organización que, por ley, deben conservarse durante muchos años.

Un servicio Web es una instancia de una noción más general de servicio, que se define (Lovelock *et al*., 1996) como:

> *un acto o una función ofrecidos por una parte a otra. Aunque el proceso puede asociarse a un proceso físico, la función es esencialmente intangible y, por lo general, no da por resultado la propiedad de alguno de los factores de producción.*

Por consiguiente, la particularidad de un servicio es que el hecho de proveer el servicio es independiente de la aplicación que usa el servicio (Turner *et al*., 2003). Los proveedores de servicio pueden desarrollar servicios especializados y ofrecer éstos a varios usuarios de servicio de diferentes organizaciones.

Las arquitecturas orientadas a servicios (SOA, por las siglas de *service-oriented architectures*) son una forma de desarrollar sistemas distribuidos en la que los componentes del sistema son servicios independientes y se ejecutan en computadoras distribuidas geográficamente. Los protocolos estándar basados en XML, tales como SOAP y WSDL, se diseñaron para dar soporte al servicio de comunicación e intercambio de información. Por consiguiente, los servicios son independientes de la plataforma y del lenguaje de implementación. Los sistemas de software pueden construirse al componer servicios locales y servicios externos de diferentes proveedores, con interacción uniforme entre los servicios del sistema.

La figura 19.1 resume la idea de una SOA. Los proveedores de servicio diseñan e implementan servicios y especifican las interfaces a estos últimos. También transmiten información acerca de dichos servicios en un registro accesible. Los solicitantes de servicio (llamados en ocasiones clientes) que quieren usar un servicio detectan la especificación de éste y ubican al proveedor del servicio. Entonces pueden unir su aplicación con dicho servicio específico y comunicarse con él, mediante protocolos de servicio estándar.

Desde el principio hay un proceso de estandarización activo para SOA, que trabaja junto a los desarrollos técnicos. Todas las grandes compañías de hardware y software se comprometen con dichos estándares. Como resultado, SOA no ha sufrido de las incom-

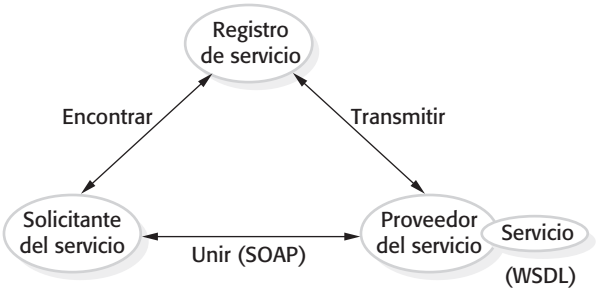


Figura 19.1 Arquitectura orientada a servicios

patibilidades que surgen comúnmente con las innovaciones técnicas, cuando diferentes proveedores mantienen su versión privada de la tecnología. La figura 19.2 muestra la pila de estándares clave que se establecieron para dar soporte a los servicios Web. Debido a este inicio de estandarización, los problemas discutidos en el capítulo 17, tales como los modelos de múltiples componentes incompatibles en CBSE, aún no surgen en el desarrollo de sistemas orientados a servicios.

Los protocolos de servicios Web cubren todos los aspectos de la SOA, desde los mecanismos básicos para el servicio de intercambio de información (SOAP) hasta estándares de lenguaje de programación (WS-BPEL). Dichos estándares se basan en XML, una notación humana y legible para una máquina que permite la definición de datos estructurados donde el texto se etiqueta con un identificador significativo. XML tiene una gama de tecnologías de apoyo, tal como XSD para definición de esquemas, que se usan para extender y manipular descripciones XML. Erl (2004) aporta un buen compendio sobre tecnologías XML y su función en los servicios Web.

Brevemente, los estándares clave de las SOA Web son los siguientes:

1. *SOAP* Éste es un estándar de intercambio de mensajes que soporta la comunicación entre servicios. Define el componente esencial y opcional de los mensajes transmitidos entre los servicios.
2. *WSDL* El Lenguaje de Definición de Servicio Web (WSDL, por las siglas de *Web Service Definition Language*) es un estándar para la definición de interfaz de servicio. Establece cómo deben definirse las operaciones de servicios (nombres de operación, parámetros y sus tipos) y los enlaces de servicio.
3. *WS-BPEL* Éste es un estándar para un lenguaje de flujo de trabajo que se usa para definir programas de proceso que implican varios servicios diferentes. En la sección 19.3 se explica la noción de programas de proceso.

También se propuso un estándar de descubrimiento de servicio, UDDI, pero esto no se ha adoptado ampliamente. El estándar UDDI (por las siglas de *Universal Description, Discovery and Integration*, es decir, descripción, descubrimiento e integración universales) define los componentes de una especificación de servicio, que puede usarse para descubrir la existencia de un servicio. Incluye información del proveedor, los servicios proporcionados, la ubicación de la descripción WSDL de la interfaz de servicio e información sobre las relaciones empresariales. La intención era que este estándar permitiría establecer a las compañías registros con descripciones UDDI que definen los servicios que éstas ofrecen.

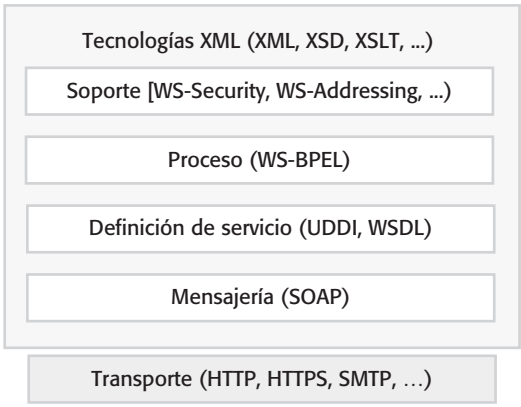


Figura 19.2 Estándares de servicio Web

Algunas compañías, como Microsoft, establecieron registros UDDI durante los primeros años del siglo XXI, aunque ahora todos están cerrados. Las mejoras en la tecnología de motores de búsqueda los hicieron redundantes. El descubrimiento de servicios usando un motor de búsqueda estándar para examinar descripciones WSDL comentadas de manera adecuada, es ahora el enfoque preferido en el descubrimiento de servicios externos.

Los principales estándares SOA están soportados por una variedad de estándares de apoyo que se enfocan en aspectos más especializados de SOA. Existe gran número de estándares de soporte porque tienen la intención de soportar SOA en diferentes tipos de aplicación empresarial. Algunos ejemplos de estos estándares incluyen los siguientes:

1. WS-Reliable Messaging (mensajería confiable WS), un estándar para el intercambio de mensajes que garantiza que los mensajes se entregarán una vez y sólo una vez.
2. WS-Security (seguridad WS), un conjunto de estándares que soportan la seguridad del servicio Web, incluyendo estándares que especifican la definición de políticas y estándares de seguridad que cubren el uso de firmas digitales.
3. WS-Addressing (direccionamiento WS), que define cómo debe representarse la información de dirección en un mensaje SOAP.
4. WS-Transactions (transacciones WS), que definen cómo se coordinan las transacciones a través de los servicios distribuidos.

Los estándares de servicio Web son un tema muy amplio y aquí no hay espacio para estudiarlos a detalle. Para conocer un panorama de dichos estándares, se recomienda el libro de Erl (2004; 2005). Sus descripciones detalladas están disponibles también como documentos públicos en la Web.

A los actuales estándares de servicios Web se les critica como estándares "pesados", ya que son muy generales e ineficientes. La implementación de tales estándares requiere de una considerable cantidad de procesamiento para crear, transmitir e interpretar los mensajes XML asociados. Por esta razón, algunas organizaciones, tales como Amazon, utilizan un enfoque más simple y eficiente para atender la comunicación utilizando los llamados servicios RESTful (Richardson y Ruby, 2007). El enfoque RESTful soporta interacción de servicio eficiente, pero no soporta características a nivel de empresa como

Servicios Web RESTful

REST (acrónimo de REpresentational State Transfer, es decir, transferencia de estado representacional) es un estilo arquitectónico basado en la transferencia de representaciones de recursos de un servidor a un cliente. Es el estilo que subyace en la Web como un todo y se ha usado como un método mucho más simple que SOAP/WSDL para implementar servicios Web.

Un servicio Web RESTful se identifica mediante su URI (identificador universal de recurso) y se comunica a través del protocolo HTML. Responde a métodos HTML como GET, PUT, POST y DELETE y regresa una representación de recurso al cliente. De manera sencilla, POST significa crear; GET, leer; PUT, actualizar y DELETE, borrar.

Los servicios RESTful implican una carga más baja que los llamados "grandes servicios Web" y los utilizan muchas organizaciones que implementan sistemas basados en servicios que no dependen de servicios ofrecidos de manera externa.

http://www.SoftwareEngineering-9.com/Web/Services/REST/

WS-Reliability (fiabilidad WS) y WS-Transactions. Pautasso y sus colaboradores (2008) comparan el enfoque RESTful con los servicios Web estandarizados.

Construir aplicaciones con base en servicios permite a las compañías y otras organizaciones cooperar y usar de manera mutua las funciones empresariales. Por lo tanto, los sistemas que implican un amplio intercambio de información a través de las fronteras de una compañía, tales como los sistemas de cadena de abastecimiento en los que una compañía solicita los bienes de otra, pueden automatizarse fácilmente. Las aplicaciones basadas en servicios pueden construirse al vincular los servicios de varios proveedores con el uso de un lenguaje de programación estándar o de un lenguaje de flujo de trabajo especializado, como se examina en la sección 19.3.

Las SOA son arquitecturas acopladas holgadas, en las que los enlaces de servicio pueden cambiar durante la ejecución. Esto significa que una versión distinta del servicio, pero equivalente, puede ejecutarse en diferentes momentos. Algunos sistemas se construirán exclusivamente utilizando servicios Web y otros combinarán servicios Web con componentes desarrollados localmente. Para ejemplificar cómo pueden organizarse las aplicaciones que usan una mezcla de servicios y componentes, considere el siguiente escenario:

> *Un sistema de información a bordo de un automóvil ofrece al conductor datos sobre el clima, las condiciones del tránsito, información local y así sucesivamente. El sistema se vincula al radio del automóvil, de modo que la información se entrega como una señal en un canal de radio específico. El automóvil está equipado con un receptor GPS para detectar su posición y, con base en dicha posición, el sistema accede a una gama de servicios de información. Entonces la información puede entregarse en el lenguaje específico del conductor.*

La figura 19.3 ilustra una posible organización de tal sistema. El software a bordo de un automóvil incluye cinco módulos. Éstos manejan las comunicaciones con el conductor, con un receptor GPS que reporta la posición del auto y con el radio del automóvil. Los módulos Transmisor y Receptor manejan todas las comunicaciones con los servicios externos.

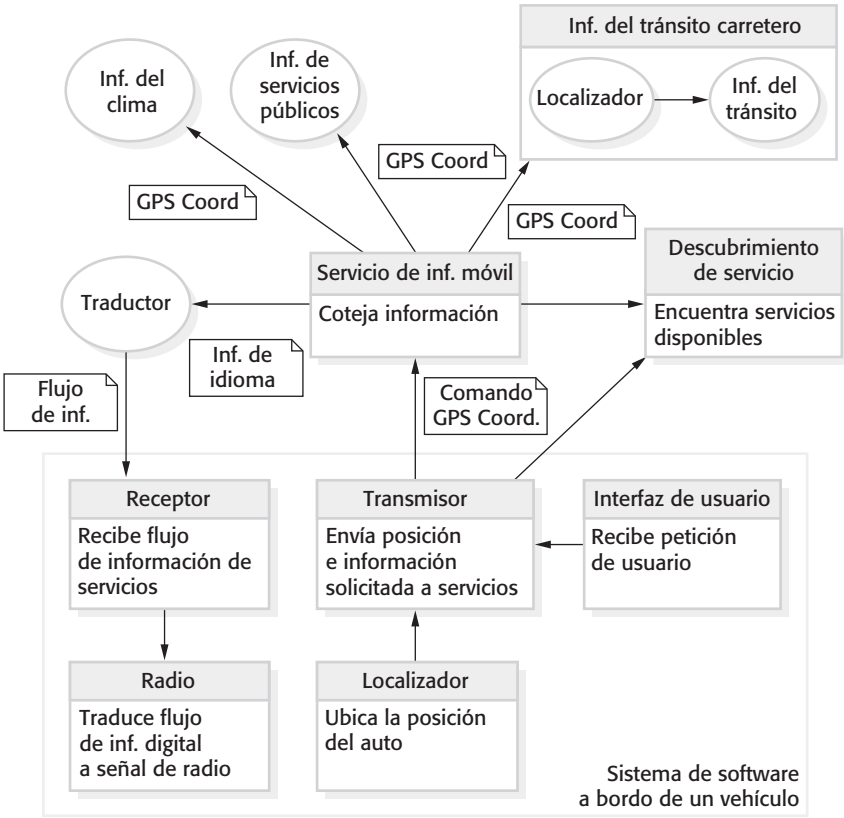


Figura 19.3 Sistema de información a bordo de un automóvil basado en servicios

El automóvil se comunica con un servicio de información móvil externo que agrega información de muchos otros servicios, y proporciona información acerca del clima, de las condiciones del tránsito y de servicios públicos locales. Diferentes proveedores en diversos lugares ofrecen tales servicios, y el sistema a bordo usa un servicio de descubrimiento para localizar los servicios de información adecuados y los enlaza. El servicio de información móvil también utiliza el servicio de descubrimiento para conectar a los servicios adecuados de clima, tránsito y servicios públicos. Los servicios intercambian mensajes SOAP que incluyen información de posición GPS usada por los servicios para seleccionar la información adecuada. Entonces la información añadida se envía al automóvil mediante un servicio que traduce dicha información al lenguaje preferido del conductor.

Este ejemplo ilustra una de las ventajas clave del enfoque orientado a servicios. No es necesario decidir cuándo se programa o despliega el sistema, qué proveedor de servicio se debe elegir o en qué servicios específicos se debe ingresar. Conforme el vehículo avanza, el software a bordo usa el servicio de descubrimiento de servicios para encontrar el servicio de información más adecuado y enlazarlo con él. Gracias al uso de un servicio de traducción, es posible cruzar fronteras y, por consiguiente, poner a disposición información local para las personas que no hablan el idioma.

Un enfoque orientado a servicios para la ingeniería de software es un nuevo paradigma en la ingeniería de software que, desde la perspectiva del autor, es un desarrollo

Ingeniería de software orientada a servicios y orientada a componentes

Evidentemente, los servicios y componentes tienen mucho en común. Ambos son elementos de reutilización y, como se estudió en el capítulo 17, es posible considerar un componente como un proveedor de servicios. Sin embargo, existen importantes diferencias entre servicios y componentes, y entre un enfoque a la ingeniería de software orientada a servicios y una orientada a componentes.

http://www.SoftwareEngineering-9.com/Web/Services/Comps.html

tan importante como la ingeniería de software orientada a objetos. Este cambio de paradigma se acelerará mediante el desarrollo de la "computación en nube" (Carr, 2009), en que los servicios se ofrecen en una infraestructura de computación utilitaria instalada por los grandes proveedores, como Google y Amazon. Esto ha tenido, y seguirá teniendo, profundos efectos sobre los productos de sistemas y los procesos empresariales. Newcomer y Lomow (2005), en su libro acerca de SOA, resumen el potencial de los enfoques orientados a servicios:

> *Impulsada por la convergencia de tecnologías clave y la adopción universal de servicios Web, la empresa orientada a servicios promete mejorar significativamente la agilidad corporativa, acelerar el tiempo de llegada al mercado para los nuevos productos y servicios, reducir los costos de TI y mejorar la eficiencia operativa.*

Aún se está en una etapa relativamente temprana en cuanto al desarrollo de aplicaciones orientadas a servicios a las que se accede a través de la Web. Sin embargo, ya se vislumbran grandes cambios en las formas en que se implementa y despliega el software, con el surgimiento de sistemas como Google Apps y Salesforce.com. Los enfoques orientados a servicios tanto al nivel de aplicaciones como al de implementación, significan que la Web evoluciona desde un almacén de información hacia una plataforma de implementación de sistemas.

19.1 Servicios como componentes de reutilización

En el capítulo 17 se introdujo la ingeniería de software basada en componentes (CBSE), en la que los sistemas de software se construyen al combinar componentes de software que se basan en un modelo de componentes estándar. Los servicios son un desarrollo natural de los componentes de software donde el modelo de componentes es, en esencia, un conjunto de estándares asociados con servicios Web. Por lo tanto, un servicio puede definirse como:

> *Un componente de software de reutilización, debidamente ajustado, que ofrece discreta funcionalidad, la cual puede distribuirse y a la que se accede de manera programática. Un servicio Web es un servicio al que se accede mediante protocolos estándar de Internet y basados en XML.*

Una distinción fundamental entre un servicio y un componente de software, como se define en CBSE, es que los servicios deben ser independientes y ajustarse debidamente; esto es, siempre deben operar en la misma forma, sin importar su entorno de ejecución. Sus interfaces son una interfaz "proporciona" que permite el acceso a la funcionalidad del servicio. Los servicios tienen la intención de ser independientes y utilizables en diferentes contextos. Por consiguiente, no tienen una interfaz "requiere" que, en CBSE, define a los otros componentes del sistema que deben estar presentes.

Los servicios se comunican mediante el intercambio de mensajes, expresados en XML, y dichos mensajes se distribuyen con protocolos estándar de transporte de Internet como HTTP y TCP/IP. En la sección 18.1.1 se estudió este enfoque basado en mensajes de la comunicación de componentes. Un servicio define lo que necesita de otro servicio al establecer sus requerimientos en un mensaje y al enviarlo a dicho servicio. El servicio receptor analiza los mensajes, realiza el cálculo y, al concluir, envía una respuesta, como mensaje, al servicio solicitante. Entonces este servicio examina la respuesta para extraer la información requerida. A diferencia de los componentes de software, los servicios no usan solicitudes de procedimiento o método remotos para acceder a la funcionalidad asociada con otros servicios.

Cuando usted trata de usar un servicio Web, debe saber dónde se ubica el servicio (su URI) y los detalles de su interfaz. Éstos se detallan en una descripción de servicio expresada en un lenguaje basado en XML llamado WSDL. La especificación WSDL define tres cosas acerca de un servicio Web: qué hace el servicio, cómo se comunica y dónde encontrarlo:

1. La parte "qué" de un documento WSDL, llamada interfaz, especifica qué operaciones soporta el servicio, y define el formato de los mensajes que se envían y reciben por parte del servicio.
2. La parte "cómo" de un documento WSDL, llamado enlace, mapea la interfaz abstracta a un conjunto concreto de protocolos. El enlace especifica los detalles técnicos de cómo comunicarse con un servicio Web.
3. La parte "dónde" de un documento WSDL describe la ubicación de una implementación de servicio Web específica (su punto final).

El modelo conceptual WSDL (figura 19.4) muestra los elementos de una descripción de servicio. Cada uno de ellos se expresa en XML y puede proporcionarse en archivos separados. Dichas partes son:

1. Una parte introductoria que define por lo general los espacios de nombre (namespaces) XML utilizados y cuáles puede incluir una sección de documentación que brinde información adicional sobre el servicio.
2. Una descripción opcional de los tipos que se usan en los mensajes intercambiados por el servicio.
3. Una descripción de la interfaz de servicio; esto es, las operaciones que ofrece el servicio para otros servicios o usuarios.
4. Una descripción de los mensajes de entrada y salida procesados por el servicio.

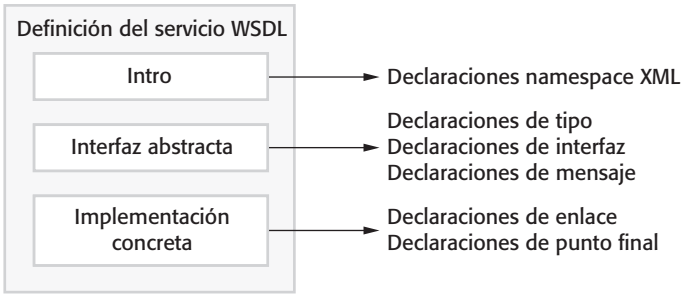


Figura 19.4 Organización de una especificación WSDL

5. Una descripción de los enlaces usados por el servicio (es decir, el protocolo de mensajería que utilizará para enviar y recibir mensajes). Por defecto, es SOAP, pero también pueden especificarse otros enlaces. El enlace establece cómo los mensajes de entrada y salida asociados con el servicio deben empaquetarse en un mensaje, y especifica los protocolos de comunicación utilizados. El enlace también puede especificar cómo se incluye soporte a la información, como credenciales de seguridad o identificadores de transacción.

6. Una especificación de punto final, que es la ubicación física del servicio, expresada como un Identificador Universal de Recursos (URI): la dirección de un recurso al que se puede acceder a través de Internet.

Las descripciones completas de servicio, escritas en XML, son largas, detalladas y tediosas de leer. Por lo general, incluyen definiciones de namespaces XML, que son calificadores de nombres. Un identificador namespace puede preceder a cualquier identificador usado en la descripción XML, lo que posibilita distinguir entre identificadores con el mismo nombre definidos en diferentes partes de una descripción XML. Aquí no es necesario comprender los detalles de los namespaces para entender los ejemplos. Basta con saber que dichos nombres pueden prefijarse con un identificador namespace y que el par namespace:nombre debe ser único.

Las especificaciones WSDL ahora se escriben pocas veces a mano y la mayor parte de la información en una especificación puede generarse automáticamente. No es necesario conocer los detalles de una especificación para comprender los principios de WSDL, de manera que aquí el enfoque está sobre la descripción de la interfaz abstracta. Ésta es la parte de una especificación WSDL que es igual a la interfaz "proporcionar" de un componente de software. La figura 19.5 muestra parte de la interfaz para un servicio simple que, a partir de una fecha y un lugar, especificado como una ciudad dentro de un país, indica la temperatura máxima y mínima registrada en tal lugar y en esa fecha. El mensaje de entrada especifica también si dichas temperaturas se desplegarán en grados Celsius o en grados Fahrenheit.

En la figura 19.5, la primera parte de la descripción muestra un aspecto del elemento y la definición tipo que se usa en la especificación de servicio. Ésta define los elementos PlaceAndDate (lugar y fecha), MaxMinTemp (temperaturas máxima y mínima) e InDataFault (falla de datos de entrada). Sólo se incluyó la especificación de PlaceAndDate, que se puede considerar como un registro con tres campos: ciudad, país y fecha. Un enfoque similar se usaría para definir MaxMinTemp e InDataFault.

```
Define algunos de los tipos usados. Supone que el prefijo namespace "ws"
se refiere al namespace URI para esquemas XML y el prefijo namespace
asociado con esta definición es weathns

<types>
   <xs: schema targetNameSpace = "http://.../weathns"
        xmlns: weathns = "http://. . ./weathns" >
        <xs:element name = "PlaceAndDate" type = "pdrec" />
        <xs:element name = "MaxMinTemp" type = "mmtrec" />
        <xs:element name = "InDataFault" type = "errmess" />

        <xs:complexType name = "pdrec"
        <xs:sequence>
            <xs:element name = "town" type = "xs:string"/>
            <xs:element name = "country" type = "xs:string"/>
            <xs:element name = "day" type = "xs:date" />
        </xs:complexType>

        Definiciones aquí de MaxMinType e InDataFault

   </schema>

</types>

Ahora defina la interfaz y sus operaciones. En este caso, sólo hay una
sola operación para regresar temperatura máxima y mínima.

<interface name = "weatherInfo" >
   <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
   <input messageLabel = "In" element = "weathns: PlaceAndDate" />
   <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
   <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

Figura 19.5 Parte de una descripción WSDL para un servicio Web

La segunda parte de la descripción indica cómo se define la interfaz de servicio. En este ejemplo, el servicio weatherInfo tiene una sola operación, aunque no hay restricciones sobre el número de operaciones que pueden definirse. La operación weatherInfo tiene un patrón asociado in-out que significa que toma un mensaje de entrada y genera un mensaje de salida. La especificación WSDL 2.0 permite que algunos mensajes diferentes intercambien patrones, como in-only (sólo entrada), in-out (entrada-salida), out-only (sólo salida), in-optional-out (entrada-opcional-salida), out-in (salida-entrada), etcétera. Entonces se definen los mensajes de entrada y salida, que se refieren a las definiciones hechas anteriormente en la sección tipos.

El principal problema con WSDL es que la definición de la interfaz de servicio no incluye información alguna acerca de la semántica del servicio o sus características no funcionales, como rendimiento y confiabilidad. Es simplemente una descripción de la firma del servicio (es decir, las operaciones y sus parámetros). El programador que planea usar

el servicio debe calcular qué hace realmente el servicio y qué significan los diferentes campos en los mensajes de entrada y salida. El rendimiento y la confiabilidad tienen que descubrirse mediante la experimentación con el servicio. Nombres significativos y documentación ayudan a comprender la funcionalidad que se ofrece, aun cuando todavía es posible que los lectores malinterpreten el servicio.

19.2 Ingeniería de servicio

La ingeniería de servicio es el proceso de desarrollo de servicios para reutilización en aplicaciones orientadas a servicios. Tiene mucho en común con la ingeniería de componentes. Los ingenieros de servicio deben garantizar que el servicio represente una abstracción de reutilización que podría ser útil en diferentes sistemas. Deben diseñar y desarrollar funcionalidad generalmente útil asociada con dicha abstracción, y garantizar que el servicio es robusto y fiable. Deben documentar el servicio de modo que puedan descubrir y comprender los usuarios potenciales.

Existen tres etapas lógicas en el proceso de ingeniería de servicio, como se muestra en la figura 19.6. Se trata de las siguientes:

1. Identificación de candidatos a servicio, donde se identifican los posibles servicios que se podrían implementar y se definen los requerimientos del servicio.
2. Diseño del servicio, donde se diseñan las interfaces lógica y de servicio WSDL.
3. Implementación y despliegue del servicio, donde el servicio se implementa, se prueba y se pone a disposición del usuario.

Como se estudió en el capítulo 16, el desarrollo de un componente de reutilización puede comenzar con un componente existente que ya se haya implementado y usado en una aplicación. Lo mismo es verdadero para los servicios: el punto de partida de este proceso será con frecuencia un servicio existente o un componente que se convertirá a servicio. En esta situación, el proceso de diseño implica generalizar el componente existente de manera que se eliminen características específicas de la aplicación. Implementación significa adaptar el componente al agregar interfaces de servicio e implementar las generalizaciones requeridas.

19.2.1 Identificación de candidatos a servicio

La noción básica de computación orientada a servicios es que tales servicios deben soportar procesos empresariales. Puesto que toda organización tiene una gran variedad de procesos, existen muchos posibles servicios que pueden implementarse. Por lo tanto, la identificación de los candidatos a servicio implica comprender y analizar los procesos empresariales de la organización para decidir cuáles servicios de reutilización podrían implementarse para soportar dichos procesos.

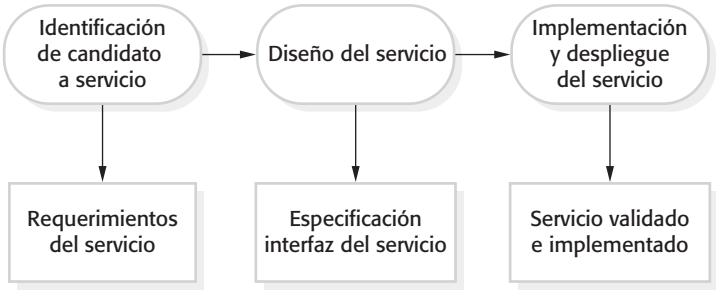


Figura 19.6 El proceso de ingeniería de servicio

Erl señala que existen tres tipos fundamentales de servicios que pueden identificarse:

1. *Servicios utilitarios* Se trata de servicios que implementan alguna funcionalidad general que pueden usar diferentes procesos empresariales. Un ejemplo de servicio utilitario es un servicio de conversión de divisas al que puede accederse para calcular la conversión de una divisa (por ejemplo, dólares) a otra (por ejemplo, euros).
2. *Servicios empresariales* Se trata de servicios asociados con una función empresarial específica. Un ejemplo de una función empresarial en una universidad sería la inscripción de estudiantes para un curso.
3. *Servicios de coordinación o proceso* Se trata de servicios que soportan un proceso empresarial más general que por lo general implican diferentes actores y actividades. Un ejemplo de un servicio de coordinación en una compañía es un servicio de pedidos que permite la colocación de pedidos con proveedores, bienes aceptados y pagos realizados.

Erl también plantea que los servicios pueden considerarse como orientados a tareas u orientados a entidades. Los servicios orientados a tareas son aquellos asociados con alguna actividad, mientras que los servicios orientados a entidades son como objetos. Se asocian con una entidad empresarial, como, por ejemplo, un formato de solicitud de empleo. La figura 19.7 muestra algunos ejemplos de servicios orientados a tareas o a entidades. Los servicios utilitarios o empresariales pueden orientarse a entidades o a tareas, pero los servicios de coordinación siempre son orientados a tareas.

Su meta en la identificación de candidatos a servicio debe ser identificar los servicios que sean lógicamente coherentes, independientes y de reutilización. La clasificación de Erl es útil en este aspecto, pues sugiere cómo descubrir servicios de reutilización al observar entidades y actividades empresariales. Sin embargo, identificar a los candidatos a servicio en ocasiones es difícil, pues se debe vislumbrar cómo se usarán los servicios. Se debe pensar en los posibles candidatos y luego plantear una serie de preguntas acerca de ellos para ver si es probable que los servicios sean útiles. Las posibles preguntas que se pueden plantear para identificar servicios de reutilización potencial son:

1. Para un servicio orientado a entidades, ¿el servicio está asociado con una sola entidad lógica que se usa en diferentes procesos empresariales? ¿Qué operaciones que deban soportarse se realizan usualmente sobre dicha entidad?

	Utilitaria	Empresarial	Coordinación
Tarea	Convertidor de divisas Localizador de empleo	Formato de validación de reclamo Comprobación de calificación crediticia	Proceso de reclamo de gastos Pago a proveedor externo
Entidad	Verificador de estilo de documento Convertidor de formato Web a XML	Formato de gastos Formato de solicitud estudiantil	

Figura 19.7 Clasificación de servicios

2. Para un servicio orientado a tareas, ¿se trata de una tarea que realizan diferentes personas en la organización? ¿Querrán aceptar la inevitable estandarización que ocurrirá cuando se brinde un solo servicio de soporte?
3. ¿El servicio es independiente? (Es decir, ¿en qué medida depende de la disponibilidad de otros servicios?).
4. Para su operación, ¿el servicio debe mantener estado? Los servicios no tienen estado, lo que significa que no mantienen estado interno. Si se requiere información de estado, debe usarse una base de datos y esto puede limitar la reutilización del sistema. En general, los servicios en que el estado pasa al servicio son más fáciles de reutilizar, pues no se requiere enlace a base de datos.
5. ¿El servicio podrían usarlo clientes fuera de la organización? Por ejemplo, tanto usuarios internos como externos podrían ingresar a un servicio orientado a entidades asociado con un catálogo.
6. ¿Es probable que diferentes usuarios del servicio tengan distintos requerimientos no funcionales? Si los tienen, entonces esto sugiere que quizá deba implementarse más de una versión de un servicio.

Las respuestas a tales preguntas ayudan a seleccionar y refinar abstracciones que pudieran implementarse como servicio. Sin embargo, no hay fórmulas mágicas para decidir cuáles son los mejores servicios y, por lo tanto, la identificación de servicios es un proceso basado en la habilidad y la experiencia.

El resultado del proceso de selección de servicios es un conjunto de servicios identificados y requerimientos asociados para dichos servicios. Los requerimientos funcionales del servicio deben definir qué debe hacer el servicio. Los requerimientos no funcionales deben definir los requerimientos de seguridad, rendimiento y disponibilidad del servicio.

Para ayudar a entender el proceso de identificación e implementación de candidatos a servicio, considere el siguiente ejemplo:

Una compañía grande, que vende equipo de cómputo, dispuso precios especiales para configuraciones aprobadas de algunos clientes. Para facilitar la automatización de los pedidos, la compañía desea producir un servicio de catálogo que permita a los clientes seleccionar el equipo que necesitan. A diferencia de un catálogo de consumidor, los pedidos no se realizan directamente a través de una interfaz de catalogo. En vez de ello, los bienes se solicitan mediante un sistema de procuración de cada compañía basado en la Web, que accede al catálogo como un servicio Web. La mayoría de las compañías tienen sus propios procedimientos de

presupuesto y aprobación para los pedidos, y deben seguir sus propios procesos cuando se realice un pedido.

El servicio de catálogo es un ejemplo de un servicio orientado a entidades que soporta operaciones empresariales. Los siguientes son los requerimientos funcionales del servicio de catálogo:

1. Cada compañía usuaria debe contar con una versión específica del catálogo. Éste debe incluir las configuraciones y los equipos que pueden solicitar los empleados de la compañía cliente y los precios acordados para los artículos del catálogo.
2. El catálogo debe permitir a un empleado cliente descargar una versión del catálogo para consulta fuera de línea.
3. El catálogo debe permitir a los usuarios comparar las especificaciones y los precios de hasta seis artículos del catálogo.
4. El catálogo debe proporcionar instalaciones de navegación y búsqueda para los usuarios.
5. Los usuarios del catálogo podrán descubrir la fecha de entrega prevista para un número dado de artículos específicos del catálogo.
6. Los usuarios del catálogo podrán colocar "pedidos virtuales" donde los artículos requeridos se guardarán durante 48 horas. Los pedidos virtuales deberán confirmarse mediante un pedido real efectuado a través de un sistema de procuración. Éste debe recibirse dentro de 48 horas después del pedido virtual.

Además de estos requerimientos funcionales, el catálogo tiene algunos requerimientos no funcionales:

1. El acceso al servicio de catálogo está restringido a empleados de organizaciones acreditadas.
2. Los precios y las configuraciones ofrecidos a un cliente deben ser confidenciales y no estarán disponibles para los empleados de algún otro cliente.
3. El catálogo estará disponible sin interrupción de servicio de 0700 GMT a 1100 GMT.
4. El servicio de catálogo podrá procesar hasta 10 solicitudes por segundo en carga pico.

Observe que no hay requerimientos no funcionales relacionados con el tiempo de respuesta del servicio del catálogo. Esto depende del tamaño del catálogo y del número esperado de usuarios simultáneos. Puesto que éste no es un servicio crítico en términos de tiempo, no hay necesidad de especificarlo en esta etapa.

19.2.2 Diseño de interfaces del servicio

Una vez seleccionados los servicios candidatos, la siguiente etapa en el proceso de ingeniería de servicios es diseñar las interfaces del servicio. Esto implica definir las operaciones asociadas con el servicio y sus parámetros. También se debe considerar cuidadosamente

Operación	Descripción
MakeCatalog	Crea una versión del catálogo ajustada a un cliente específico. Incluye un parámetro opcional para crear una versión PDF que se descargue del catálogo.
Compare	Proporciona una comparación de hasta seis características (por ejemplo, precio, dimensiones, rapidez de procesamiento, etc.) de hasta cuatro artículos del catálogo.
Lookup	Despliega todos los datos asociados con un artículo específico del catálogo.
Search	Esta operación toma una expresión lógica y busca en el catálogo de acuerdo con dicha expresión. Muestra una lista de todos los artículos que coinciden con la expresión de búsqueda.
CheckDelivery	Indica la fecha de entrega prevista para un artículo si se solicita ese día.
MakeVirtualOrder	Reserva el número de artículos que puede solicitar un cliente y brinda información del artículo para el sistema de procuración del cliente.

Figura 19.8 Descripciones funcionales de las operaciones del servicio de catálogo

el diseño de las operaciones y los mensajes del servicio. Su meta consiste en minimizar el número de intercambios de mensajes que deben tener lugar para completar la petición del servicio. Usted tiene que garantizar que tanta información como sea posible pase al servicio en un mensaje en lugar de usar interacciones de servicio sincrónicas.

También debe recordar que los servicios no tienen estado y que la gestión del estado de la aplicación específica del servicio es responsabilidad del usuario del servicio, y no del servicio en sí. Por lo tanto, tendrá que transmitir esta información del estado hacia y desde los servicios en mensajes de entrada y salida.

Existen tres etapas en el diseño de la interfaz del servicio:

1. Diseño de interfaz lógica, donde se identifican las operaciones asociadas con el servicio, sus entradas y salidas, así como las excepciones asociadas con dichas operaciones.
2. Diseño de mensajes, donde se diseña la estructura de los mensajes que envía y recibe el servicio.
3. Desarrollo WSDL, donde el diseño lógico y de mensajes se traducen a una descripción de interfaz abstracta escrita en WSDL.

La primera etapa, el diseño de interfaz lógica, comienza con los requerimientos del servicio y define los nombres y parámetros de operación. En esta etapa también se deben definir las excepciones que podrían surgir cuando se invoca una operación de servicio. Las figuras 19.8 y 19.9 muestran las operaciones que implementan los requerimientos, así como las entradas, salidas y excepciones para cada una de las operaciones del catálogo. En esta etapa no hay necesidad de que las especificaciones sean minuciosas; los detalles se agregan en la siguiente etapa del proceso de diseño.

La definición de excepciones y cómo éstas se comunican a los usuarios del servicio son particularmente importantes. Los ingenieros de servicio no saben cómo se usarán sus

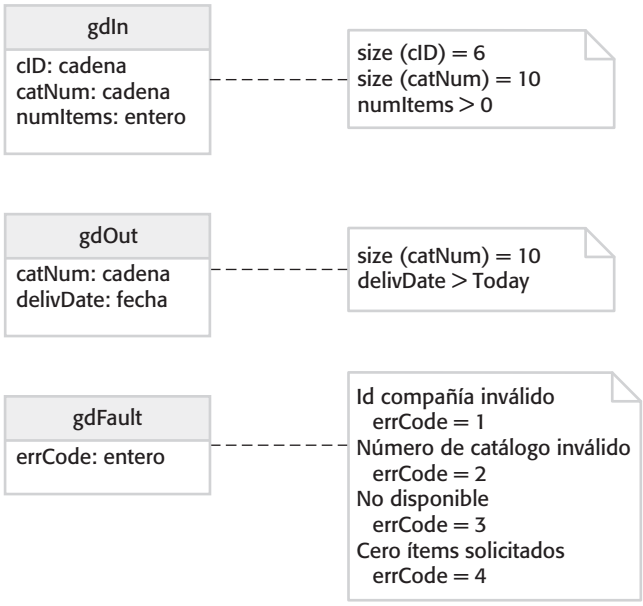


Figura 19.9 Diseño de interfaz de catálogo

servicios. Por lo general, no es aconsejable hacer conjeturas de que los usuarios del servicio tendrán comprensión completa de la especificación de éste. Los mensajes de entrada pueden ser incorrectos, de modo que habrá que definir excepciones que reporten las entradas incorrectas al cliente del servicio. En el desarrollo de componentes de reutilización, comúnmente es una buena práctica dejar todo el manejo de excepciones al usuario del componente. El desarrollador del servicio no debe imponer su visión acerca de cómo deben manejarse las excepciones.

Una vez establecida una descripción lógica informal de lo que debe hacer el servicio, la siguiente etapa es definir la estructura de los mensajes de entrada y salida, así como los tipos utilizados en dichos mensajes. XML es una notación inconveniente para usar en esta etapa. Es mejor representar los mensajes como objetos y definirlos usando el UML o un lenguaje de programación, como Java. Entonces pueden convertirse manual o automáticamente a XML. La figura 19.10 muestra la estructura de los mensajes de entrada y salida para la operación getDelivery en el servicio de catálogo.

Observe cómo se agregaron detalles a la descripción al anotar el diagrama UML con restricciones. Éstas definen la longitud de las cadenas que representan a la compañía y al artículo de catálogo, y especifican que el número de artículos debe ser mayor que cero y que la entrega debe ser posterior a la fecha actual. Las notaciones también muestran cuáles códigos de error se asocian con cada posible falla.

La etapa final del proceso de diseño de servicio es traducir el diseño de interfaz del servicio a WSDL. Como se discutió en la sección previa, una representación WSDL es amplia y detallada, y por lo tanto si se hace manualmente es fácil cometer errores en esta etapa. Sin embargo, la mayoría de los entornos de programación que soportan el desarrollo orientado a servicios (por ejemplo, el entorno ECLIPSE) incluyen herramientas que pueden traducir una descripción de interfaz lógica en su correspondiente representación WSDL.

Operación	Entradas	Salidas	Excepciones
MakeCatalog	*mcIn* Id de compañía Bandera PDF	*mcOut* URL del catálogo para dicha compañía	*mcFault* Id de compañía inválido
Compare	*compIn* Id de compañía Atributo de entrada (hasta 6) Número de catálogo (hasta 4)	*compOut* URL de página que muestra la tabla de comparación	*compFault* Id de compañía inválido Número de catálogo inválido Atributo desconocido
Lookup	*lookIn* Id de compañía Número de catálogo	*lookOut* URL de página con información del artículo	*lookFault* Id de compañía inválido Número de catálogo inválido
Search	*searchIn* Id de compañía Cadena de búsqueda	*searchOut* URL de página Web con resultados de búsqueda	*searchFault* Id de compañía inválido Cadena de búsqueda mal formada
CheckDelivery	*gdIn* Id de compañía Número de catálogo Número de artículos solicitados	*gdOut* Número de catálogo Fecha entrega esperada	*gdFault* Id de compañía inválido Número de catálogo inválido No disponible Cero artículos solicitados
PlaceOrder	*poIn* Id de compañía Número de artículos solicitados Número de catálogo	*poOut* Número de catálogo Número de artículos solicitados Fecha de entrega predicha Precio unitario estimado Precio total estimado	*poFault* Id de compañía inválido Número de catálogo inválido Cero artículos solicitados

Figura 19.10 Definición UML de los mensajes de entrada y salida

19.2.3 Implementación y despliegue del servicio

Una vez identificados los servicios candidatos y diseñadas sus interfaces, la etapa final del proceso de ingeniería de servicios es la implementación del servicio. Esta implementación puede implicar la programación del servicio usando un lenguaje de programación estándar como Java o C#. Ambos lenguajes incluyen librerías con extenso soporte para desarrollo del servicio.

Alternativamente, los servicios pueden desarrollarse al implementar las interfaces del servicio a componentes existentes o, como se discutió anteriormente, a sistemas heredados. Esto significa que los activos de software que ya probaron ser útiles pueden hacerse más disponibles. En el caso de sistemas heredados, tal vez signifique que es posible acceder a la funcionalidad del sistema mediante nuevas aplicaciones. También se pueden desarrollar nuevos servicios al definir composiciones de los servicios existentes. Este enfoque al desarrollo de servicios se analiza en la sección 19.3.

Una vez implementado un servicio, tiene que probarse antes de desplegarse. Esto supone la exploración y partición de las entradas del servicio (como se explicó en el

capítulo 8), y crear mensajes de entrada que reflejen dichas combinaciones de entrada, y entonces comprobar que las salidas son las esperadas. Siempre se debe tratar de generar excepciones durante la prueba para comprobar que el servicio puede hacer frente a entradas inválidas. Están disponibles herramientas de prueba que permiten examinar y verificar los servicios, y que generan pruebas a partir de especificación WSDL. Sin embargo, éstas sólo pueden probar la conformidad de la interfaz del servicio con el WSDL. No pueden poner a prueba el comportamiento funcional del servicio.

El despliegue del servicio, la etapa final del proceso, implica poner a disposición el servicio para su uso en un servidor Web. La mayoría del software servidor hace esto muy simple. Sólo hay que instalar el archivo que contiene el servicio ejecutable en un directorio específico. Entonces, automáticamente queda disponible para su uso. Si el servicio tiene la intención de estar a disposición pública, se debe proporcionar en tal caso información para usuarios externos del servicio. Esta información ayuda a los usuarios potenciales externos a decidir si es probable que el servicio cubra sus necesidades y si pueden confiar en usted, como proveedor del servicio, para entregar este último de manera fiable y con seguridad. La información que es posible incluir en una descripción del servicio puede ser la siguiente:

1. Información acerca de su empresa, detalles de contacto, etcétera. Esto es importante por razones de confianza. Los usuarios de un servicio deben estar seguros de que no se comportará maliciosamente. La información acerca del proveedor del servicio les permite comprobar sus acreditaciones con agencias de información empresarial.
2. Una descripción informal de la funcionalidad proporcionada por el servicio. Esto ayuda a los usuarios potenciales a decidir si el servicio es lo que esperan. Sin embargo, la descripción funcional está en lenguaje natural, de modo que no es una descripción semántica sin ambigüedades sobre lo que hace el servicio.
3. Una descripción detallada de los tipos de interfaz y de semántica.
4. Información de suscripción que permite a los usuarios registrarse mediante información sobre actualizaciones al servicio.

Como se expuso, un problema general con las especificaciones del servicio es que el comportamiento funcional del servicio se especifica de manera regular e informal, como una descripción en lenguaje natural. Las descripciones en lenguaje natural son fáciles de leer, pero están sujetas a malas interpretaciones. Para enfrentar este problema, existe una activa comunidad de investigación preocupada por indagar cómo puede especificarse la semántica de los servicios. El enfoque más prometedor a la especificación semántica se basa en una descripción sustentada en ontología, en la que el significado específico de los términos en una descripción se define en una ontología. Las ontologías son una manera de estandarizar las formas en que se usa dicha terminología y definen las relaciones entre diferentes términos. Se emplean cada vez más para ayudar a asignar semántica a descripciones en lenguaje natural. Un lenguaje llamado OWL-S se desarrolló para describir ontologías de servicio Web (OWL_Services_Coalition, 2003).

19.2.4 Servicios de sistemas heredados

Los sistemas heredados son sistemas de software antiguos que emplea una organización. Por lo general, dependen de tecnología obsoleta, pero todavía son esenciales para

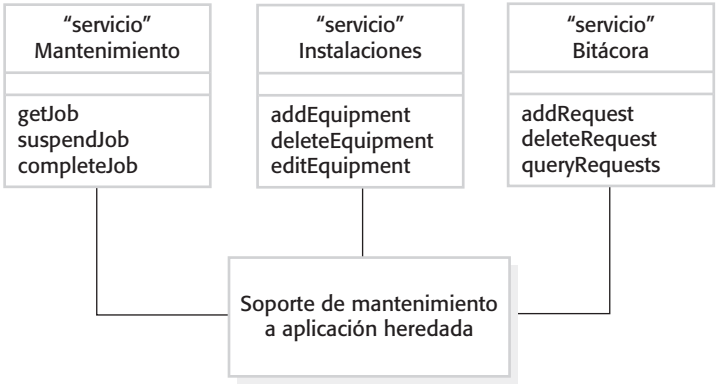


Figura 19.11 Servicios que proporcionan acceso a un sistema heredado

la empresa. Tal vez no sea efectivo en términos de costo reescribir o sustituir dichos sistemas, y muchas organizaciones quisieran usarlos en conjunción con sistemas más modernos. Uno de los usos más importantes de los servicios es implementar envolturas (*wrappers*) para sistemas heredados que brinden acceso a las funciones y datos de un sistema. Entonces se puede acceder a dichos sistemas a través de la Web e integrarlos con otras aplicaciones.

Para ilustrar esto, imagine que una compañía grande mantiene un inventario de su equipo y una base de datos asociada que sigue la huella del mantenimiento y las reparaciones del equipo. De esta forma, se da seguimiento a qué peticiones de mantenimiento se han realizado para diferentes piezas de equipo, qué mantenimiento regular está programado, cuándo se realizó el servicio de mantenimiento, cuánto tiempo se empleó en éste, etcétera. Este sistema heredado se usó originalmente para generar listas de trabajo diarias para el personal de mantenimiento, pero, con el tiempo, se agregaron nuevas instalaciones. Éstas proporcionan datos acerca de cuánto se ha gastado en el mantenimiento de cada pieza de equipo e información para ayudar a costear el trabajo de mantenimiento a realizar por contratistas externos. El sistema funciona como un sistema cliente-servidor con software cliente de propósito especial que se ejecuta en una PC.

Ahora la compañía quiere ofrecer acceso en tiempo real a este sistema desde terminales portátiles utilizadas por el personal de mantenimiento. Ellos actualizarán el sistema directamente con el tiempo y los recursos empleados en el mantenimiento, y consultarán el sistema para encontrar su siguiente labor de mantenimiento. Además, el personal del centro telefónico requiere acceso al sistema para registrar las peticiones de mantenimiento y verificar su estatus.

Es prácticamente imposible mejorar el sistema para soportar dichos requerimientos, de manera que la compañía decide ofrecer nuevas aplicaciones para el personal de mantenimiento y del centro telefónico. Dichas aplicaciones se apoyan en los sistemas heredados, que se usan como base para implementar algunos servicios. Esto se ilustra en la figura 19.11, donde se usó un estereotipo UML para indicar un servicio. Las aplicaciones nuevas intercambian mensajes con dichos servicios para acceder a la funcionalidad del sistema heredado.

Algunos de los servicios ofrecidos son los siguientes:

1. *Un servicio de mantenimiento* Esto incluye operaciones para recuperar una labor de mantenimiento de acuerdo con su número de trabajo, prioridad y ubicación geográfica, y para subir a la base de datos detalles del mantenimiento que se realizó.

El servicio también ofrece operaciones que permiten suspender o reiniciar una labor de mantenimiento que se inició, pero que está incompleta.

2. *Un servicio de instalaciones* Incluye operaciones para agregar y borrar nuevo equipo y modificar la información asociada con el equipo en la base de datos.

3. *Un servicio de bitácora* Implica operaciones para agregar una nueva petición de servicio, borrar peticiones de mantenimiento y consultar el estatus de peticiones atrasadas.

Observe que el sistema heredado existente no se representa simplemente como un solo servicio. En vez de ello, los servicios que se desarrollan para acceder al sistema heredado son coherentes y soportan una sola área de funcionalidad. Esto reduce su complejidad y los hace más fáciles de entender y reutilizar en otras aplicaciones.

19.3 Desarrollo de software con servicios

El desarrollo de software utilizando servicios se basa en la idea de que usted combina y configura servicios para crear nuevos servicios compuestos. Éstos pueden integrarse con una interfaz de usuario implementada en un navegador para crear una aplicación Web, o pueden usarse como componentes en algún otro servicio de composición. Los servicios implicados en la composición pueden desarrollarse especialmente para la aplicación, pueden ser servicios empresariales desarrollados dentro de una compañía o pueden ser servicios de un proveedor externo.

Muchas compañías ahora convierten sus aplicaciones empresariales en sistemas orientados a servicios, donde el bloque constructor básico de la aplicación es un servicio en vez de un componente. Esto abre la posibilidad de reutilización más difundida dentro de la compañía. La siguiente etapa será el desarrollo de aplicaciones interorganizacionales entre proveedores confiables, quienes intercambiarán servicios. La realización final de la visión a largo plazo de la SOA dependerá del desarrollo de un "mercado de servicios", donde los servicios se compran a proveedores externos.

La composición de servicios puede usarse en la elaboración de procesos empresariales separados para dar un proceso integrado que ofrezca funcionalidad más extensa. Suponga que una aerolínea quiere ofrecer un paquete vacacional completo para los viajeros. Además de reservar sus vuelos, los viajeros también pueden reservar hoteles en su destino preferido, ordenar renta de automóviles o reservar un taxi desde el aeropuerto, navegar en una guía de viaje y hacer reservaciones para visitar lugares atractivos. Para crear esta aplicación, la aerolínea combina su servicio de reservaciones con los servicios que ofrece una agencia de reservaciones hoteleras, agencias de alquiler de autos y compañías de taxis, y con los servicios de reservación que ofrecen los propietarios de los lugares de interés. El resultado final es un solo servicio que integra los servicios de diferentes proveedores.

Este proceso se puede considerar como una secuencia de pasos separados como se muestra en la figura 19.12. La información se transmite de un paso al siguiente: por ejemplo, la compañía de alquiler de autos está informada de la hora en que llegará el vuelo. La secuencia de pasos se llama flujo de trabajo (*workflow*): un conjunto de actividades ordenadas en el tiempo, en que cada actividad realiza parte del trabajo. Un flujo de trabajo

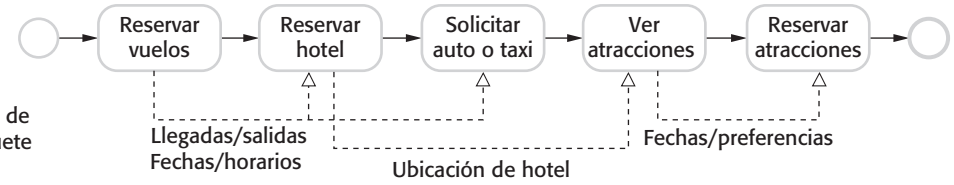


Figura 19.12 Flujo de trabajo de un paquete vacacional

es un modelo de proceso empresarial (es decir, establece los pasos necesarios para alcanzar una meta particular que sea importante para la empresa). En este caso, el proceso empresarial es el servicio de reservación vacacional que ofrece la aerolínea.

El flujo de trabajo es una idea simple y el escenario anterior para reservar unas vacaciones parece ser directo. En la práctica, la combinación de servicios es mucho más compleja de lo que implica este modelo simple. Por ejemplo, se debe considerar la posibilidad de falla del servicio e incorporar mecanismos para manejar dichas fallas. También se deben tomar en cuenta demandas excepcionales hechas por los usuarios de la aplicación. Suponga que un viajero tiene discapacidad y requiere la renta de una silla de ruedas y la entrega de ésta en el aeropuerto. Esto requeriría la implementación y combinación de servicios adicionales, por lo que al flujo de trabajo se agregarían otros pasos.

Usted debe ser capaz de enfrentar estas situaciones en las que el flujo de trabajo deba modificarse porque la ejecución normal de uno de los servicios deriva, por lo general, en una incompatibilidad con la ejecución de algún otro servicio. Por ejemplo, suponga que se reserva un vuelo que saldrá el 1 de junio y regresará el 7 de junio. Entonces el flujo de trabajo procede a la etapa de reservación de hotel. Sin embargo, el hotel tendrá una gran convención hasta el 2 de junio, de modo que no hay habitaciones disponibles. El servicio de reservaciones del hotel reporta esta falta de disponibilidad. Ésta no es una falla; la falta de disponibilidad es una situación común. Por lo tanto, hay que "deshacer" la reservación del vuelo y transmitir la información acerca de la falta de disponibilidad al usuario. Entonces, él deberá decidir si cambia sus fechas o su hotel. En terminología de flujo de trabajo, a esto se le conoce como "acción de compensación". Las acciones de compensación se usan para deshacer acciones que ya se completaron, pero que deben cambiar como resultado de posteriores actividades de flujo de trabajo.

En esencia, el proceso de diseñar nuevos servicios reutilizando los servicios existentes es un proceso de diseño de software con reutilización (figura 19.13). El diseño con reutilización implica inevitablemente compromisos de requerimientos. Deben modificarse los requerimientos "ideales" del sistema para reflejar los servicios que están realmente disponibles, cuyos costos se hallan dentro del presupuesto y cuya calidad de servicio es aceptable.

En la figura 19.13 se muestran seis etapas clave del proceso de construcción de servicio mediante composición:

1. *Formular un bosquejo de flujo de trabajo* En esta etapa inicial del diseño del servicio, se usan los requerimientos para el servicio compuesto como base para la creación de un diseño de servicio "ideal". En esta etapa se debe crear un diseño bastante abstracto con la intención de agregar detalles una vez que se conozca más acerca de los servicios disponibles.

2. *Descubrimiento de servicios* Durante esta etapa del proceso se buscan registros o catálogos de servicios para descubrir cuáles servicios existen, quién los proporciona y los detalles de la provisión del servicio.

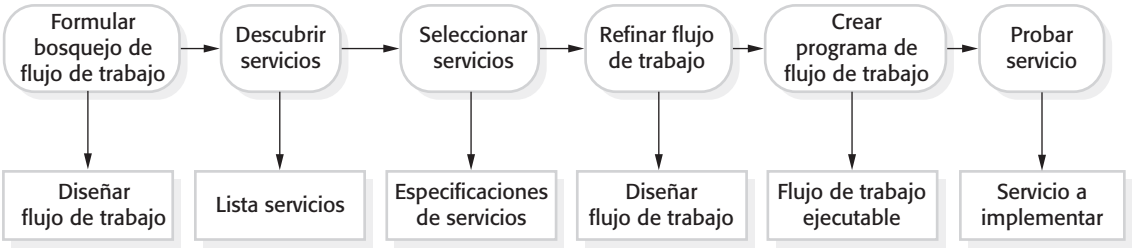


Figura 19.13 Construcción de servicio mediante composición

3. *Seleccionar posibles servicios* A partir del conjunto de posibles candidatos a servicio que se haya descubierto, se seleccionan entonces los posibles servicios que puedan implementar actividades de flujo de trabajo. Desde luego, sus criterios de selección incluirán la funcionalidad de los servicios ofrecidos. También pueden comprender el costo de los servicios y la calidad del servicio ofrecido (respuesta, disponibilidad, etcétera). Es posible decidir elegir algunos servicios con funcionalidad equivalente, que pudieran vincularse con una actividad de flujo de trabajo, dependiendo de los detalles de costo y calidad del servicio.

4. *Refinar el flujo de trabajo* Sobre la base de la información acerca de los servicios que seleccionó, se refina el flujo de trabajo. Esto implica añadir detalles a la descripción abstracta y, tal vez, agregar o eliminar actividades de flujo de trabajo. Entonces se pueden repetir las etapas de descubrimiento y selección del servicio. Una vez que se eligió un conjunto estable de los servicios y se estableció el diseño de flujo de trabajo final, se da paso a la siguiente etapa en el proceso.

5. *Crear un programa de flujo de trabajo* Durante esta etapa, el diseño del flujo de trabajo abstracto se transforma en un programa ejecutable y se define la interfaz del servicio. Se puede usar un lenguaje de programación convencional, como Java o C#, para la implementación del servicio, o un lenguaje de flujo de trabajo, como WS-BPEL. Como se discutió en la sección anterior, la especificación de la interfaz de servicio debe escribirse en WSDL. Esta etapa también puede implicar la creación de interfaces de usuario basadas en Web para permitir el acceso a los nuevos servicios desde un navegador Web.

6. *Prueba de servicio o aplicación terminada* El proceso de probar el servicio terminado y compuesto es más complejo que la prueba de componentes en situaciones donde se usan servicios externos. En la sección 19.3.2 se analizan los conflictos en las pruebas.

En el resto del capítulo nos ocuparemos del diseño y de las pruebas del flujo de trabajo. En la práctica, el descubrimiento de servicio no parece ser un gran problema. Pero todavía es el caso de que la mayor parte de reutilización de servicios se da dentro de las organizaciones, donde pueden descubrirse los servicios usando registros internos y comunicaciones informales entre ingenieros de software. Es posible usar motores de búsqueda estándar para descubrir servicios a disposición pública.

19.3.1 Diseño e implementación del flujo de trabajo

El diseño del flujo de trabajo implica analizar los procesos empresariales existentes o planeados para comprender las diferentes actividades que se realizan y cómo éstas

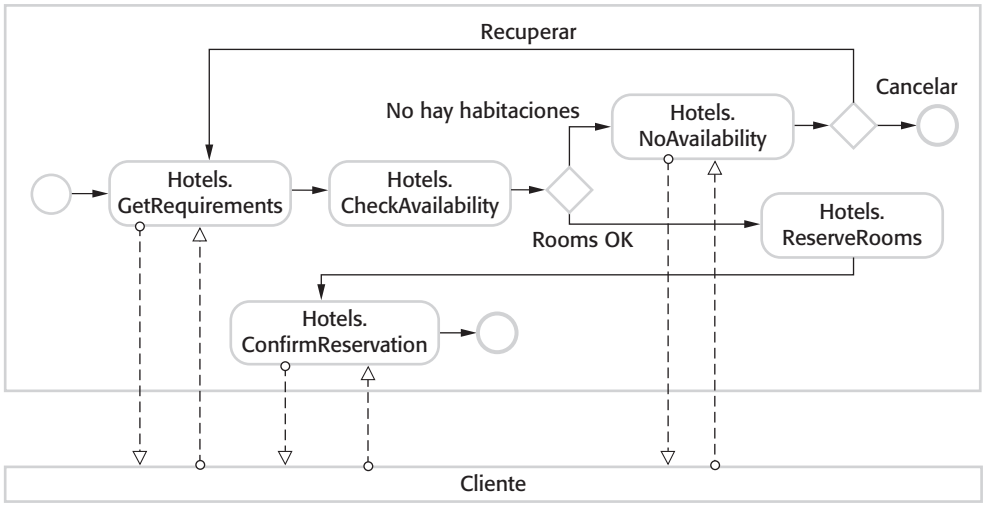


Figura 19.14
Fragmento de un flujo de trabajo de reservación de hotel

intercambian información. Luego, se define el nuevo proceso empresarial en una notación de diseño de flujo de trabajo. Esto establece las etapas implicadas para realizar el proceso y la información que se transmite entre las diferentes etapas del proceso. Sin embargo, los procesos existentes pueden ser informales y depender de las habilidades y capacidades de las personas implicadas: quizá no haya una forma "normal" de trabajar o una definición de proceso. En tales casos, usted deberá usar su conocimiento del proceso actual para diseñar un flujo de trabajo que logre las mismas metas.

Los flujos de trabajo constituyen modelos de proceso empresarial y, por lo general, se representan usando una notación gráfica, como los diagramas de actividad UML o BPMN (Business Process Modeling Notation) (White, 2004a; White y Miers, 2008). Esto ofrece características similares (White, 2004b). Es probable que en el futuro se integren BPMN y los diagramas de actividad UML, y que se defina un estándar de modelado de flujo de trabajo con base en este lenguaje integrado. El autor usa BPMN para los ejemplos de este capítulo.

BPMN es un lenguaje gráfico que es razonablemente fácil de entender. Los mapeos se definen para traducir el lenguaje a descripciones de bajo nivel basadas en XML, en WS-BPEL. Por lo tanto, BPMN se conforma con la pila de estándares de servicio Web que se mostraron en la figura 19.2.

La figura 19.14 es un ejemplo de un modelo BPMN simple de parte del escenario del paquete vacacional anterior. El modelo ilustra un flujo de trabajo simplificado para reservación de hotel y supone la existencia de un servicio Hotels con operaciones asociadas llamadas GetRequirements, CheckAvailability, ReserveRooms, NoAvailability, ConfirmReservation y CancelReservation. El proceso incluye obtener requerimientos del cliente, verificar disponibilidad de habitación, y después, si hay habitaciones disponibles, hacer una reservación para las fechas requeridas.

Este modelo introduce algunos de los conceptos centrales de BPMN que se usan para crear modelos de flujo de trabajo:

1. Las actividades se representan mediante un rectángulo con esquinas redondeadas. Una actividad puede ejecutarse por una persona o mediante un servicio automatizado.

2. Los eventos se representan por medio de círculos. Un evento es algo que sucede durante un proceso empresarial. Un círculo sencillo se usa para representar un evento inicial y un círculo más oscuro para un evento final. Un círculo doble (no se ilustra) se usa para representar un evento intermedio. Los eventos pueden ser eventos de reloj, lo que en consecuencia permite que los flujos de trabajo se ejecuten periódicamente o de manera cronometrada.

3. Un diamante se usa para representar una compuerta. Una compuerta es una etapa en el proceso donde se hace una elección. Por ejemplo, en la figura 19.14, hay una elección tomada con base en si existen o no habitaciones disponibles.

4. Una flecha sólida se usa para mostrar la secuencia de actividades, mientras que una flecha punteada representa mensajes que fluyen entre actividades. En la figura 19.14 dichos mensajes se transmiten entre el servicio de reservación de hotel y el cliente.

Estas características clave son suficientes para describir la esencia de la mayoría de los flujos de trabajo. Sin embargo, BPMN incluye muchas características adicionales que aquí no se pueden describir por falta de espacio. Éstas agregan información a una descripción de proceso empresarial que permite su traducción automática en un servicio ejecutable. Por lo tanto, los servicios Web, basados en las composiciones de servicio descritas en BPMN, pueden generarse directamente a partir de un modelo de proceso empresarial.

La figura 19.14 ilustra el proceso que se realiza en una organización, la compañía que proporciona un servicio de reservaciones. Sin embargo, el beneficio clave de un enfoque orientado a servicios es que soporta computación entre organizaciones. Esto significa que un cómputo implica servicios en diferentes compañías. Lo anterior se representa en BPMN mediante el desarrollo de flujos de trabajo separados para cada una de las organizaciones implicadas con las interacciones entre ellas.

Para ilustrar esto se usa aquí un ejemplo diferente, extraído de computación de alto rendimiento. Se ha propuesto un enfoque orientado a servicios para permitir que se compartan recursos como las computadoras. En este ejemplo, suponga que una computadora de procesamiento de vector (una máquina que puede realizar computaciones paralelas sobre arreglos de valores) se ofrece como un servicio (VectorProcService) por parte de un laboratorio de investigación. A ella se accede a través de otro servicio llamado SetupComputation. En la figura 19.15 se muestran dichos servicios y sus interacciones.

En este ejemplo, el flujo de trabajo para el servicio SetupComputation solicita acceso a un procesador de vectores y, si está disponible un procesador, establece el cómputo requerido y descarga datos al servicio de procesamiento. Una vez completo el cómputo, los resultados se almacenan en la computadora local. El flujo de trabajo para VectorProcService verifica si está disponible algún procesador, asigna recursos para el cómputo, inicia el sistema, realiza el cómputo y regresa los resultados al servicio cliente.

En términos BPMN, el flujo de trabajo para cada organización se representa en una pool separada. Se muestra de manera gráfica al encerrar en un rectángulo el flujo de trabajo para cada participante en el proceso, con el nombre escrito verticalmente en la arista izquierda. Los flujos de trabajo definidos en cada pool se coordinan mediante el intercambio de mensajes; no se permite el flujo de secuencia entre las actividades de diferentes pools. En situaciones donde diferentes partes de una organización están implicadas en un flujo de trabajo, esto puede mostrarse mediante la separación de pools en

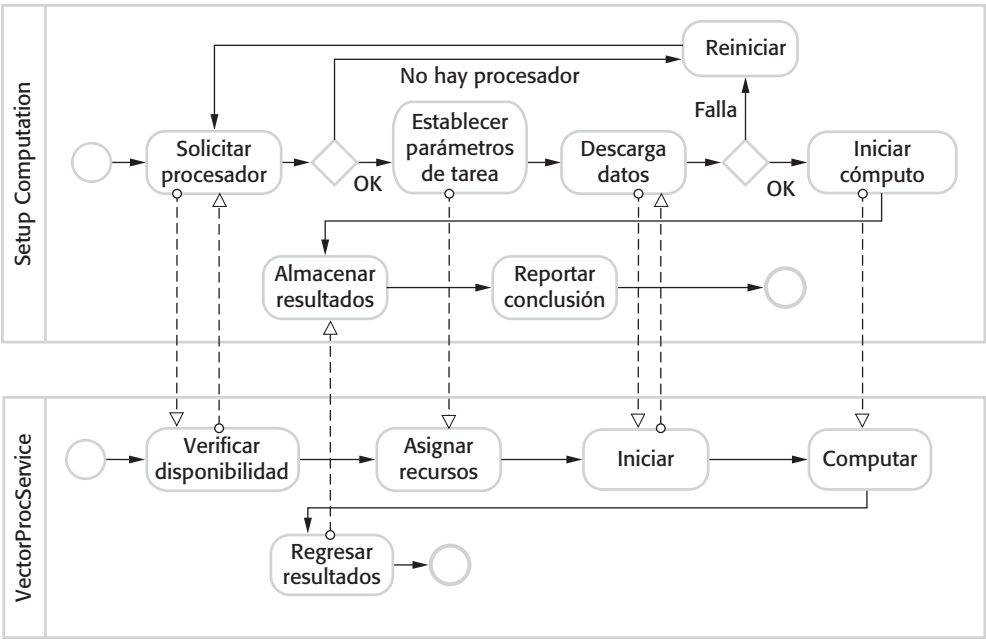


Figura 19.15 Flujos de trabajo en interacción

"carriles" (*lanes*) nombrados. Cada carril muestra las actividades en dicha parte de la organización.

Una vez diseñado el modelo de proceso empresarial, debe redefinirse dependiendo de los servicios descubiertos. Como se sugiere en el análisis de la figura 19.13, el modelo puede pasar por algunas iteraciones hasta que se cree un diseño que permita la máxima reutilización posible de los servicios disponibles.

Si está disponible el diseño final, debe convertirse entonces en un programa ejecutable. Esto puede implicar dos actividades:

1. Implementar los servicios que no están disponibles para reutilización. Puesto que los servicios son independientes del lenguaje de implementación, dichos servicios pueden escribirse en cualquier lenguaje. Los entornos de desarrollo Java y C# brindan soporte para composición de servicios Web.
2. Generar una versión ejecutable del modelo de flujo de trabajo. Por lo general, esto implica traducir el modelo en WS-BPEL, de manera manual o automática. Aunque existen muchas herramientas disponibles para automatizar el proceso BPMN-WS-BPEL, también hay algunas circunstancias donde es difícil generar un código WS-BPEL legible desde un modelo de flujo de trabajo.

Para brindar soporte directo a la implementación de las composiciones de servicios Web, se han desarrollado muchos estándares de servicio Web. Como se explicó en la introducción del capítulo, el lenguaje estándar basado en XML es WS-BPEL (Business Process Execution Language, esto es, lenguaje de ejecución de proceso empresarial), que es un "lenguaje de programación" para controlar interacciones entre servicios. Éste

recibe apoyo de estándares adicionales como WS-Coordination (Cabrera *et al*., 2005), que se usa para especificar cómo se coordinan los servicios y WS-CDL (Choreography Description Language, es decir, lenguaje de descripción de coreografía) (Kavantzas *et al*., 2004), que es un medio para definir los intercambios de mensaje entre participantes (Andrews *et al*., 2003).

19.3.2 Pruebas del servicio

Las pruebas son importantes en todos los procesos de desarrollo de sistemas, pues demuestran que un sistema cumple con sus requerimientos funcionales y no funcionales, y detectan defectos introducidos durante el proceso de desarrollo. Muchas técnicas de prueba, como las inspecciones del programa y las pruebas de cobertura, dependen del análisis del código fuente del software. Sin embargo, cuando un proveedor externo ofrece sus servicios, no está disponible el código fuente de la implementación del servicio. Por lo tanto, las pruebas del sistema basado en servicios no pueden usar técnicas probadas basadas en código fuente.

Además de los problemas de comprender la implementación del servicio, los examinadores pueden enfrentar también más dificultades cuando se prueban los servicios y la combinación de servicios:

1. Los servicios externos están bajo el control del proveedor del servicio y no del usuario del servicio. El proveedor del servicio puede retirar dichos servicios en cualquier momento o puede modificarlos, lo que invalida cualquier prueba previa de la aplicación. Dichos problemas se manejan en los componentes de software al mantener diferentes versiones del componente. Sin embargo, en la actualidad, no hay estándares propuestos para lidiar con las versiones del servicio.

2. La visión a largo plazo de la SOA es para que los servicios se vinculen de manera dinámica a aplicaciones orientadas a servicios. Esto significa que una aplicación no siempre puede usar el mismo servicio cada vez que se ejecuta. En consecuencia, las pruebas pueden ser exitosas cuando una aplicación se enlaza a un servicio particular, pero no se puede garantizar que dicho servicio se usará durante una ejecución real del sistema.

3. El comportamiento no funcional de un servicio no depende simplemente de cómo se usa por parte de la aplicación que se pone a prueba. Un servicio puede desempeñarse bien durante las pruebas porque no opera bajo una carga pesada. En la práctica, el comportamiento de servicio observado podría ser diferente debido a las demandas hechas por otros usuarios del servicio.

4. El modelo de pago para servicios podría hacer que las pruebas del servicio sean muy costosas. Existen diferentes modelos de pago posibles: algunos servicios pueden estar a disposición gratuita, por algunos hay que pagar una suscripción, y otros se pagan sobre una base "por uso". Si los servicios son gratuitos, entonces el proveedor del servicio no querrá que se carguen en aplicaciones sujetas a prueba; si se requiere una suscripción, entonces tal vez un usuario del servicio podría estar renuente a comprometerse en un acuerdo de suscripción antes de probar el servicio. De igual modo, si el servicio se basa en pago con base en el uso, los usuarios del servicio pueden descubrir que el costo de las pruebas resulta prohibitivo.

5. Ya se discutió la noción de acciones de compensación que se invocan cuando ocurre una excepción y los compromisos previos realizados (como una reservación de vuelo) deben revocarse. Existe un problema en poner a prueba tales acciones, pues dependen de la falla de otros servicios. Asegurar que estos servicios realmente fallan durante el proceso de pruebas puede ser muy difícil.

Dichos problemas son particularmente agudos cuando se usan servicios externos. Son menos serios cuando los servicios se emplean dentro de la misma compañía o cuando compañías cooperativas confían en los servicios ofrecidos por sus socios. En tales casos, puede estar disponible el código fuente para guiar el proceso de pruebas, y es improbable que sea un problema el pago por servicios. La resolución de estos problemas de pruebas y la elaboración de lineamientos, herramientas y técnicas para probar las aplicaciones orientadas a servicios siguen siendo un importante tema de investigación.

PUNTOS CLAVE

- La arquitectura orientada a servicios es un enfoque a la ingeniería de software donde servicios estandarizados de reutilización son los bloques constructores básicos para los sistemas de aplicación.
- Las interfaces de servicio pueden definirse en un lenguaje que se base en XML llamado WSDL. Una especificación WSDL incluye una definición de los tipos de interfaz y operaciones, el protocolo de enlace que usa el servicio y la ubicación del servicio.
- Los servicios pueden clasificarse como utilitarios que ofrecen una funcionalidad de propósito general, empresariales que implementan parte de un proceso empresarial, o de coordinación que regulan la ejecución de otros servicios.
- El proceso de ingeniería de servicio implica la identificación de servicios candidatos para su implementación, la definición de la interfaz del servicio, y la implementación, la prueba y el despliegue del servicio.
- Pueden definirse interfaces de servicio para sistemas de software heredado que siguen siendo útiles para una organización. Entonces la funcionalidad de los sistemas heredados puede reutilizarse en otras aplicaciones.
- El desarrollo de software usando servicios se basa en la idea de que los programas se crean al combinar y configurar servicios para desarrollar nuevos servicios compuestos.
- Los modelos de proceso empresarial definen las actividades y el intercambio de información que tienen lugar en un proceso empresarial. Las actividades en el proceso empresarial pueden implementarse mediante servicios, de manera que el modelo de proceso empresarial representa una composición de servicios.

LECTURAS SUGERIDAS

Existe muchísimo material de tutoriales en la Web que cubre todos los aspectos de los servicios Web. Sin embargo, los siguientes dos libros de Thomas Erl ofrecen las mejores perspectivas y descripciones de los servicios y los estándares de servicio. A diferencia de la mayoría de los libros,

Erl incluye cierta discusión de los conflictos de la ingeniería de software en la computación orientada a servicios. También ha escrito libros más especializados acerca del diseño de servicios y patrones de diseño SOA, aunque por lo general se dirigen a lectores con experiencia en la implementación de SOA.

Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. El enfoque principal de este libro radica en las tecnologías subyacentes basadas en XML (SOAP, WSDL, BPEL, etcétera), que son un marco de referencia para SOA. (T. Erl, Prentice Hall, 2004.)

Service-Oriented Architecture: Concepts, Technology and Design. Se trata de un libro más general acerca de la ingeniería de sistemas orientados a servicios. Hay un poco de traslape con el texto anterior, pero Erl se concentra principalmente en explicar cómo puede usarse un enfoque orientado a servicios en todas las etapas del proceso de software. (T. Erl, Prentice Hall, 2005.)

"SOA realization: Service design principles". Este breve artículo Web es un excelente panorama de los conflictos a considerar en el diseño de servicios. (D. J. N. Artus, IBM, 2006.) http://www.ibm.com/developerworks/webservices/library/ws-soa-design/.

EJERCICIOS

19.1. ¿Cuáles son las distinciones más importantes entre servicios y componentes de software?

19.2. Explique por qué las SOA deben basarse en estándares.

19.3. Con la misma notación, extienda la figura 19.5 para incluir definiciones para MaxMinType e InDataFault. Las temperaturas deben representarse como enteros con un campo adicional que indique si la temperatura está en grados Fahrenheit o grados Celsius. InDataFault debe ser un tipo sencillo que consta de un código de error.

19.4. Defina una especificación de interfaz para los servicios Convertidor de divisas y Comprobación calificación crediticia que se muestran en la figura 19.7.

19.5. Diseñe posibles mensajes de entrada y salida para los servicios que se muestran en la figura 19.11. Puede especificarlos en UML o en XML.

19.6. Fundamentando con razones su respuesta, sugiera dos tipos de aplicación importantes donde *no* recomendaría el uso de arquitectura orientada a servicios.

19.7. En la sección 19.2.1 se introdujo un ejemplo de una compañía que desarrolló un servicio de catálogo que usan los sistemas de procuración basados en la Web de los clientes. Con BPMN, diseñe un flujo de trabajo que use este servicio de catálogo para buscar y realizar pedidos para equipo de cómputo.

19.8. Explique qué se entiende por "acción de compensación" y, con un ejemplo, demuestre por qué estas acciones deben incluirse en los flujos de trabajo.

19.9. Para el ejemplo del servicio de reservación de paquete vacacional, diseñe un flujo de trabajo que reservará transporte terrestre para un grupo de pasajeros que llegan a un aeropuerto. Debe ofrecer la opción de reservar un taxi o rentar un auto. Puede suponer que las compañías de taxis y de alquiler de autos ofrecen servicios Web para hacer una reservación.

19.10. Con un ejemplo, explique con detalle por qué son difíciles las pruebas extensas de los servicios que incluyen acciones de compensación.

REFERENCIAS

Andrews, T., Curbera, F., Goland, Y., Klein, J. y Al., E. (2003). "Business Process Execution Language for Web Services". http://www-128.ibm.com/developerworks/library/ws-bpel/.

Cabrera, L. F., Copeland, G. y Al., E. 2005. "Web Services Coordination (WS-Coordination)". ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

Carr, N. (2009). *The Big Switch: Rewiring the World from Edison to Google, Reprint edition*. Nueva York: W.W. Norton & Co.

Erl, T. (2004). *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice Hall.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice Hall.

Kavantzas, N., Burdett, D. y Ritzinger, G. 2004. "Web Services Choreography Description Language Version 1.0". http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

Lovelock, C., Vandermerwe, S. y Lewis, B. (1996). *Services Marketing*. Englewood Cliffs, NJ: Prentice Hall.

Newcomer, E. y Lomow, G. (2005). *Understanding SOA with Web Services*. Boston: Addison-Wesley.

Owl_Services_Coalition. 2003. "OWL-S: Semantic Markup for Web Services". http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

Pautasso, C., Zimmermann, O. y Leymann, F. (2008). "RESTful Web Services vs 'Big' Web Services: Making the Right Architectural Decision". Proc. *WWW 2008*, Beijing, China: 805–14.

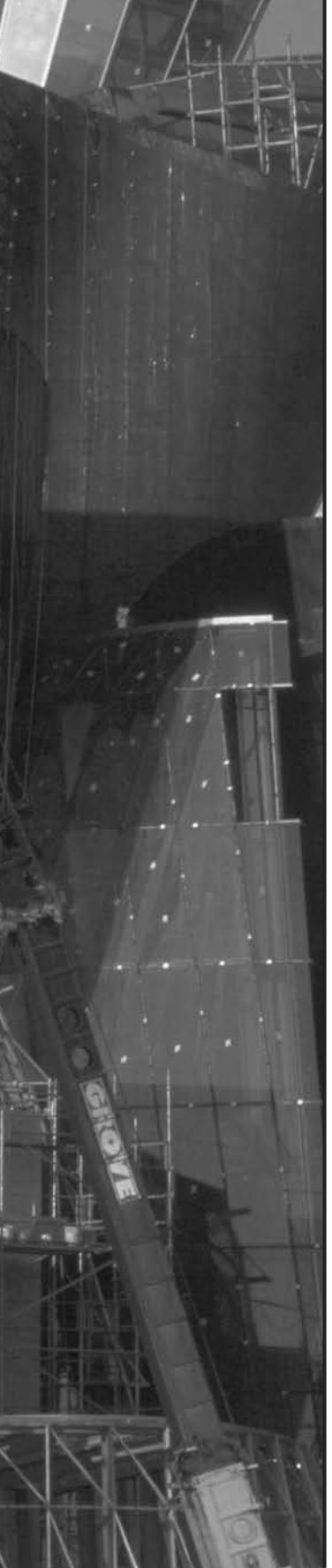
Richardson, L. y Ruby, S. (2007). *RESTful Web Services*. Sebastopol, Calif.: O'Reilly Media Inc.

Turner, M., Budgen, D. y Brereton, P. (2003). "Turning Software into a Service". *IEEE Computer*, **36** (10), 38–45.

White, S. A. (2004a). "An Introduction to BPMN". http://www.bpmn.org/Documents/Introduction%20to%20BPMN.

White, S. A. (2004b). "Process Modelling Notations and Workflow Patterns". En *Workflow Handbook 2004*. Fischer, L. (ed.). Lighthouse Point, Fla.: Future Strategies Inc. 265–294.

White, S. A. y Miers, D. (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, Fla.: Future Strategies Inc.



20

Software embebido

Objetivos

El objetivo de este capítulo es introducirlo a algunas de las características de los sistemas embebidos de tiempo real y la ingeniería de software en tiempo real. Al estudiar este capítulo:

- comprenderá el concepto de software embebido, que se usa para controlar sistemas que deben reaccionar frente a eventos externos en su entorno;
- se introducirá a un proceso de diseño para sistemas de tiempo real, en el que los sistemas de software se organizan como un conjunto de procesos cooperativos;
- conocerá tres patrones arquitectónicos utilizados comúnmente en el diseño de sistemas embebidos de tiempo real;
- entenderá la organización de los sistemas operativos de tiempo real y el papel que éstos desempeñan en un sistema embebido de tiempo real.

Contenido

20.1 Diseño de sistemas embebidos
20.2 Patrones arquitectónicos
20.3 Análisis de temporización
20.4 Sistemas operativos de tiempo real

Las computadoras se utilizan para controlar una amplia gama de sistemas, desde máquinas domésticas simples y controladores de juego, hasta el total de las plantas manufactureras. Estas computadoras interactúan directamente con dispositivos de hardware. Su software debe reaccionar a eventos generados por el hardware y emitir a menudo señales de control en respuesta a tales eventos. Estas señales dan por resultado una acción, como el inicio de una llamada telefónica, el movimiento de un carácter en la pantalla, la apertura de una válvula o el despliegue del estatus del sistema. El software en dichos sistemas está embebido en el hardware del sistema, con frecuencia en la memoria de sólo lectura, y por lo general responde, en tiempo real, a eventos del entorno del sistema. Por tiempo real se entiende que el sistema de software tiene un plazo para responder a los eventos externos. Si éste no se cumple, entonces el sistema hardware-software global no funcionará correctamente.

El software embebido es muy importante desde el punto de vista económico, porque ahora casi todos los dispositivos eléctricos incluyen software. Por consiguiente, existen muchos más sistemas de software embebido que de otros tipos. Quizás usted tenga en casa tres o cuatro computadoras personales, pero es probable que cuente también con 20 o 30 sistemas embebidos, como sistemas en teléfonos, hornos de microondas, etcétera.

La respuesta en tiempo real es la diferencia crítica entre los sistemas embebidos y otros sistemas de software, tales como los sistemas de información, los basados en la Web o los de software personal, cuyo propósito fundamental es el procesamiento de datos. Para sistemas que no son de tiempo real, la corrección de un sistema se puede definir al especificar cómo las entradas del sistema se mapean a las salidas correspondientes que debe producir el sistema. En respuesta a una entrada, el sistema debe generar una salida correspondiente y, muchas veces, deben almacenarse algunos datos. Por ejemplo, si se elige un comando *create* en un sistema de información de pacientes, entonces la respuesta correcta del sistema es crear en una base de datos un nuevo registro del paciente, y confirmar que lo ha hecho; dentro de límites razonables, no importa cuánto tarde.

Sin embargo, en un sistema de tiempo real, la corrección depende tanto de la respuesta a una entrada como del tiempo que tarda en generar dicha respuesta. Si el sistema aplaza mucho la respuesta, entonces la respuesta requerida puede resultar ineficaz. Por ejemplo, si el software embebido que controla el sistema de frenado de un automóvil es muy lento, puede ocurrir un accidente al ser imposible detener a tiempo el vehículo.

Por consiguiente, el tiempo es inseparable en la definición de un sistema de software de tiempo real:

> *Un sistema de software de tiempo real es un sistema cuya correcta operación depende tanto de los resultados producidos por el sistema como del tiempo en que se producen dichos resultados. Un "sistema blando de tiempo real" es un sistema cuya operación se degrada si los resultados no se producen de acuerdo con los requerimientos de tiempo especificados. Si los resultados no se producen según la especificación de tiempo en un "sistema duro de tiempo real", se considera una falla del sistema.*

La respuesta oportuna es un factor importante en todos los sistemas embebidos, aunque no todos estos sistemas requieren una respuesta muy rápida. Por ejemplo, el software de la bomba de insulina que se usó como modelo en varios capítulos de este libro es un sistema embebido. No obstante, aun cuando requiere verificar el nivel de glucosa a intervalos periódicos, no precisa responder muy rápidamente a eventos externos. El software

de la estación meteorológica a campo abierto también es un sistema embebido, aunque, de nuevo, no requiere una respuesta rápida a eventos externos.

Además de la necesidad de respuesta en tiempo real, existen entre los sistemas embebidos otras diferencias importantes y otros tipos de sistema de software:

1. Los sistemas embebidos por lo general operan de manera continua, es decir, su operación no tiene fin. Comienzan cuando el hardware se activa y deben ejecutarse hasta que el hardware se desactiva. Esto significa que también pueden usarse técnicas para ingeniería de software fiable, como se explicó en el capítulo 13, para garantizar la operación continua. El sistema de tiempo real puede incluir mecanismos de actualización que soporten reconfiguración dinámica, de forma que el sistema pueda actualizarse mientras se encuentra en servicio.

2. Las interacciones con el entorno del sistema son incontrolables e impredecibles. En sistemas interactivos el ritmo de la interacción se controla mediante el sistema y, al limitar las opciones del usuario, los eventos a procesar se conocen por adelantado. En contraste, los sistemas embebidos de tiempo real deben responder en cualquier momento a sucesos inesperados. Esto conduce a un diseño de sistemas de tiempo real basado en concurrencia, con algunos procesos que se ejecutan en paralelo.

3. Puede haber limitaciones físicas que afecten el diseño de un sistema. Ejemplos de esto incluyen las limitaciones en la energía disponible al sistema y al espacio físico que ocupa el hardware. Dichas limitaciones pueden generar requerimientos para el software embebido, tal como la necesidad de conservar energía y así prolongar la vida de la batería. Las limitaciones de tamaño y peso pueden significar que el software debe hacerse cargo de algunas funciones de hardware debido a la necesidad de restringir el número de chips usados en el sistema.

4. Tal vez se requiera interacción directa con el hardware. En los sistemas interactivos y en los sistemas de información, hay una capa de software (los controladores del dispositivo) que ocultan el hardware del sistema operativo. Esto es posible porque sólo se puede conectar ciertos tipos de dispositivos a dichos sistemas, tales como teclados, ratones, pantallas, etcétera. En contraste, los sistemas embebidos deben interactuar con una amplia gama de dispositivos de hardware que no tienen controladores de dispositivos separados.

5. Los conflictos de protección y fiabilidad pueden dominar el diseño del sistema. Muchos sistemas embebidos controlan dispositivos cuyas fallas pueden tener altos costos humanos o económicos. Por lo tanto, la confiabilidad es crítica y el diseño del sistema debe garantizar comportamiento crítico para la protección en todo momento. Con frecuencia, esto conduce al diseño a un enfoque conservador, en el que se usan técnicas de probada eficacia en vez de técnicas más recientes que pueden introducir nuevos modos de falla.

Los sistemas embebidos pueden considerarse como sistemas reactivos; esto es, deben reaccionar ante los eventos a la velocidad de ese entorno (Berry, 1989; Lee, 2002). A menudo, los tiempos de respuesta se rigen por las leyes de la física en vez de ser elegidos por la conveniencia humana. Esto está en contraste con otros tipos de software en que el sistema controla la velocidad de la interacción. Por ejemplo, el procesador de texto,

usado para escribir el presente libro, puede comprobar la ortografía y gramática, y no hay límites prácticos en el tiempo que tarda en ello.

20.1 Diseño de sistemas embebidos

El proceso de diseño para sistemas embebidos es un proceso de ingeniería de sistemas en el que los diseñadores de software deben considerar a detalle el diseño y el rendimiento del hardware del sistema. Parte del proceso de diseño del sistema puede comprender la decisión de cuáles capacidades del sistema han de implementarse en software y cuáles en hardware. Para numerosos sistemas de tiempo real embebidos en productos al consumidor, como los sistemas en los teléfonos celulares, los costos y el consumo de energía del hardware son críticos. Para soportar sistemas embebidos pueden usarse procesadores específicos, y en algunos sistemas tal vez deba diseñarse y construirse hardware de propósito especial.

Ello significa que, en la mayoría de los sistemas de tiempo real, es poco práctico un proceso de diseño de software descendente, en el que el diseño comience con un modelo abstracto que se descomponga y se desarrolle en una serie de etapas. Las decisiones de bajo nivel en hardware, software de soporte y temporización (*timing*) del sistema deben considerarse al inicio del proceso. Esto limita la flexibilidad de los diseñadores del sistema y puede significar que la funcionalidad de software adicional, como la batería y la administración de energía, deba incluirse en el sistema.

Puesto que los sistemas embebidos son sistemas reactivos que reaccionan a los eventos en su entorno, el enfoque más general al diseño de software embebido de tiempo real se basa en un modelo estímulo-respuesta. Un estímulo es un evento que ocurre en el entorno de sistemas de software que hace que el sistema reaccione de alguna forma; una respuesta es una señal o un mensaje que envía el software a su entorno.

Es posible definir el comportamiento de un sistema de tiempo real al elaborar una lista de los estímulos recibidos por el sistema, las respuestas asociadas y el momento cuando debe producirse la respuesta. Por ejemplo, la figura 20.1 muestra posibles estímulos y respuestas de sistema para un sistema de alarma contra robo. En la sección 20.2.1 se encuentra más información sobre este sistema.

Los estímulos se presentan en dos clases:

1. *Estímulos periódicos* Ocurren a intervalos predecibles. Por ejemplo, el sistema puede examinar un sensor cada 50 milisegundos y actuar (responder) a partir del valor de dicho sensor (el estímulo).
2. *Estímulos no periódicos* Ocurren de manera irregular e impredecible, y por lo general se señalan mediante el mecanismo de interrupción de la computadora. Un ejemplo de tal estímulo sería una interrupción que indique que se completó una transferencia I/O y que había datos disponibles en un buffer.

Los estímulos provienen de sensores en el entorno del sistema, y las respuestas se envían a actuadores, como se muestra en la figura 20.2. Un lineamiento de diseño general

Estímulos	Respuesta
Sensor individual positivo	Iniciar alarma; encender luces alrededor del sitio del sensor positivo.
Dos o más sensores positivos	Iniciar alarma; encender luces alrededor de los sitios de los sensores positivos; llamar a la policía con ubicación de entrada forzada del sospechoso.
Caída de voltaje entre el 10 y 20%	Cambiar a batería de respaldo; efectuar prueba de suministro de energía.
Caída de voltaje de más del 20%	Cambiar a batería de respaldo; iniciar alarma; llamar a la policía; efectuar prueba de suministro de energía.
Falla en el suministro de energía	Llamar a servicio técnico.
Falla de sensor	Llamar a servicio técnico.
Consola de botón de pánico positivo	Iniciar alarma; encender luces alrededor de consola; llamar a la policía.
Inicializar valores de la alarma	Apagar todas las alarmas activas; apagar todas las luces que se hayan encendido.

Figura 20.1 Estímulos y respuestas para un sistema de alarma contra robo

para los sistemas de tiempo real es tener dos procesos independientes para cada tipo de sensor y actuador (figura 20.3). Dichos actuadores controlan el equipo, tal como una bomba, que hace entonces cambios al entorno del sistema. Los mismos actuadores también pueden generar estímulos. Los estímulos de los actuadores indican con frecuencia que ocurrió algún problema, el cual debe manejar el sistema.

Para cada tipo de sensor puede haber un proceso de gestión de sensor que maneje la recolección de datos de los sensores. Los procesamientos de datos calculan las respuestas requeridas para los estímulos recibidos por el sistema. Los procesos de control

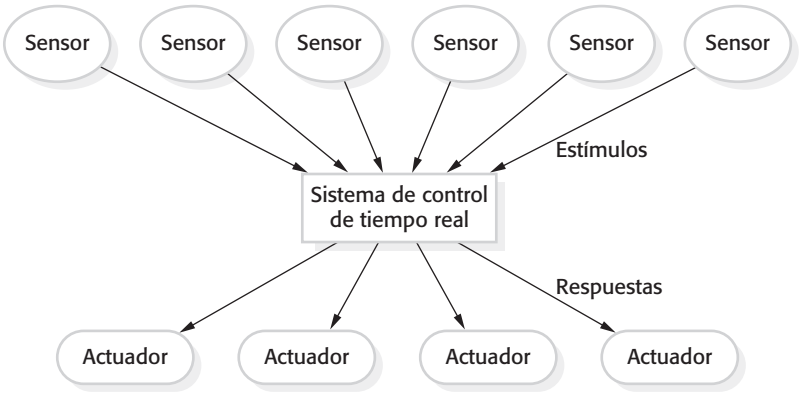


Figura 20.2 Modelo general de un sistema embebido de tiempo real

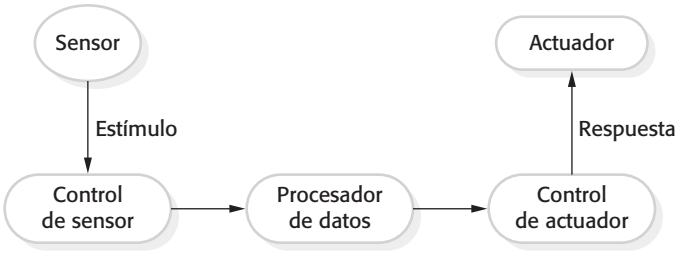


Figura 20.3 Procesos de sensor y actuador

del actuador se asocian con cada actuador y gestionan la operación de ese actuador. Este modelo permite la rápida recopilación de datos desde el sensor (antes de que sobrescriba la entrada siguiente) y hace posible que después se realicen el procesamiento y la respuesta asociada del actuador.

Un sistema de tiempo real debe responder a estímulos que ocurren en diferentes momentos. Por lo tanto, debe organizar la arquitectura del sistema para que, tan pronto como se reciba un estímulo, el control se transfiera al manejador correcto. Esto es poco práctico en programas secuenciales. Por consiguiente, los sistemas de software de tiempo real se diseñan por lo general como un conjunto de procesos cooperativos concurrentes. Para apoyar la gestión de dichos procesos, la plataforma de ejecución en que se efectúa el sistema de tiempo real puede incluir un sistema operativo de tiempo real (que se explica en la sección 20.4). Se accede a las funciones proporcionadas por este sistema operativo a través del sistema de soporte en tiempo de ejecución para el lenguaje de programación de tiempo real utilizado.

No existe un proceso de diseño del sistema embebido estándar. En vez de ello, se usan diferentes procesos que dependen del tipo de sistema, el hardware disponible y la organización que desarrolle el sistema. Las siguientes actividades pueden incluirse en un proceso de diseño de software de tiempo real:

1. *Selección de plataforma* En esta actividad se elige una plataforma de ejecución para el sistema (es decir, el hardware y el sistema operativo de tiempo real a utilizar). Los factores que influyen dichas elecciones comprenden restricciones de temporización sobre el sistema, limitaciones en la energía disponible, experiencia del equipo de desarrollo y precio tope para el sistema entregado.
2. *Identificación de estímulos/respuestas* Esto implica identificar los estímulos que debe procesar el sistema y la respuesta o respuestas asociadas para cada estímulo.
3. *Análisis de temporización* Para cada estímulo y respuesta asociada se identifican las restricciones de temporización que se aplican tanto al estímulo como al procesamiento de la respuesta. Se usan para establecer los plazos de los procesos del sistema.
4. *Diseño de procesos* En esta etapa se agrega el estímulo y el procesamiento de respuesta en algunos procesos concurrentes. Un buen punto de partida para diseñar la arquitectura del proceso lo constituyen los patrones arquitectónicos descritos en la sección 20.2. Posteriormente, se optimiza la arquitectura del proceso para reflejar los requerimientos específicos que deben implementarse.
5. *Diseño de algoritmo* Para cada estímulo y respuesta se diseñan algoritmos que realizan los cálculos requeridos. Tal vez se deban desarrollar los diseños de algoritmo

en etapas relativamente tempranas del proceso de diseño, para dar un indicio de la cantidad de procesamiento requerido y del tiempo necesario para completar dicho procesamiento. Esto es especialmente importante para tareas de cómputo intenso, como el procesamiento de señales.

6. *Diseño de datos* Se especifica la información que intercambian los procesos y eventos que coordinan el intercambio de información, y se diseñan las estructuras de datos para administrar este intercambio de información. Varios procesos concurrentes pueden compartir estas estructuras de datos.
7. *Planeación del proceso* Se diseña un sistema de planeación que garantice que los procesos iniciarán a tiempo para cumplir sus plazos.

El orden de dichas actividades en el proceso de diseño de software de tiempo real depende del tipo de sistema a desarrollar, así como de sus requerimientos de proceso y plataforma. En algunos casos se podrá seguir un enfoque bastante abstracto, que comience con los estímulos y el procesamiento asociado, y decidir al final del proceso sobre las plataformas de hardware y de ejecución. En otros casos, la elección del hardware y del sistema operativo se efectúa antes de comenzar el diseño del software. Ante tal situación, se debe diseñar el software para considerar las restricciones impuestas por las capacidades del hardware.

Los procesos en un sistema de tiempo real deben coordinarse y compartir información. Los mecanismos de coordinación de proceso garantizan la exclusión mutua para los recursos compartidos. Cuando un proceso modifica un recurso compartido, otros procesos no podrán cambiar dicho recurso. Los mecanismos para probar la exclusión mutua incluyen semáforos (Dijkstra, 1968), monitores (Hoare, 1974) y regiones críticas (Brinch-Hansen, 1973). Estos mecanismos de sincronización de proceso se describen en la mayoría de los textos acerca de sistemas operativos (Silberschatz *et al*., 2008; Tanenbaum, 2007).

Al diseñar el intercambio de información entre procesos, se debe considerar el hecho de que tales procesos pueden ejecutarse a diferentes velocidades. Un proceso genera información; el otro proceso consume esa información. Si el productor se ejecuta más rápido que el consumidor, nueva información podría sobrescribir un ítem de información leído previamente antes de que el proceso consumidor lea la información original. Si el proceso consumidor se ejecuta más rápido que el proceso productor, el mismo ítem podría leerse dos veces.

Para solucionar este problema, se debe implementar intercambio de información mediante un buffer compartido y usar mecanismos de exclusión mutua para controlar el acceso a ese buffer. Esto significa que la información no puede sobrescribirse antes de leerse y que la información no debe leerse dos veces. La figura 20.4 ilustra la noción de buffer compartido. Por lo general, esto se implementa como una cola circular, de manera que la falta de concordancia de velocidades entre los procesos productor y consumidor pueda acomodarse sin tener que demorar la ejecución del proceso.

El proceso productor siempre ingresa datos en la ubicación del buffer al final de la cola (representada como v10 en la figura 20.4). El proceso consumidor recupera en todo momento información del inicio de la cola (representada como v1 en la figura 20.4). Después de que el proceso consumidor recupera la información, el inicio de la cola se ajusta para apuntar al siguiente ítem (v2). Luego de que el proceso productor agrega información, el final de la cola se ajusta para apuntar al siguiente espacio (*slot*) libre en la cola.

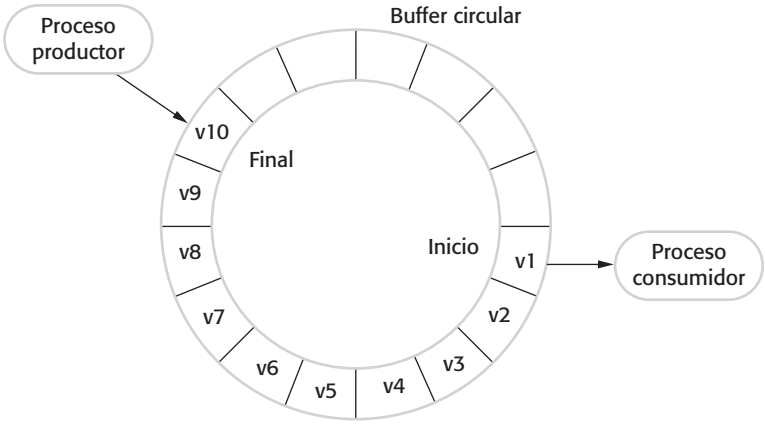


Figura 20.4 Procesos productor/consumidor que comparten un buffer circular

Desde luego, es importante garantizar que los procesos productor y consumidor no traten de acceder al mismo ítem al mismo tiempo (es decir, cuando Inicio = Final). También debemos asegurarnos de que el proceso productor no agregue ítems a un buffer lleno y que el proceso consumidor no tome ítems desde un buffer vacío. Para ello, implemente el buffer circular como un proceso con las operaciones Get y Put para acceder al buffer. La operación Put (ingresar) es llamada por el proceso productor y la operación Get (obtener) por el proceso consumidor. Las primitivas de sincronización, como los semáforos o las regiones críticas, se usan para asegurar que las operaciones de Get y Put están sincronizadas, de manera que no accedan a la misma ubicación al mismo tiempo. Si el buffer está lleno, el proceso Put debe esperar hasta que un espacio (*slot*) esté libre; si el buffer está vacío, el proceso Get debe esperar hasta hacer una entrada.

Una vez que se ha elegido la plataforma de ejecución para el sistema, se ha diseñado una arquitectura de proceso y se ha determinado una política de planeación, es necesario comprobar que el sistema cumplirá sus requerimientos de temporización. Esto se puede hacer mediante análisis estático del sistema a través del conocimiento del comportamiento de temporización de los componentes, o por medio de simulación. Este análisis puede revelar que el sistema no se desempeñará de manera adecuada. La arquitectura del proceso, política de planeación, plataforma de ejecución o todo ello, pueden rediseñarse entonces para mejorar el rendimiento del sistema.

Las restricciones de temporización u otros requerimientos en ocasiones pueden significar que es mejor implementar en el hardware algunas funciones del sistema, tales como el procesamiento de señales, en hardware. Los modernos componentes de hardware, como los FPGA (por las siglas de *Field Programmable Gate Array*), son flexibles, así que pueden adaptarse a diferentes funciones. Los componentes de hardware ofrecen mucho mejor rendimiento que el software equivalente. Los cuellos de botella de procesamiento del sistema pueden identificarse y sustituirse por hardware, de manera que se evita la costosa optimización de software.

20.1.1 Modelado de sistemas de tiempo real

Los eventos a los que un sistema de tiempo real debe reaccionar causan con frecuencia que el sistema se mueva de un estado a otro. Por esta razón, se usan por lo general los modelos de estado, que se exponen en el capítulo 5, para describir los sistemas de tiempo real.

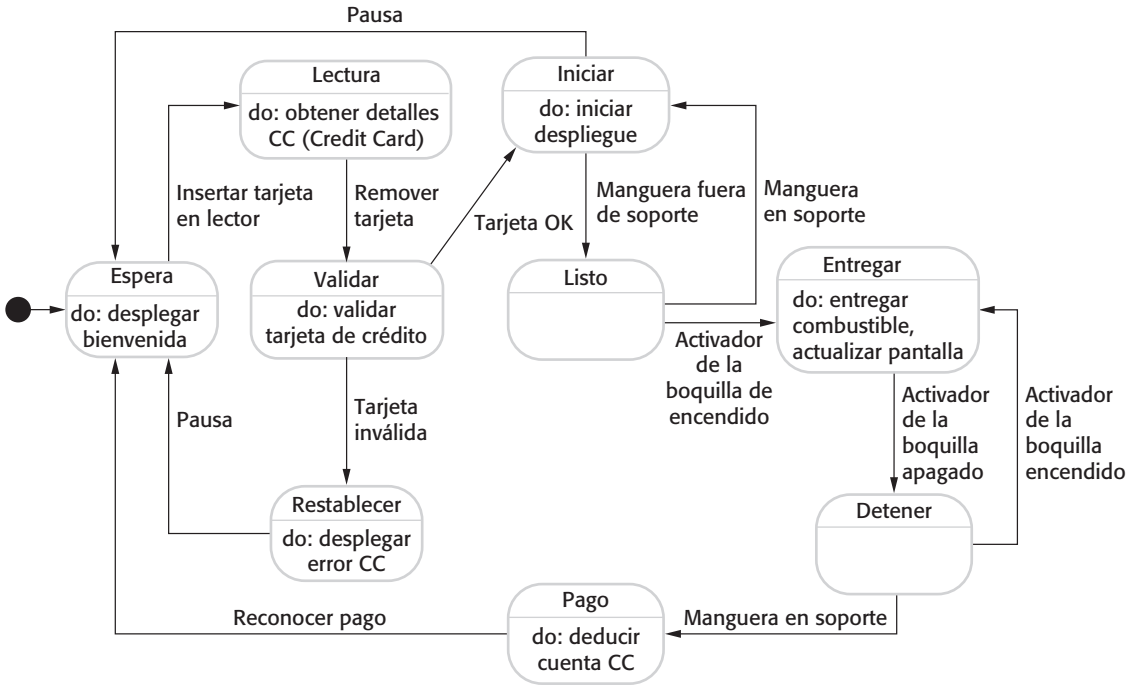


Figura 20.5 Modelo de máquinas de estado de una bomba de petróleo (gasolina)

Un modelo de estado de un sistema supone que, en cualquier momento, el sistema está en uno de ciertos estados posibles. Cuando se recibe un estímulo, esto puede provocar una transición a un estado diferente. Por ejemplo, un sistema que controla una válvula puede moverse desde un estado "Válvula abierta" a un estado "Válvula cerrada" cuando se recibe un comando operador (el estímulo).

Los modelos de estado son una forma independiente de lenguaje para representar el diseño de un sistema de tiempo real y, por lo tanto, son una parte integral de los métodos de diseño de sistemas de tiempo real (Gomaa, 1993). El UML soporta el desarrollo de modelos de estado basados en diagramas de estado (*statecharts*) (Harel, 1987; Harel, 1988). Los diagramas de estado son modelos formales de máquinas de estado que soportan estados jerárquicos, de modo que grupos de estados pueden considerarse como una sola entidad. Douglass analiza el uso del UML en el desarrollo de sistemas de tiempo real (Douglass, 1999). Los modelos de estado se usan en la ingeniería dirigida por modelo, que se estudió en el capítulo 5, para definir la operación de un sistema. Pueden transformarse automáticamente a un programa ejecutable.

Este enfoque ya se ilustró en el modelado de sistemas también en el capítulo 5, en el que se usó un ejemplo de un modelo de un horno de microondas simple. La figura 20.5 es otra muestra de un modelo de máquinas de estado que presenta la operación de un sistema de software embebido para entrega de combustible en una bomba de petróleo (gasolina). Los rectángulos redondeados representan estados del sistema, mientras que las flechas representan estímulos que fuerzan una transición de un estado a otro. Los nombres elegidos en el diagrama de la máquina de estado son descriptivos. La información asociada indica las acciones tomadas por los actuadores del sistema o la información que se despliega. Observe que este sistema nunca termina, pero se encuentra pasivo en un estado de espera cuando la bomba no está funcionando.

El sistema de entrega de combustible está diseñado para permitir el funcionamiento sin atención. El comprador inserta una tarjeta de crédito en un lector de tarjetas colocado en la bomba. Esto provoca una transición a un estado Lectura donde se leen detalles de la tarjeta, y entonces se pide al comprador retirar la tarjeta. Al retirarla se activa una transición hacia un estado Validar donde se valida la tarjeta. Si la tarjeta es válida, el sistema pone en acción la bomba y, cuando la manguera de combustible se retira de su soporte, se traslada hacia el estado Entregar, donde está lista para suministrar el combustible. Al accionar el activador de la boquilla se bombea el combustible; esto se detiene cuando se suelta el activador (por simplicidad, se ignoró el interruptor de presión diseñado para detener el derrame de combustible). Después de completar la entrega de combustible y de que el comprador coloca la manguera en su soporte, el sistema se mueve hacia un estado Pago donde se carga a la cuenta del usuario. Después del pago, el software de la bomba regresa al estado Espera.

20.1.2 Programación en tiempo real

Los lenguajes de programación para el desarrollo de sistemas de tiempo real deben incluir instalaciones para acceder al hardware del sistema, y debe ser factible predecir la temporización de operaciones particulares en dichos lenguajes. Los sistemas de tiempo real duros se programan todavía en ocasiones en lenguaje ensamblador, de modo que pueda cumplirse con los plazos ajustados. También se usan ampliamente los lenguajes a nivel de sistemas, tales como C, que permiten la generación de un código eficiente.

La ventaja de usar un lenguaje de programación de sistemas como C es que permite el desarrollo de programas muy eficientes. Sin embargo, dichos lenguajes no incluyen sentencias para soportar concurrencia o la gestión de recursos compartidos. Concurrencia y gestión de recursos se implementan a través de llamadas a primitivas proporcionadas por el sistema operativo de tiempo real, tales como semáforos para exclusión mutua. Dichas llamadas no pueden probarse por el compilador, así que son más probables los errores de programación. Con frecuencia, los programas también son difíciles de comprender, porque los lenguajes no incluyen características de tiempo real. Además de comprender el programa, el lector debe conocer cómo se brinda el soporte de tiempo real mediante llamadas de sistema.

Puesto que los sistemas de tiempo real deben satisfacer sus restricciones de temporización, no podrán usar desarrollo orientado a objetos para sistemas de tiempo real duros. El desarrollo orientado a objetos implica ocultar representaciones de datos y dar acceso a valores de atributos a través de operaciones definidas con el objeto. Esto quiere decir que hay una significativa carga en rendimiento en los sistemas orientados a objetos, debido a que se requiere código adicional para mediar el acceso a los atributos y manejar las llamadas a las operaciones. La consecuente pérdida de rendimiento puede hacer imposible cumplir con los plazos en tiempo real.

Se diseñó una versión de Java para el desarrollo de sistemas embebidos (Dibble, 2008), con implementaciones de diferentes compañías, como IBM y Sun. Este lenguaje incluye un mecanismo de hilo modificado, que permite especificar hilos que no se interrumpirán por el mecanismo de recolección de basura del lenguaje. También se deben incluir manejo de eventos asíncronos y especificación de temporización. Sin embargo, al momento de escribir el libro, esto se ha usado principalmente en plataformas con representativa capacidad de procesador y de memoria (por ejemplo, un teléfono celular), y no en sistemas embebidos más simples, con recursos más limitados. Estos sistemas generalmente se siguen implementando en C.

Java en tiempo real

El lenguaje de programación Java se modificó de varias formas para adecuarlo al desarrollo de sistemas de tiempo real. Tales modificaciones comprenden comunicaciones asíncronas; la adición de tiempo, incluido tiempo absoluto y relativo; un nuevo modelo de hilo donde los hilos no pueden interrumpirse por la recolección de basura; y un nuevo modelo de gestión de memoria, el cual evita las demoras impredecibles que pueden resultar de la recolección de basura.

http://www.SoftwareEngineering-9.com/Web/RTS/Java.html

20.2 Patrones arquitectónicos

Los patrones arquitectónicos, que se explicaron en el capítulo 6, son descripciones abstractas estilizadas de buenas prácticas de diseño. Contienen conocimiento acerca de la organización de las arquitecturas del sistema, cuándo deben usarse dichas arquitecturas, y sus ventajas y desventajas. Con todo, no se debe pensar en un patrón arquitectónico como si se tratara de un diseño genérico a representarse mediante casos. En vez de ello, el patrón se usa para comprender una arquitectura y, como punto de partida, en la creación de su propio diseño arquitectónico específico.

Como se esperaría, las diferencias entre software embebido e interactivo significan que, para los sistemas embebidos, se utilizan diferentes patrones arquitectónicos, en vez de los patrones arquitectónicos expuestos en el capítulo 6. Los patrones de los sistemas embebidos son orientados a procesos más que orientados a objetos o componentes. En esta sección se analizan tres patrones arquitectónicos de tiempo real que se emplean comúnmente:

1. *Observar y reaccionar* Este patrón se utiliza cuando un conjunto de sensores se monitorizan y despliegan de manera rutinaria. En el momento en que los sensores indican que sucedió cierto evento (por ejemplo, una llamada entrante en un teléfono celular), el sistema reacciona iniciando un proceso para manejar dicho evento.
2. *Control ambiental* Este patrón se emplea cuando un sistema incluye sensores que proporcionan información sobre el entorno y los actuadores que pueden cambiar el entorno. En respuesta a los cambios ambientales detectados por el sensor, se envían señales de control a los actuadores del sistema.
3. *Segmentación de proceso* (*process pipeline*) Este patrón se usa al transformarse datos de una representación a otra antes de que puedan procesarse. La transformación se implementa como una secuencia de pasos de procesamiento, que pueden realizarse de manera concurrente. Esto permite un procesamiento de datos muy rápido, debido a que un núcleo o procesador separado puede ejecutar cada transformación.

Desde luego, los patrones pueden combinarse y usted percibirá con frecuencia más de uno de ellos en un solo sistema. Por ejemplo, cuando se usa el patrón Control Ambiental, es muy común que los actuadores se monitoricen mediante el patrón Observar y Reaccionar. En caso de falla de un actuador, el sistema puede reaccionar desplegando un

Nombre	Observar y reaccionar
Descripción	Se recopilan y analizan los valores de entrada de un conjunto de sensores de los mismos tipos. Dichos valores se despliegan en alguna forma. Si los valores de sensor indican que surgió alguna condición excepcional, entonces se inician acciones para llamar la atención del operador hacia dicho valor y, en ciertos casos, realizar acciones en respuesta al valor excepcional.
Estímulos	Valores de los sensores unidos al sistema.
Respuestas	Salidas a desplegar, activadores de alarma, señales a sistemas que reaccionan.
Procesos	Observador, Análisis, Despliegue, Alarma, Reactor.
Usado en	Sistemas de monitorización, sistemas de alarma.

Figura 20.6 El patrón Observar y Reaccionar

mensaje de advertencia, desactivando el actuador, conmutando a un sistema de respaldo, etcétera.

Los patrones estudiados aquí son patrones arquitectónicos que describen la estructura global de un sistema embebido. Douglass (2002) refiere patrones de diseños de bajo nivel de tiempo real que se usan para ayudar a tomar decisiones más detalladas de diseño. Dichos patrones incluyen patrones de diseño para control de ejecución, comunicaciones, asignación de recursos, y seguridad y fiabilidad.

Estos patrones arquitectónicos deben ser el inicio para un diseño de sistemas embebidos; sin embargo, no son plantillas de diseño. Si se usan como tales, probablemente se terminará con una arquitectura de proceso ineficiente. Por lo tanto, se deberá optimizar la estructura del proceso para garantizar que no tiene demasiados procesos. También se debe probar que existe una correspondencia clara entre los procesos y sensores y los actuadores del sistema.

20.2.1 Observar y reaccionar

Los sistemas de monitorización son una importante clase de sistemas embebidos de tiempo real. Un sistema de monitorización examina su entorno mediante un conjunto de sensores y, por lo general, despliega de alguna forma el estado del entorno. Esto podría ser en una pantalla interna, en paneles de instrumentos de propósito especial o en una pantalla remota. Si el sistema detecta cierto evento o estado de sensor excepcional, el sistema de monitorización toma alguna acción. Con frecuencia, esto implica emitir una alarma para llamar la atención de un operador hacia el evento. En ocasiones, el sistema puede iniciar alguna otra acción preventiva, como desactivar el sistema para protegerlo del daño.

El patrón Observar y Reaccionar (figuras 20.6 y 20.7) es un patrón que se usa comúnmente en los sistemas de monitorización. Se observan los valores de los sensores y,

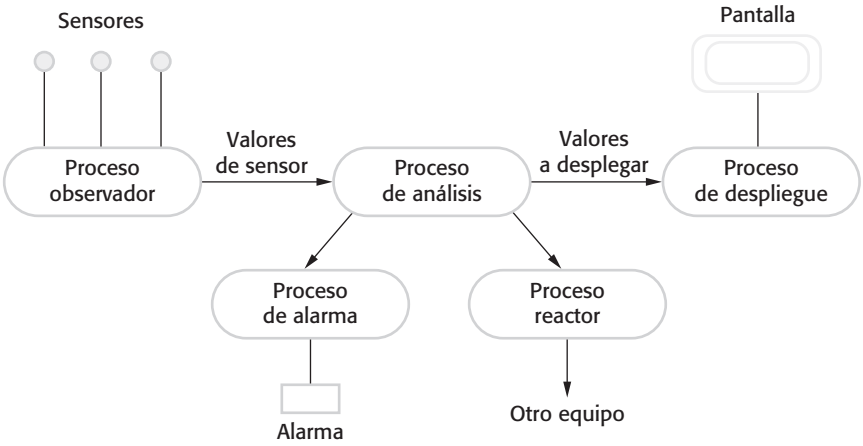


Figura 20.7 Estructura del proceso Observar y Reaccionar

cuando se detectan valores particulares, el sistema reacciona de alguna manera. Los sistemas de monitorización pueden componerse de varias instancias del patrón Observar y Reaccionar, una para cada tipo de sensor en el sistema. Según los requerimientos del sistema, se podrá optimizar el diseño al combinar los procesos (por ejemplo, se puede usar un solo proceso de despliegue para mostrar la información de todos los diferentes tipos de sensores).

Como ejemplo del uso de este patrón, considere el diseño de un sistema de alarma antirrobos que puede instalarse en un edificio de oficinas:

> *Como parte de un sistema de alarma antirrobos, se implementará un sistema de software para edificios comerciales. El sistema usa varios tipos de sensores diferentes, que incluyen: detectores de movimiento en habitaciones individuales, sensores que detectan apertura de puertas del corredor, y sensores en ventanas a nivel del suelo que pueden detectar cuándo se abre alguna de éstas.*

> *Cuando un sensor detecta la presencia de un intruso, el sistema automáticamente llama a la policía local y, mediante un sintetizador de voz, reporta la ubicación de la alarma. Enciende las luces en las habitaciones alrededor del sensor activo y suena una alarma. Por lo general, el sistema de sensor se alimenta del sistema de electricidad principal, pero está equipado con una batería de respaldo. La pérdida de electricidad se detecta con un monitor de circuito eléctrico separado que revisa el voltaje principal. Si detecta una caída de voltaje, el sistema supone que intrusos interrumpieron el suministro eléctrico, de modo que se activa la alarma.*

En la figura 20.8 se muestra una posible arquitectura del proceso para el sistema de alarma. En este diagrama, las flechas representan señales enviadas de un proceso a otro. Este sistema es un sistema de tiempo real "blando" que no tiene requerimientos de temporización rigurosos. Los sensores no necesitan detectar eventos de alta velocidad, por lo que sólo deben consultarse con poca frecuencia. En la sección 20.3 se explican los requerimientos de temporización para este sistema.

En la figura 20.1 se presentaron los estímulos y las respuestas para este sistema de alarma, usados como punto de partida en el diseño del sistema. En este diseño se usa el

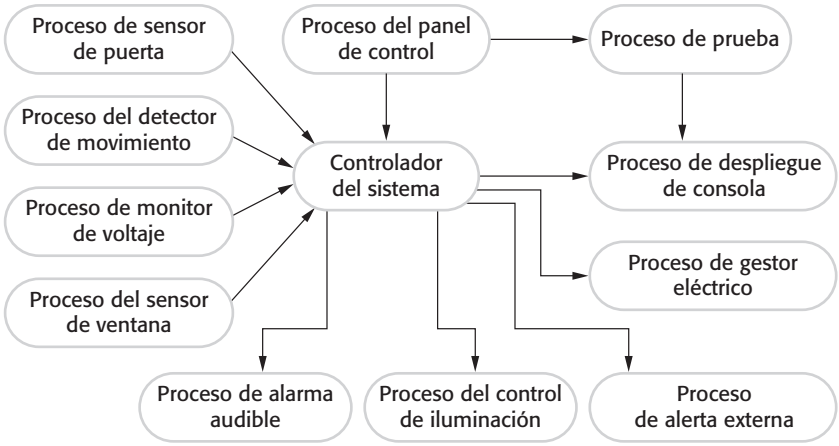


Figura 20.8 Estructura de proceso para un sistema de alarma antirrobos

patrón Observar y Reaccionar. Existen procesos observadores asociados con cada tipo de sensor, y procesos reactor para cada tipo de reacción. Hay un solo proceso de análisis que verifica los datos de todos los sensores. Los procesos de despliegue en el patrón se combinan en un solo proceso de despliegue.

20.2.2 Control ambiental

Tal vez el uso más difundido del software embebido se encuentre en los sistemas de control. En dichos sistemas, el software controla la operación del equipo, con base en estímulos del entorno del equipo. Por ejemplo, un sistema de frenado antiderrapante en un automóvil monitoriza las ruedas y el sistema de frenos del vehículo (el entorno del sistema). Busca signos de que las ruedas derrapan cuando se aplica presión al freno. Si es el caso, el sistema ajusta la presión del freno para evitar el bloqueo de las ruedas y reducir la probabilidad de un deslizamiento.

Los sistemas de control pueden usar el patrón Control Ambiental, que es un patrón de control general que incluye procesos de sensor y actuador. Este patrón se describe en la figura 20.9 con la arquitectura de proceso que se muestra en la figura 20.10. Una variante de este patrón deja fuera el proceso de despliegue. Esta variante se usa en situaciones en que no hay requerimientos para la intervención del usuario o la tasa de control es tan alta que una pantalla no sería significativa.

Este patrón puede ser la base de un diseño de sistema de control con una reformulación del patrón de Control Ambiental para cada actuador (o tipo de actuador) que se controla. Entonces se optimiza el diseño para reducir el número de procesos. Por ejemplo, es posible combinar procesos de monitorización de actuador y de control de actuador, o tener un solo proceso de monitorización y control para varios actuadores. Las optimizaciones que elija dependerán de los requerimientos de temporización. Tal vez se necesite monitorizar sensores con más frecuencia de la que se envían señales de control; en tal caso, será poco práctico combinar procesos de control y monitorización. También puede haber retroalimentación directa entre el control del actuador y el proceso de monitoriza-

Nombre	Control Ambiental
Descripción	El sistema analiza información de un conjunto de sensores que recopilan datos del entorno del sistema. También se puede recopilar más información del estado de los actuadores que se conectan al sistema. Con base en datos de los sensores y actuadores, se envían señales de control a los actuadores, que en ese momento provocan cambios al entorno del sistema. Puede desplegarse información de los valores del sensor y el estado de los actuadores.
Estímulos	Valores de los sensores unidos al sistema y el estado de los actuadores del sistema.
Respuestas	Señales de control a actuadores, despliegue de información.
Procesos	Monitor, de control, de despliegue, controlador de actuador, monitor de actuador.
Usado en	Sistemas de control.

Figura 20.9 El patrón de Control Ambiental

ción del actuador, lo que permite tomar decisiones de control de grano fino mediante el proceso de control del actuador.

En la figura 20.11 se puede ver cómo se usa este patrón; ahí se muestra un ejemplo de un controlador para un sistema de frenado de automóvil. El punto de partida para el diseño es asociar una instancia del patrón con cada tipo de actuador en el sistema. En este caso, hay cuatro actuadores, cada uno de los cuales controla el freno de una rueda. Los procesos de sensor individuales se combinan en un solo proceso de monitorización de rueda que monitoriza los sensores en todas las ruedas. Esto monitoriza el estado de cada

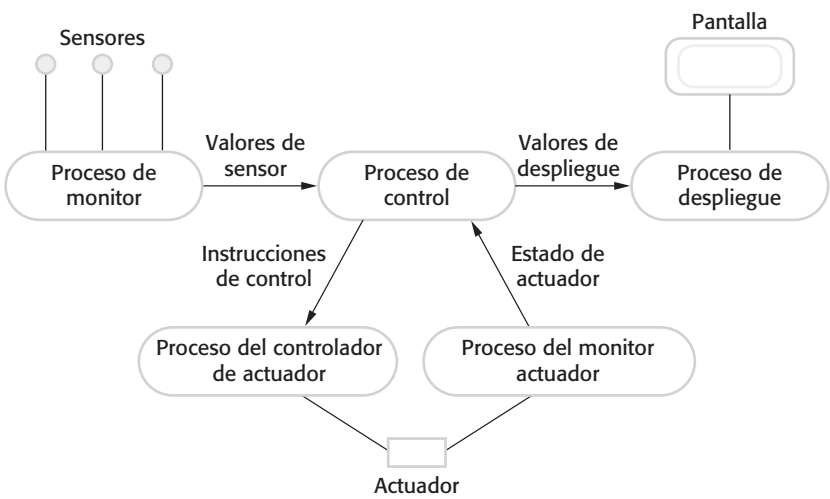


Figura 20.10 Estructura del proceso Control Ambiental

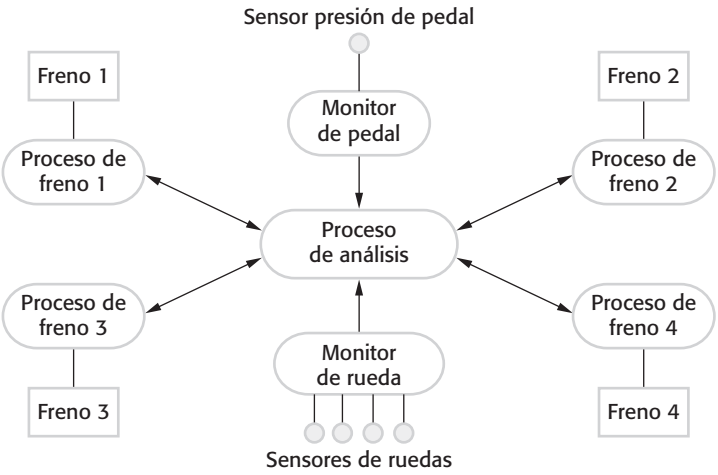


Figura 20.11 Arquitectura de sistema de control para un sistema de frenado antiderrapante

rueda para comprobar si la rueda gira o está bloqueada. Un proceso separado monitoriza la presión en el pedal del freno ejercida por el conductor del automóvil.

El sistema incluye una característica de antiderrapante, que se activa si los sensores indican que una rueda se bloquea al aplicar el freno. Esto significa que existe insuficiente fricción entre el camino y la llanta; en otras palabras, el automóvil derrapa. Si la rueda está bloqueada, el conductor no puede girar el volante. Para contrarrestar esto, el sistema envía una secuencia rápida de señales encendido/apagado (on/off) al freno de dicha rueda, lo que permite que la rueda gire y se recupere el control.

20.2.3 Segmentación de proceso (*process pipeline*)

Muchos sistemas de tiempo real se ocupan de recopilar datos del entorno del sistema, y luego los transforman, de su representación original, a alguna otra representación digital que el sistema pueda analizar y procesar más fácilmente. El sistema también puede convertir los datos digitales a datos analógicos, que entonces se envían a su entorno. Por ejemplo, un software de radio acepta paquetes entrantes de datos digitales que representan la transmisión de radio y los transforman en una señal sonora que puede escuchar la gente.

El procesamiento de datos implicado en muchos de estos sistemas debe realizarse muy rápidamente. De otro modo, los datos entrantes podrían perderse y las señales de salida pueden romperse, por falta de información esencial. El patrón Segmentación de Proceso (*process pipeline*) hace posible este rápido procesamiento, al descomponer el procesamiento de datos requerido en una secuencia de transformaciones separadas, en que cada transformación la realiza un proceso independiente. Ésta es una arquitectura muy eficiente para sistemas que usan procesadores múltiples o procesadores multinúcleo. Cada proceso en la segmentación (*pipeline*, literalmente: tubería) puede asociarse con un procesador o núcleo separado, de modo que los pasos del procesamiento pueden realizarse en paralelo.

Nombre	Segmentación de proceso
Descripción	Una segmentación (*pipeline*) de procesos se establece con datos que se mueven en secuencia de un extremo de la "tubería" a otro. Con frecuencia, los procesos están vinculados mediante buffers sincronizados para permitir que los procesos productor y consumidor se ejecuten a diferentes velocidades. La culminación de una segmentación puede desplegarse o almacenar los datos, o la "tubería" puede terminar en un actuador.
Estímulos	Valores de entrada del entorno o algún otro proceso
Respuestas	Valores de salida al entorno o un buffer compartido
Procesos	Productor, Buffer, Consumidor
Usado en	Sistemas de adquisición de datos, sistemas multimedia

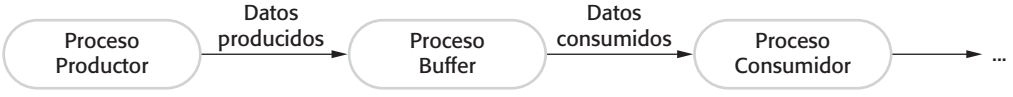
Figura 20.12 El patrón Segmentación de Proceso

La figura 20.12 es una breve descripción del patrón de segmentación de datos, y la figura 20.13 muestra la arquitectura de proceso para este patrón. Observe que los procesos implicados pueden generar y consumir información. Están vinculados mediante buffers sincronizados, como se estudió en la sección 20.1. Esto permite que los procesos productor y consumidor operen a diferentes velocidades sin pérdida de datos.

Un ejemplo de sistema que puede usar una segmentación de proceso es un sistema de adquisición de datos a alta velocidad. Los sistemas de adquisición de datos recolectan datos de los sensores para su posterior procesamiento y análisis. Dichos sistemas se usan en situaciones en que los sensores recolectan muchos datos del entorno del sistema y no es posible o necesario procesar dichos datos en tiempo real. En vez de ello, se recolectan y almacenan para su análisis posterior. Con frecuencia, los sistemas de adquisición de datos se usan en experimentos científicos y sistemas de control de proceso en que los procesos físicos, como las reacciones químicas, son muy rápidos. En estos sistemas, los sensores pueden generar datos muy rápidamente, y el sistema de adquisición de datos debe garantizar que se recopile la lectura de un sensor antes de que cambie el valor del sensor.

La figura 20.14 es un modelo simplificado de un sistema de adquisición de datos que puede ser parte del software de control en un reactor nuclear. Éste es un sistema que recopila datos de sensores que monitorizan el flujo de neutrones (la densidad de neutrones) en el reactor. Los datos del sensor se colocan en un buffer desde el cual se extraen y procesan. El nivel de flujo promedio se muestra en una pantalla del operador y se almacena para procesamiento futuro.

Figura 20.13 Estructura de proceso Segmentación de Proceso



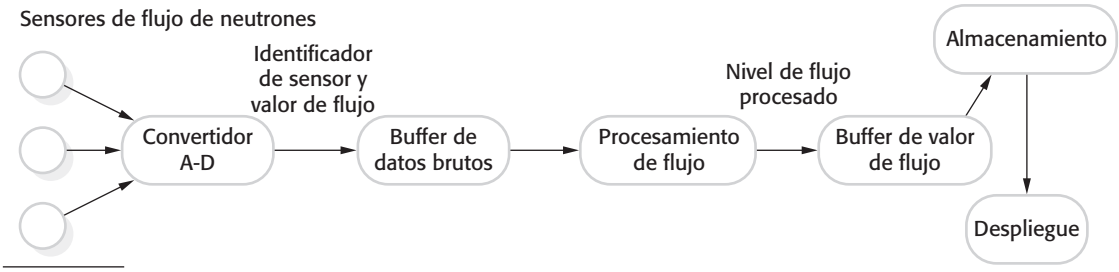


Figura 20.14
Adquisición de datos de flujo de neutrones

20.3 Análisis de temporización

Como se estudió en la introducción, la exactitud de un sistema de tiempo real depende no sólo de la exactitud de sus salidas, sino también del tiempo en que se produjeron dichas salidas. Esto significa que una actividad importante en el proceso de desarrollo de software embebido de tiempo real es el análisis de temporización. En tal análisis, se calcula con qué frecuencia debe ejecutarse cada proceso en el sistema para garantizar que todas las entradas se procesen y que todas las respuestas del sistema se produzcan en forma oportuna. Los resultados del análisis de temporización se usan para decidir con qué continuidad debe ejecutarse cada proceso y cómo el sistema operativo de tiempo real debe organizar temporalmente dichos procesos.

El análisis de temporización (*timing*) para sistemas de tiempo real es particularmente difícil cuando los sistemas deben hacer frente a una mezcla de estímulos y respuestas periódicos y no periódicos. Puesto que los estímulos no periódicos son impredecibles, usted tendrá que hacer suposiciones acerca de la probabilidad de que dichos estímulos ocurran y, por lo tanto, de que se requiera servicio en algún momento particular. Dichos supuestos pueden ser incorrectos y el rendimiento del sistema después de la entrega podría ser inadecuado. El libro de Cooling (2003) examina técnicas para el análisis de rendimiento de los sistemas de tiempo real que toman en cuenta eventos no periódicos.

Sin embargo, conforme las computadoras se vuelven más rápidas ha sido posible, en muchos sistemas, diseñar usando sólo estímulos periódicos. Cuando los procesadores son lentos, deben usarse estímulos no periódicos para garantizar que los eventos críticos se procesen antes de sus plazos, pues las demoras en el procesamiento suponen, por lo general, alguna pérdida al sistema. Por ejemplo, la falla de una fuente de energía en un sistema embebido puede significar que el sistema debe apagar el equipo vinculado en una forma controlada, dentro de un tiempo muy corto (por ejemplo, 50 milisegundos). Esto podría implementarse como una interrupción de "falla de energía". Sin embargo, también puede implementarse usando un proceso periódico que opere muy frecuentemente y compruebe la energía. Siempre y cuando el tiempo entre invocaciones de procesos sea corto, todavía habrá tiempo para realizar un apagado controlado del sistema antes de que la falta de energía cause daños. Por esta razón, se tratarán los conflictos de temporización para procesos periódicos.

Cuando se analizan los requerimientos de temporización de los sistemas embebidos de tiempo real y se diseñan sistemas para cumplir dichos requerimientos, existen tres factores clave que se deben considerar:

1. *Plazos* Los tiempos en que deben procesarse los estímulos y producir alguna respuesta por parte del sistema. Si el sistema no cumple un plazo, entonces, si es un

sistema de tiempo real duro, se trata de una falla de sistema; en un sistema de tiempo real blando, el resultado es un servicio de sistema degradado.

2. *Frecuencia* El número de veces por segundo que debe ejecutarse un proceso para tener la seguridad de que siempre puede cumplir los plazos.

3. *Tiempo de ejecución* El tiempo requerido para procesar un estímulo y producir una respuesta. Con frecuencia se deben tomar en cuenta dos tiempos de ejecución: el tiempo de ejecución promedio de un proceso y el tiempo de ejecución del peor escenario para dicho proceso. No siempre el tiempo de ejecución es el mismo, debido a la ejecución condicional del código, demoras en espera de otros procesos, etcétera. En un sistema de tiempo real duro, tal vez deba hacer suposiciones con base en el tiempo de ejecución del peor escenario para asegurarse de que no venzan los plazos. En los sistemas de tiempo real blandos, quizá deba basar sus cálculos en el tiempo de ejecución promedio.

Para continuar con el ejemplo de una falla en el suministro de energía, suponga que, después de un evento de falla, transcurren 50 ms para que el voltaje suministrado caiga a un nivel en que pueda dañarse el equipo. Por lo tanto, el proceso de apagado del equipo debe comenzar en menos de 50 ms luego de un evento de falla de energía. Ante tales casos, sería prudente establecer un plazo más corto de 40 ms, debido a las variaciones físicas en el equipo. Esto significa que las instrucciones de apagado para todo el equipo conectado que esté en riesgo deben emitirse y procesarse dentro de 40 ms, suponiendo que el equipo también depende de la falla del suministro de energía.

Si usted detecta una falla de energía al monitorizar un nivel de voltaje, debe hacer más de una observación para detectar que cae el voltaje. Si usted ejecuta el proceso 250 veces por segundo, esto significa que ejecuta cada 4 ms y pueden requerirse hasta dos periodos para detectar la caída de voltaje. Por consiguiente, tarda hasta 8 ms detectar el problema. En consecuencia, el tiempo de ejecución del peor escenario del proceso de apagado no debe superar los 16 ms para garantizar que se cumpla el plazo de 40 ms. Esta cifra se calcula restando los periodos de proceso (8 ms) del plazo (40 ms) y dividiendo el resultado entre dos, puesto que se necesitan dos ejecuciones de proceso.

En realidad, por lo general uno se inclinaría hacia un lapso considerablemente menor que 16 ms para dar un margen de seguridad en caso de que los cálculos estuvieran equivocados. De hecho, el tiempo requerido para examinar un sensor y comprobar que no hay pérdida significativa de voltaje debe ser mucho menor que 16 ms. Esto sólo implica una comparación simple de dos valores. El tiempo de ejecución promedio del proceso del monitor eléctrico debe ser menor que 1 ms.

El punto de partida del análisis de temporización en un sistema de tiempo real lo constituyen los requerimientos de temporización, que deben establecer los plazos para cada respuesta requerida en el sistema. La figura 20.15 muestra posibles requerimientos de temporización para el sistema de alarma antirrobo del edificio de oficinas que se estudió en la sección 20.2.1. Para simplificar este ejemplo, ignore los estímulos generados por los procedimientos de prueba del sistema y las señales externas para restablecer el sistema en caso de una falsa alarma. Esto significa que hay sólo dos tipos de estímulo a procesar por parte del sistema:

1. *Falla de energía* Ésta se detecta al observar una caída de voltaje de más del 20 por ciento. La respuesta requerida es encender el circuito de energía de respaldo enviando una señal a un dispositivo electrónico de conmutación de energía, que cambia la electricidad principal a batería de respaldo.

Estímulo/respuesta	Requerimientos de temporización
Falla de energía	El cambio a energía de respaldo debe completarse dentro de un plazo de 50 ms.
Alarma de puerta	Cada alarma de puerta debe revisarse dos veces por segundo.
Alarma de ventana	Cada alarma de ventana debe revisarse dos veces por segundo.
Detector de movimiento	Cada detector de movimiento debe revisarse dos veces por segundo.
Alarma audible	La alarma audible debe encenderse dentro del medio segundo posterior al que un sensor emite una alarma.
Encendido de luces	Las luces deben encenderse dentro del medio segundo posterior al que un sensor emite una alarma.
Comunicaciones	El llamado a la policía debe iniciarse dentro de los 2 segundos posteriores a los que un sensor emite una alarma.
Sintetizador de voz	Un mensaje sintetizado debe estar disponible dentro de los 2 segundos posteriores a los que un sensor emite una alarma.

Figura 20.15 Requerimientos de temporización para el sistema de alarma antirrobo

2. *Alarma contra intrusos* Éste es un estímulo generado por uno de los sensores del sistema. La respuesta a este estímulo es determinar el número de habitación del sensor activo, establecer un llamado a la policía, iniciar el sintetizador de voz para gestionar la llamada, emitir una alarma audible de intrusos y encender las luces en el área.

Como se muestra en la figura 20.15, se deben listar las restricciones de tiempo para cada clase de sensor por separado, aun cuando (como en este caso) sean iguales. Al considerarlas por separado, se deja espacio para cambios futuros y se facilita el cálculo del número de veces que debe ejecutarse por segundo el proceso de control.

Asignar las funciones del sistema a procesos concurrentes es la siguiente etapa del diseño. Existen cuatro tipos de sensores que deben revisarse periódicamente, cada uno con un proceso asociado. Se trata del sensor de voltaje, sensores de puertas, sensores de ventanas y detectores de movimiento. Por lo general, los procesos asociados con el sensor se ejecutarán muy rápidamente, pues todo lo que hacen es comprobar si un sensor cambió o no su estatus (por ejemplo, de apagado a encendido). Es razonable suponer que el tiempo de ejecución para comprobar y valorar el estado de un sensor no es más de 1 ms.

Para garantizar que se cumplen los plazos definidos por los requerimientos de temporización, hay que decidir con qué frecuencia se ejecutan los procesos relacionados y cuántos sensores deben examinarse durante cada ejecución del proceso. Aquí existen ventajas y desventajas obvias entre frecuencia y tiempo de ejecución:

1. Si usted examina un sensor durante cada ejecución de proceso, entonces, si hay N sensores de un tipo particular, debe programar el proceso 4N veces por segundo con la finalidad de garantizar que se cumpla el plazo para detectar un cambio de estado dentro de 0.25 segundos.

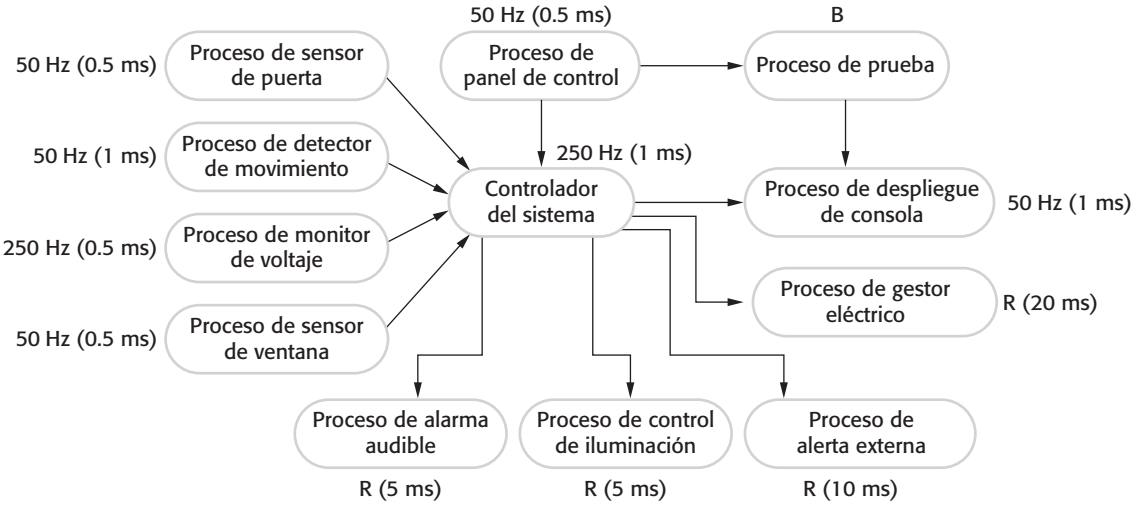


Figura 20.16 Temporización del proceso de alarma

2. Si usted examina cuatro sensores, por ejemplo, durante cada ejecución de proceso, entonces el tiempo de ejecución aumenta a 4 ms, pero sólo necesita ejecutar el proceso N veces/segundo para satisfacer el requerimiento de temporización.

En este caso, puesto que los requerimientos del sistema definen acciones cuando dos o más sensores son positivos, es conveniente examinar sensores en grupos, con grupos basados en la proximidad física de los sensores. Si un intruso entra al edificio, entonces probablemente habrá sensores adyacentes que sean positivos.

Una vez que se ha completado el análisis de temporización, entonces se puede anotar el modelo de proceso con información sobre la frecuencia de ejecución y su tiempo de ejecución esperado (véase como ejemplo la figura 20.16). Aquí, se anotan los procesos periódicos con su frecuencia, los procesos que comienzan en respuesta a un estímulo se anotan con **R**, y el proceso de prueba es un proceso de segundo plano, anotado con **B**. Esto significa que sólo ejecuta cuando está disponible tiempo del procesador. En general, es más sencillo diseñar un sistema de manera que exista un número pequeño de frecuencias de proceso. Los tiempos de ejecución representan los tiempos de ejecución del peor escenario requeridos de los procesos.

El paso final en el proceso de diseño es diseñar un sistema de planeación que garantice que siempre se debe programar un proceso para cumplir con sus plazos. Esto sólo es posible si se conocen los enfoques de planeación que soporta el sistema operativo de tiempo real utilizado (Burns y Wellings, 2009). El planificador en el sistema operativo de tiempo real asigna un proceso a un procesador durante una cantidad de tiempo dada. El tiempo puede fijarse, o variar dependiendo de la prioridad del proceso.

Al asignar prioridades de proceso se deben considerar los plazos de cada uno, de modo que aquellos con plazos cortos reciban tiempo de procesador para cumplir dichos plazos. Por ejemplo, el proceso del monitor de voltaje en la alarma antirrobo necesita programarse de forma que puedan detectarse las caídas de voltaje y se realice un cambio a energía de respaldo antes de que falle el sistema. En consecuencia, esto tiene mayor prioridad que los procesos que comprueban los valores de sensor, pues éstos tienen plazos bastante relajados en comparación con su tiempo de ejecución esperado.

20.4 Sistemas operativos de tiempo real

La plataforma de ejecución para la mayoría de los sistemas de aplicación es un sistema operativo que gestiona recursos compartidos y brinda características tales como un sistema de archivo, gestión de proceso en tiempo de ejecución, etcétera. Sin embargo, la amplia funcionalidad en un sistema operativo convencional toma gran cantidad de espacio y vuelve lenta la ejecución de los programas. Más aún, las características de gestión de proceso en el sistema pueden no estar diseñadas para permitir control de grano fino sobre la planeación de procesos.

Por estas razones, los sistemas operativos estándar, como Linux y Windows, no se usan por lo general como la plataforma de ejecución para sistemas de tiempo real. Los sistemas embebidos muy simples pueden implementarse como sistemas "metal al descubierto". Los sistemas en sí incluyen arranque y apagado de sistema, gestión de proceso y recursos, y planeación de proceso. Sin embargo, más comúnmente, las aplicaciones embebidas se construyen en la parte superior de un sistema operativo de tiempo real (RTOS, por las siglas de *Real-Time Operating System*), el cual es un sistema operativo eficiente que ofrece las características que necesitan los sistemas de tiempo real. Ejemplos de RTOS son Windows/CE, Vxworks y RTLinux.

Un sistema operativo de tiempo real gestiona la asignación de procesos y recursos para un sistema de tiempo real. Inicia y detiene procesos de modo que los estímulos puedan manejarse, y asigna memoria y recursos de procesador. Los componentes de un RTOS (figura 20.17) dependen del tamaño y la complejidad del sistema de tiempo real a desarrollar. Salvo para los sistemas más simples, generalmente incluyen:

1. Un reloj de tiempo real, que proporciona la información requerida para programar los procesos periódicamente.
2. Un manipulador de interrupciones, el cual gestiona peticiones no periódicas de servicios.
3. Un planificador, que es responsable de examinar los procesos que pueden ejecutarse y elegir uno de ellos para su ejecución.
4. Un gestor de recursos, el cual asigna memoria adecuada y recursos de procesador para aquellos procesos que se programaron para su ejecución.
5. Un despachador, que es responsable de iniciar la ejecución de los procesos.

Los sistemas operativos de tiempo real para sistemas grandes, como los sistemas de control de proceso o los de telecomunicaciones, pueden tener instalaciones adicionales, a saber: gestión de almacenamiento de disco, instalaciones para gestión de fallas que detecten y reporten fallas del sistema, y un gestor de configuración que soporte la reconfiguración dinámica de aplicaciones de tiempo real.

20.4.1 Gestión de proceso

Los sistemas de tiempo real tienen que manejar rápidamente los eventos externos y, en algunos casos, cumplir con los plazos para el procesamiento de dichos eventos. Esto

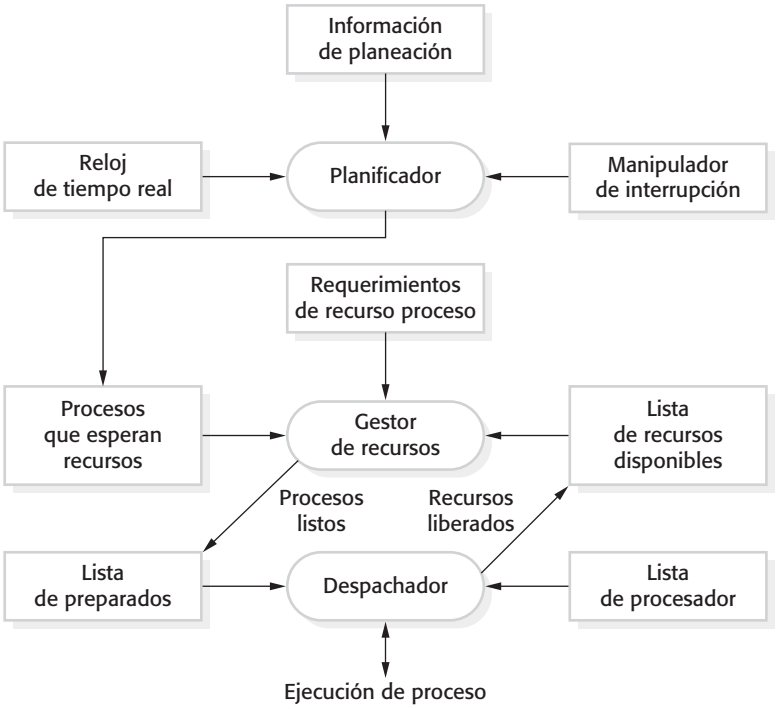


Figura 20.17
Componentes de un sistema operativo de tiempo real

significa que los procesos de manejo de eventos deben programarse para su ejecución a tiempo y detectar el evento. También deben asignar suficientes recursos de procesador para cumplir sus plazos. El gestor de proceso en un RTOS es responsable de elegir los procesos para su ejecución, asignar recursos de procesador y memoria, e iniciar y detener la ejecución del proceso en un procesador.

El gestor de proceso debe tratar los procesos con diferentes prioridades. Para algunos estímulos, como los asociados con ciertos eventos excepcionales, es esencial que su procesamiento deba completarse dentro de los límites de tiempo especificados. Otros procesos pueden demorarse de forma segura si un proceso más crítico requiere servicio. En consecuencia, el RTOS debe gestionar al menos dos niveles de prioridad para los procesos del sistema:

1. *Nivel de interrupción* Éste es el nivel de prioridad más alto. Se asigna a procesos que necesitan una respuesta muy rápida. Uno de dichos procesos será el proceso de reloj de tiempo real.
2. *Nivel de reloj* Este nivel de prioridad se asigna a los procesos periódicos.

Puede haber un nivel de prioridad adicional que se asigna a los procesos en segundo plano (como los procesos de autoverificación) que no necesitan cumplir plazos en tiempo real. Dichos procesos se programan para ejecutarse cuando la capacidad de procesador está disponible.

Dentro de cada uno de dichos niveles de prioridad, a diferentes clases de proceso pueden asignarse distintas prioridades. Por ejemplo, puede haber varias líneas de interrupción.

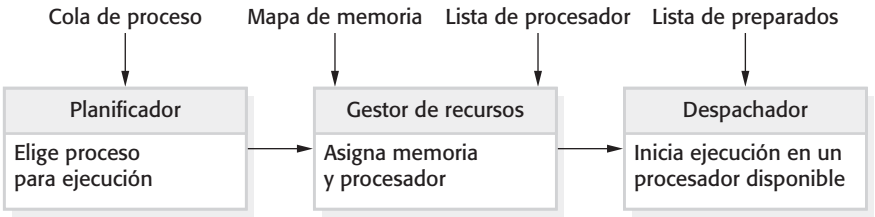


Figura 20.18 Acciones RTOS requeridas para iniciar un proceso

Una interrupción de un dispositivo muy rápido tal vez tenga que invalidar el proceso de una interrupción de parte de un dispositivo más lento para evitar pérdidas de información. La asignación de prioridades de proceso, para que todos los procesos se atiendan a tiempo, requiere por lo general análisis extensos y simulación.

Los procesos periódicos son procesos que deben ejecutarse a intervalos específicos para la adquisición de datos y control de actuador. En la mayoría de los sistemas de tiempo real, habrá muchos tipos de proceso periódico. Al usar los requerimientos de temporización especificados en el programa de aplicación, el RTOS ordena la ejecución de los procesos periódicos para que todos puedan cumplir con sus plazos.

En la figura 20.18 se muestran las acciones tomadas por el sistema operativo para la gestión de procesos periódicos. El planificador examina la lista de procesos periódicos y selecciona un proceso a ejecutar. La elección depende de la prioridad del proceso, periodos del proceso, tiempos de ejecución esperados y plazos de los procesos listos. En ocasiones, dos procesos con diferentes plazos deben ejecutarse al mismo instante del reloj; en tal situación, un proceso debe demorarse. Comúnmente, el sistema elegirá retrasar el proceso con el plazo más largo.

Los procesos que deben responder rápidamente a eventos asíncronos pueden estar dirigidos por interrupciones. El mecanismo de interrupción de la computadora hace que el control se transfiera a una ubicación de memoria predeterminada. Esta ubicación contiene una instrucción para saltar a una rutina de servicio de interrupción simple y rápida. La rutina de servicio deshabilita más interrupciones para evitar interrumpirse a sí misma. Entonces descubre la causa de la interrupción e inicia, con una alta prioridad, un proceso para manejar el estímulo que provocó la interrupción. En algunos sistemas de adquisición de datos de alta velocidad, el manipulador de interrupción guarda los datos que la interrupción señaló como disponibles en un buffer para su procesamiento posterior. Luego, las interrupciones se habilitan nuevamente y el control regresa al sistema operativo.

En cualquier momento puede haber varios procesos, todos con diferentes prioridades, que podrían ejecutarse. El planificador de procesos implementa políticas de planeación del sistema que determinan el orden de ejecución del proceso. Existen dos estrategias de planeación usadas generalmente:

1. *Planeación no sustitutiva* Una vez que se planea un proceso para su ejecución, se ejecuta hasta completarse o bloquearse por alguna razón, como la espera de una entrada. Sin embargo, esto puede causar problemas cuando existen procesos con diferentes prioridades, así que un proceso de alta prioridad debe esperar para que termine un proceso de baja prioridad.

2. *Planeación sustitutiva* Es posible detener la ejecución de un proceso en operación si un proceso de mayor prioridad requiere servicio. El proceso de prioridad más alta sustituye la ejecución del proceso de prioridad más baja y se asigna a un procesador.

Dentro de dichas estrategias se han desarrollado diferentes algoritmos de planeación. En ellos se incluyen planeación circular (*round-robin*), en la que cada proceso se ejecuta en turnos; planeación de tasa monotónica, en la que se otorga prioridad al proceso con el periodo más corto (frecuencia más alta); y planeación prioritaria del plazo más corto, en que se programa el proceso en la cola con el plazo más corto (Burns y Wellings, 2009).

La información acerca del proceso a ejecutar se transmite al gestor de recursos. El gestor de recursos asigna memoria y, en un sistema multiprocesador, también agrega un procesador a este proceso. Entonces el proceso se coloca en la "lista de preparados", una lista de procesos que están preparados para su ejecución. Cuando un procesador termina de ejecutar un proceso y queda disponible, se recurre al despachador. Éste explora la lista de preparados para encontrar un proceso que pueda ejecutarse en el procesador disponible y comienza su ejecución.

PUNTOS CLAVE

- Un sistema de software embebido es parte de un sistema hardware/software que reacciona a eventos en su entorno. El software se "embebe" en el hardware. Los sistemas embebidos, por lo general, son sistemas de tiempo real.
- Un sistema de tiempo real es un sistema de software que debe responder a eventos en tiempo real. La exactitud del sistema no sólo depende de los resultados que produce, sino también del tiempo en que se producen dichos resultados.
- Los sistemas de tiempo real se implementan por lo general como un conjunto de procesos en comunicación que reaccionan ante estímulos para producir respuestas.
- Los modelos de estado son una importante representación de diseño para sistemas embebidos de tiempo real. Se usan para mostrar cómo reacciona el sistema a su entorno conforme los eventos de activación cambian el estado del sistema.
- Existen varios patrones estándar que pueden observarse en diferentes tipos de sistemas embebidos. En ellos se incluye un patrón que monitoriza el entorno del sistema para eventos adversos, un patrón para control de actuador y un patrón de procesamiento de datos.
- Los diseñadores de sistemas de tiempo real tienen que hacer un análisis de temporización (*timing*), que es dirigido por los plazos para procesar los estímulos y responder a ellos. Tienen que decidir con qué frecuencia debe ejecutar cada proceso en el sistema y el tiempo de ejecución esperado y del peor escenario para los procesos.
- Un sistema operativo de tiempo real es responsable de la gestión de los procesos y los recursos. Siempre incluye un planificador, que es el componente responsable de decidir cuál proceso debe programarse para su ejecución.

LECTURAS SUGERIDAS

Software Engineering for Real-Time Systems. Escrito desde una perspectiva de ingeniería más que de las ciencias de la computación, este libro es una excelente guía práctica para la ingeniería de los sistemas de tiempo real. Tiene amplia cobertura de los conflictos de hardware, así que es un acertado complemento al libro de Burns y Wellings (véase la siguiente obra citada). (J. Cooling, Addison-Wesley, 2003.)

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th edition. Un extraordinario y completo texto que brinda una vasta cobertura de todos los aspectos de los sistemas de tiempo real. (A. Burns y A. Wellings, Addison-Wesley, 2009.)

"Trends in Embedded Software Engineering". Este artículo sugiere que el desarrollo orientado por modelo (como se estudió en el capítulo 5 de este libro), se convertirá en un importante enfoque para el desarrollo de sistemas embebidos. Éste es parte de un número especial sobre los sistemas embebidos, y usted también descubrirá que otros artículos son lecturas útiles. (*IEEE Software*, **26** (3), mayo-junio de 2009.) http://dx.doi.org/10.1109/MS.2009.80.

EJERCICIOS

20.1. Con ejemplos, explique por qué los sistemas de tiempo real por lo general tienen que implementarse mediante procesos concurrentes.

20.2. Identifique posibles estímulos y respuestas esperadas para un sistema embebido que controle un refrigerador o una lavadora domésticos.

20.3. Con el enfoque basado en estado para modelado, como se estudió en la sección 20.1.1, modele la operación de un sistema de software embebido para un sistema de correo de voz incluido en una línea telefónica fija. Éste debe mostrar el número de mensajes grabados en una pantalla LED y permitir al usuario marcar números telefónicos y escuchar los mensajes grabados.

20.4. Explique por qué un enfoque orientado a objetos para el desarrollo de software puede no ser adecuado para sistemas de tiempo real.

20.5. Muestre cómo podría usarse el patrón de Control Ambiental como la base del diseño de un sistema para controlar la temperatura en un invernadero. La temperatura debe estar entre 10 y 30 grados Celsius. Si cae por abajo de 10 grados, debe encenderse el sistema de calefacción; si rebasa los 30, deben abrirse automáticamente las ventanas.

20.6. Diseñe una arquitectura de proceso para un sistema de monitorización ambiental que recopile datos de un conjunto de sensores de calidad del aire colocados alrededor de la ciudad. Hay 5,000 sensores organizados en 100 vecindarios. Cada sensor debe consultarse cuatro veces por segundo. Cuando más del 30% de los sensores en un vecindario particular indiquen que la calidad del aire está por abajo de un nivel aceptable, se activan luces de advertencia locales. Todos los sensores envían las lecturas a una computadora central, la cual genera reportes cada 15 minutos sobre la calidad del aire en la ciudad.

Sistema de protección ferroviario

- El sistema adquiere información acerca del límite de velocidad de un segmento desde un transmisor al lado de la vía, que transmite continuamente el identificador del segmento y su límite de velocidad. El mismo transmisor también transfiere información sobre el estatus de la señal que controla el segmento de vía. El tiempo requerido para transmitir el segmento de vía y la información de la señal es de 50 ms.
- El tren puede recibir información del transmisor de la vía cuando está dentro de 10 m.
- La velocidad máxima de un tren es 180 kph.
- Los sensores en el tren ofrecen información acerca de la velocidad actual del tren (actualizada cada 250 ms) y el estatus de frenado del tren (actualizado cada 100 ms).
- Si la velocidad del tren supera en más de 5 kph el límite de velocidad del segmento actual, se emite una advertencia sonora en la cabina del conductor. Si la velocidad del tren supera en más de 10 kph el límite de velocidad del segmento actual, se aplican automáticamente los frenos del tren hasta que disminuye el límite de velocidad del segmento. Los frenos del tren deben aplicarse dentro de los primeros 100 ms a partir del momento en que se detectó la excesiva velocidad del tren.
- Si el tren entra a una vía señalada con una luz roja, el sistema de protección aplica los frenos del tren y reduce la velocidad a cero. Los frenos del tren deben aplicarse dentro de los primeros 100 ms a partir del momento en que se recibió la señal de luz roja.
- El sistema actualiza continuamente una pantalla de estatus en la cabina del conductor.

Figura 20.19
Requerimientos para un sistema de protección ferroviario

20.7. Un sistema de protección ferroviario aplica automáticamente los frenos de un tren si se excede el límite de velocidad para un segmento de vía o si el tren entra en un segmento de vía que actualmente se señala con una luz roja (esto es, no debe entrar en el segmento). En la figura 20.19 se muestran los detalles. Identifique los estímulos que debe procesar el sistema de control a bordo del tren y las respuestas asociadas a dichos estímulos.

20.8. Sugiera una posible arquitectura de proceso para este sistema.

20.9. Si un proceso periódico en el sistema de protección a bordo del tren se usa para recopilar datos del transmisor al lado de la vía, ¿con qué frecuencia debe programarse para asegurar que el sistema garantice la recolección de información del transmisor? Explique cómo llegó a su respuesta.

20.10. ¿Por qué los sistemas operativos de propósito general, como Linux o Windows, no son adecuados como plataformas de sistemas de tiempo real? Considere su experiencia de usar un sistema de propósito general para ayudarse a responder esta pregunta.

REFERENCIAS

Berry, G. (1989). "Real-time programming: Special-purpose or general-purpose languages". In *Information Processing*. Ritter, G. (ed.). Amsterdam: Elsevier Science Publishers, 11–17.

Brinch-Hansen, P. (1973). *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall.

Burns, A. y Wellings, A. (2009). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.

Cooling, J. (2003). *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.

Dibble, P. C. (2008). *Real-time Java Platform Programming, 2nd edition*. Charleston, SC: Booksurge Publishing.

Dijkstra, E. W. (1968). "Cooperating Sequential Processes". En *Programming Languages*. Genuys, F. (ed.). Londres: Academic Press, 43–112.

Douglass, B. P. (1999). *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd edition*. Boston: Addison-Wesley.

Douglass, B. P. (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.

Gomaa, H. (1993). *Software Design Methods for Concurrent and Real-Time Systems*. Reading, Mass.: Addison-Wesley.

Harel, D. (1987). "Statecharts: A Visual Formalism for Complex Systems". *Sci. Comput. Programming,* **8** (3), 231–74.

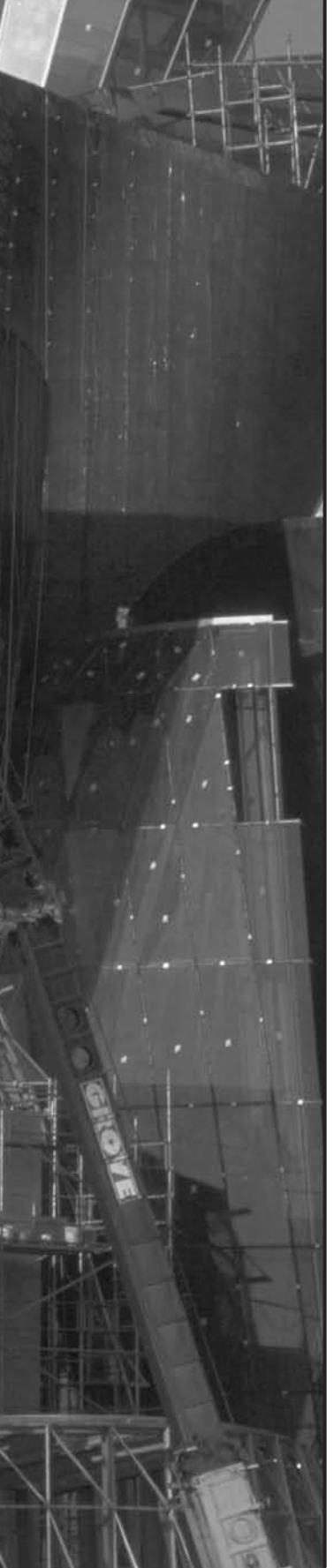
Harel, D. (1988). "On Visual Formalisms". *Comm. ACM*, **31** (5), 514–30.

Hoare, C. A. R. (1974). "Monitors: an operating system structuring concept". *Comm. ACM*, **21** (8), 666–77.

Lee, E. A. (2002). "Embedded Software". En *Advances in Computers*. Zelkowitz, M. (ed.). Londres: Academic Press.

Silberschatz, A., Galvin, P. B. y Gagne, G. (2008). *Operating System Concepts, 8th edition*. New York: John Wiley & Sons.

Tanenbaum, A. S. (2007). *Modern Operating Systems, 3rd edition*. Englewood Cliffs, NJ: Prentice Hall.



21

Ingeniería de software orientada a aspectos

Objetivos

El objetivo de este capítulo es introducirlo al desarrollo de software orientado a aspectos, que se basa en la separación de competencias, intereses, asuntos o propiedades del sistema (*separation of concerns*). Al estudiar este capítulo:

- comprenderá por qué la separación de intereses es un buen principio guía para el desarrollo de software;
- se introducirá a las ideas fundamentales que subyacen en los aspectos y el desarrollo de software orientado a aspectos;
- conocerá cómo puede usar un enfoque orientado a aspectos para la ingeniería de requerimientos, el diseño de software y la programación;
- estará al tanto de las dificultades de poner a prueba sistemas orientados a aspectos.

Contenido

21.1 La separación de intereses
21.2 Aspectos, puntos de enlace y puntos de corte
21.3 Ingeniería de software con aspectos

En la mayoría de los grandes sistemas, las relaciones entre los requerimientos y componentes del programa son complejas. Un solo requerimiento puede implementarse mediante algunos componentes y cada uno de éstos puede incluir elementos de varios requerimientos. En la práctica, esto significa que implementar un cambio a los requerimientos implica comprender y modificar varios componentes. Otra posibilidad es que un componente puede proporcionar alguna funcionalidad central, pero también incluir código que implemente muchos requerimientos de sistema. Aun cuando parezca haber un significativo potencial de reutilización, podría ser costoso reutilizar tales componentes. La reutilización tal vez implique modificarlos para eliminar el código adicional de manera que no se asocie con la funcionalidad central del componente.

La ingeniería de software orientada a aspectos (AOSE, por las siglas de *Aspect-Oriented Software Engineering*) es un enfoque al desarrollo de software que está destinado a enfrentar este problema y así elaborar programas más fáciles de mantener y reutilizar. AOSE se basa en abstracciones llamadas aspectos, que ponen en marcha la funcionalidad de sistema que pueden requerirse en varios lugares diferentes en un programa. Los aspectos encapsulan funcionalidad que atraviesa y coexiste con otra funcionalidad que se incluye en un sistema. Se usan al lado de otras abstracciones como objetos y métodos. Un programa ejecutable orientado a aspectos se crea automáticamente al combinar (tejer, *weave*) objetos, métodos y aspectos, de acuerdo con las especificaciones comprendidas en el código fuente del programa.

Una importante característica de los aspectos es que incluyen una definición sobre dónde deben incluirse en un programa, además del código que implementa la competencia que atraviesa. Puede especificar que el código transversal (*cross-cutting*) debe incluirse antes o después de una llamada de método específico o al acceder a un atributo. En esencia, el aspecto se entrelaza en el programa central para crear un nuevo sistema aumentado.

El beneficio principal de un enfoque orientado a aspectos es que soporta la separación de competencias. Como se explica en la sección 21.1, la separación de competencias en elementos independientes, en vez de incluir diferentes competencias en la misma abstracción lógica, es una buena práctica de la ingeniería de software. Al representar las competencias transversales como aspectos, éstas pueden comprenderse, reutilizarse y modificarse de manera independiente, sin tomar en cuenta dónde se use el código. Por ejemplo, la autenticación de usuario puede representarse como un aspecto que solicite un nombre de usuario y una contraseña. Esto puede integrarse automáticamente en el programa siempre que se requiera autenticación.

Considere que tiene un requerimiento en el que se precise autenticación del usuario antes de realizar cualquier cambio de información personal en una base de datos. Puede describir esto en un aspecto al enunciar que debe incluir un código de autenticación antes de cada solicitud de métodos que actualicen datos personales. Posteriormente, puede ampliar el requerimiento de autenticación a todas las actualizaciones de la base de datos. Esto se implementa fácilmente al modificar el aspecto. Sólo se cambia la definición de dónde debe incorporarse el código de autenticación en el sistema. No tiene que buscar a través del sistema para encontrar todas las incidencias de dichos métodos. Por consiguiente, tendrá menor probabilidad de cometer errores e introducir vulnerabilidades de seguridad accidentales en su programa.

La investigación y el desarrollo en la orientación a aspectos se enfocaron esencialmente en la programación orientada a aspectos. Se han diseñado lenguajes de programación orientados a aspectos, como AspectJ (Colyer y Clement, 2005; Colyer *et al*., 2005; Kiczales *et al*., 2001; Laddad, 2003a; Laddad, 2003b), que extienden la programación orientada

a objetos para incluir aspectos. Las grandes compañías usan la programación orientada a aspectos en sus procesos de producción de software (Colyer y Clement, 2005). Sin embargo, las competencias transversales son igualmente problemáticas en otras etapas del proceso de desarrollo de software. Ahora los investigadores indagan sobre cómo utilizar la orientación a aspectos en la ingeniería de requerimientos de sistema y diseño de sistemas, y la forma de poner a prueba y verificar programas orientados a aspectos.

Aquí se incluye una discusión de AOSE porque su enfoque en la separación de las competencias es una importante manera de pensar y estructurar un sistema de software. Aunque se han implementado algunos sistemas a gran escala mediante un enfoque orientado a aspectos, el uso de aspectos no es todavía parte de la ingeniería de software convencional. Como en todas las nuevas tecnologías, sus defensores se enfocan en los beneficios más que en los problemas y costos. Aunque transcurrirá algún tiempo antes de que la AOSE se emplee de modo rutinario al lado de otros enfoques de la ingeniería de software, es importante la idea de separar las competencias que subyacen en la AOSE. Considerar la separación de las competencias es un buen enfoque general para la ingeniería de software.

En las secciones restantes del capítulo se tratan los conceptos que son parte de la AOSE, y se examinan las ventajas y desventajas de usar un enfoque orientado a aspectos en diferentes etapas del proceso de desarrollo de software. Como la meta del capítulo es ayudar a comprender los conceptos subyacentes en la AOSE, no nos adentraremos en detalles de algún enfoque específico o lenguaje de programación orientado a aspectos.

21.1 La separación de intereses

La separación de competencias o intereses (*concerns*) es un principio clave del diseño e implementación de software. Significa que usted debe organizar su software de modo que cada elemento en el programa (clase, método, procedimiento, etcétera) realice una función y sólo una función. Entonces podrá enfocarse en ese elemento sin considerar los otros elementos en el programa. Es posible comprender cada parte del programa al conocer su competencia, sin necesidad de entender otros elementos. Cuando se requieren cambios, éstos se localizan en un pequeño número de elementos.

La importancia de separar las competencias se reconoció en las etapas iniciales de la ciencia de la computación. Las subrutinas, que encapsulan una unidad de funcionalidad, se inventaron a principios de la década de 1950, y se han diseñado mecanismos subsecuentes de estructuración del programa, tales como procedimientos y clases de objetos, con la finalidad de proporcionar mejores mecanismos para realizar la separación de competencias. Sin embargo, todos estos mecanismos tienen problemas para hacer frente a ciertos tipos de competencias que cruzan otras competencias. Tales competencias transversales no pueden ubicarse mediante mecanismos de estructuración como objetos o funciones. Los aspectos se inventaron para ayudar a gestionar estas competencias transversales.

Aunque en general se acordó que la separación de competencias es una buena práctica de ingeniería de software, resulta muy difícil identificar con exactitud lo que se entiende realmente por competencia. En ocasiones, ésta se define como una noción funcional (es decir, una competencia es algún elemento funcional en un sistema). Como alternativa, puede definirse de manera muy general como “cualquier pieza de interés o el propósito

de un programa", aunque ninguna de estas definiciones es particularmente útil en la práctica. Sin duda, las competencias son más que simplemente elementos funcionales, pero una definición más general es tan vaga, que en realidad resulta inútil.

Según el autor, la mayoría de los intentos por definir las competencias son problemáticos porque tratan de relacionar las competencias a programas. De hecho, como explican Jacobson y Ng (2004), las competencias se consideran en realidad como reflejos de los requerimientos y las prioridades de las partes interesadas en el sistema. El rendimiento del sistema puede ser una competencia porque los usuarios quieren tener una respuesta rápida de un sistema; algunas partes interesadas tal vez se preocupen porque el sistema incluya una particular funcionalidad; las compañías que soportan un sistema pueden inclinarse en que sea fácil de mantener. Por lo tanto, una competencia puede definirse como algo que es de interés o significado para un participante o un grupo de participantes.

Si piensa en las competencias como una forma de organizar los requerimientos, observará por qué es una buena práctica un enfoque de la implementación que separa las competencias en diferentes elementos de programa. Se considera más fácil rastrear las competencias, expresadas como un requerimiento o un conjunto relacionado de requerimientos, a partir de los componentes del programa, que implementar dichas competencias. Si los requerimientos cambian, entonces es evidente la parte del programa que debe cambiar.

Existen varios tipos diferentes de competencias o intereses para el participante:

1. Competencias funcionales, que se relacionan con la funcionalidad específica a incluir en un sistema. Por ejemplo, en un sistema de control ferroviario, una competencia funcional específica consiste en el frenado del tren.
2. Competencias de calidad del servicio, las cuales se relacionan con el comportamiento no funcional de un sistema. Éstas incluyen características como rendimiento, fiabilidad y disponibilidad.
3. Competencias de política, que se relacionan con políticas generales que rigen el uso de un sistema. Las limitaciones políticas incluyen competencias de seguridad y protección, y las competencias relacionadas con las reglas de negocio.
4. Competencias de sistema, las cuales se relacionan con atributos del sistema como un todo, tales como su mantenibilidad o configurabilidad.
5. Competencias organizacionales, que se relacionan con las metas y prioridades de la organización. Entre ellas se incluyen producir un sistema dentro de presupuesto, usar los activos de software existentes y mantener la imagen de la organización.

Las competencias centrales de un sistema son aquellas competencias funcionales relacionadas con su propósito primario. Así, para un sistema de información de pacientes en un hospital, las competencias funcionales centrales son la creación, edición, recuperación y gestión de registros de pacientes. Además de las competencias centrales, los grandes sistemas tienen también competencias funcionales secundarias. Éstas pueden incluir funcionalidad que comparte información con las competencias centrales, o que se requiere para que el sistema pueda cumplir con sus requerimientos no funcionales.

Por ejemplo, considere un sistema que tenga un requerimiento para ofrecer acceso concurrente a un buffer compartido. Un proceso agrega datos al buffer y otro proceso toma datos del mismo buffer. Este buffer compartido es parte de un sistema de adquisición de

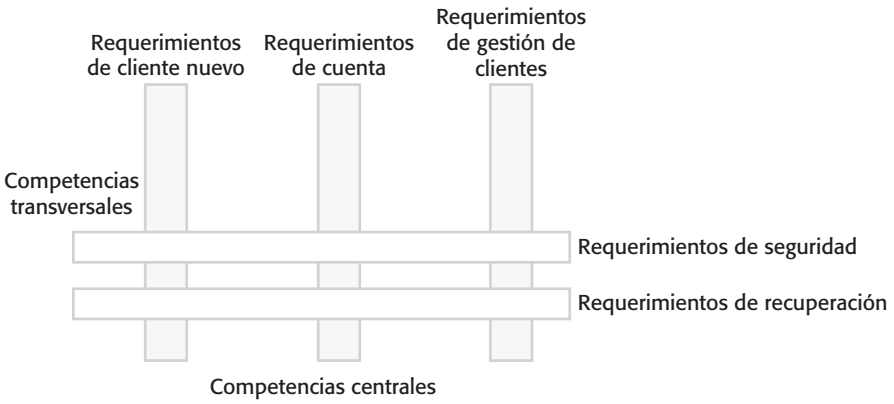


Figura 21.1
Competencias transversales

datos en el que un proceso productor pone datos en el buffer y un proceso consumidor toma los datos éste. Aquí la competencia central es mantener un buffer compartido de manera que la funcionalidad central está asociada con agregar y eliminar elementos del buffer. Sin embargo, para garantizar que los procesos productor y consumidor no interfieran entre sí, existe una competencia secundaria esencial de sincronización. El sistema debe diseñarse de forma que el proceso productor no pueda sobrescribir datos que no se hayan consumido, ni el proceso consumidor pueda tomar datos de un buffer vacío.

Aparte de estas competencias secundarias, otras competencias como calidad de servicio y políticas organizacionales reflejan requerimientos esenciales del sistema. En general, se trata de competencias del sistema: se aplican al sistema como un todo en lugar de implementarse a requerimientos individuales o a la realización de dichos requerimientos en un programa. Son las llamadas competencias transversales (*cross-cutting*), para distinguirlas de las competencias centrales. Las competencias funcionales secundarias también pueden ser transversales, aunque no siempre atraviesan todo el sistema; en vez de ello, se asocian con agrupamientos de competencias centrales que ofrecen funcionalidad relacionada.

Las competencias transversales se muestran en la figura 21.1, la cual se basa en un ejemplo de un sistema de banca por Internet. Este sistema tiene requerimientos relacionados con nuevos clientes, como comprobación de crédito y verificación de dirección. Asimismo, posee requerimientos relacionados con la administración de los clientes existentes y la gestión de cuentas de los clientes. Todas ellas son competencias centrales que se asocian con el propósito primario del sistema: proveer un servicio de banca por Internet. Sin embargo, el sistema también cuenta con requerimientos de seguridad con base en la política de seguridad del banco, y requerimientos de recuperación para garantizar que los datos no se pierdan en caso de una falla del sistema. Se trata de competencias transversales, ya que pueden influir en la implementación de todos los otros requerimientos del sistema.

Las abstracciones de lenguaje de programación, tales como procedimientos y clases, son el mecanismo que se usa normalmente para organizar y estructurar las competencias centrales de un sistema. No obstante, la implementación de las competencias centrales en lenguajes de programación convencionales incluye por lo general código adicional para implementar las competencias transversales, funcionales, de calidad de servicio y de políticas. Esto conduce a dos fenómenos indeseables: enredos (*tangling*) y dispersión (*scattering*).

```
synchronized void put (SensorRecord rec )
{
   // Comprueba que haya espacio en el buffer; espera si no
   if ( numberOfEntries == bufsize)
      wait () ;
   // Agrega registro al final del buffer
   store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
   back = back + 1 ;
   // Si está al final del buffer, la siguiente entrada está al principio
   if (back == bufsize)
      back = 0 ;
   numberOfEntries = numberOfEntries + 1 ;
   // indica que el buffer está disponible
   notify () ;
} // put
```

Figura 21.2 Enredos de código de gestión y sincronización de buffer

El enredo (*tangling*) ocurre cuando un módulo en un sistema incluye código que implementa diferentes requerimientos de sistema. El ejemplo de la figura 21.2, que es una implementación simplificada de parte del código para un sistema de buffer limitado, ilustra este fenómeno. La figura 21.2 es una implementación de la operación put (poner) que añade un ítem al buffer. Sin embargo, si el buffer está lleno, debe esperar hasta que una correspondiente operación get (conseguir) elimine un ítem del buffer. Los detalles no tienen importancia; en esencia, se usan las llamadas wait() y notify() para sincronizar las operaciones put y get. El código que soporta la principal competencia (en este caso, poner un registro en el buffer) está enredado con código que implementa sincronización. El código de sincronización, que se asocia con la competencia secundaria de asegurar exclusión mutua, debe incluirse en todos los métodos que acceden al buffer compartido. El código asociado con la competencia de sincronización se muestra en la figura 21.2.

El fenómeno relacionado de dispersión (*scattering*) ocurre cuando la implementación de una competencia individual (un requerimiento lógico o un conjunto de requerimientos) se dispersa a través de varios componentes del programa. Probablemente esto ocurre cuando se implementan requerimientos relacionados con competencias funcionales secundarias o competencias de política.

Por ejemplo, suponga que un sistema de gestión de registros médicos, como el MHC-PMS, tiene algunos componentes que se ocupan de gestionar la información del personal, los medicamentos, las consultas, las imágenes médicas, los diagnósticos y tratamientos. Es decir, implementan la competencia central del sistema: mantener registros de los pacientes. El sistema puede configurarse para diferentes tipos de clínicas al seleccionar los componentes que proporcionan la funcionalidad necesaria para la clínica.

Sin embargo, suponga que existe también una importante competencia secundaria que es el mantenimiento de información estadística; el proveedor de código de salud quiere registrar detalles de cuántos pacientes se admiten y se dan de alta cada mes, cuántos pacientes mueren, qué medicamentos se prescriben, las razones de las consultas, etcétera. Dichos requerimientos tienen que implementarse agregando código que vuelva anónimos los datos (para mantener la privacidad de los pacientes) y los escriba en una base de datos estadística. Un componente estadístico procesa los datos estadísticos y genera los reportes que se requieren.

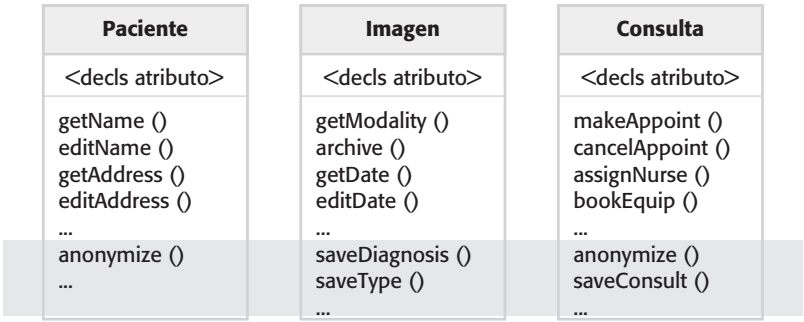


Figura 21.3 Dispersión de métodos que implementan competencias secundarias

Esto se ilustra en la figura 21.3. El diagrama muestra ejemplos de tres clases que pueden incluirse en el sistema de registro de pacientes junto con algunos de los métodos centrales para gestionar información de pacientes. El área sombrada representa los métodos que se requieren para implementar la competencia estadística secundaria. Como se observa, esta competencia estadística se dispersa a lo largo de las otras competencias centrales.

Los problemas con la dispersión y el enredo ocurren cuando cambian los requerimientos iniciales del sistema. Por ejemplo, suponga que deben recopilarse nuevos datos estadísticos en el sistema de registro de pacientes. Los cambios al sistema no se ubican todos en un lugar y, por lo tanto, uno tiene que emplear tiempo buscando los componentes en el sistema que deban cambiarse. Entonces es preciso modificar cada uno de estos componentes para incorporar los cambios requeridos. Esto puede ser costoso debido al tiempo que se necesita para analizar los componentes y, luego, para realizar y probar los cambios. Siempre existe la posibilidad de que se pierda algo de código que se debe cambiar y, por consiguiente, las estadísticas serán incorrectas. Más aún, cuanto más severos sean los cambios que deban realizarse, aumentará la probabilidad de que se cometa una falla y se introduzcan errores en el software.

21.2 Aspectos, puntos de enlace y puntos de corte

En esta sección se introducen los nuevos y más importantes conceptos asociados con el desarrollo de software orientado a aspectos y se ilustran mediante ejemplos del MHC-PMS. La terminología que se utiliza fue introducida por los desarrolladores de AspectJ a finales de la década de 1990. Sin embargo, los conceptos son de aplicación general y no específicos del lenguaje de programación AspectJ. La figura 21.4 resume los términos clave que hay que entender.

Un sistema de registros médicos como el MCH-PMS incluye componentes que manejan información relacionada lógicamente de pacientes. El componente patient (paciente) mantiene información personal acerca del paciente, el componente medication (medicación) conserva información sobre los medicamentos que deben prescribirse, etcétera. Al diseñar el sistema con un enfoque basado en componentes, pueden configurarse diferentes instancias del sistema. Por ejemplo, podría configurarse una versión para cada tipo de clínica en la que sólo se permita a los médicos prescribir medicamentos relevantes para

Término	Definición
consejo (*advice*)	El código que implementa una competencia.
aspecto (*aspect*)	Una abstracción de programa que define una competencia transversal. Incluye la definición de un punto de corte (*pointcut*) y el consejo (*advice*) asociado con dicha competencia.
punto de enlace (*join point*)	Un evento en un programa en ejecución en que puede ejecutarse el consejo asociado con un aspecto.
modelo de punto de enlace (*join point model*)	El conjunto de eventos que se pueden referenciar en un punto de corte.
punto de corte (*pointcut*)	Un enunciado, incluido en un aspecto, que define los puntos de enlace en que debe ejecutarse el consejo del aspecto asociado.
tejido (*weaving*)	La incorporación de código del consejo (*advice*) en los puntos de enlace (*join point*) especificados por un tejedor (*weaving*) de aspectos

Figura 21.4 Terminología usada en la ingeniería de software orientada a aspectos

dicha clínica. Esto simplifica la labor del personal clínico y reduce las posibilidades de que un médico prescriba por error el medicamento equivocado.

Sin embargo, esta organización significa que la información en la base de datos debe actualizarse desde algunos lugares diferentes en el sistema. Por ejemplo, la información del paciente puede modificarse cuando sus datos personales cambian, al prescribirle un nuevo medicamento, al asignarle un nuevo especialista, etcétera. En aras de la sencillez, suponga que todos los componentes del sistema usan una estrategia de nomenclatura consistente y que todas las actualizaciones de la base de datos se implementan mediante métodos que comienzan con "update". Por lo tanto, existen métodos en el sistema como:

```
updatePersonalInformation (patientId, infoupdate)

updateMedication (patientId, medicationupdate)
```

El paciente se identifica mediante patientId y los cambios a realizar se codifican en el segundo parámetro; los detalles de esta codificación no son importantes para este ejemplo. Las actualizaciones se realizan por el personal del hospital, que ingresa al sistema.

Imagine que ocurre una violación de la seguridad y se modifica maliciosamente información del paciente. Tal vez alguien dejó por accidente su computadora conectada al sistema y una persona no autorizada entró al mismo. Alternativamente, una persona autorizada puede tener acceso y modificar maliciosamente la información del paciente. Para reducir la probabilidad de que esto ocurra nuevamente, se introduce una nueva política de seguridad. Antes de realizar cualquier cambio a la base de datos del paciente, la persona que solicita el cambio de nuevo debe autenticarse en el sistema. Los detalles de quién hace el cambio también se registran en un archivo aparte. Esto ayudará a rastrear los problemas si volvieran a ocurrir.

Una forma de implementar esta nueva política es modificar el método update en cada componente para llamar a otros métodos y realizar authentication (autenticación) y

```
aspect authentication
{
   before: call (public void update* (..)) // éste es un punto de corte
   {
      // éste es el consejo -advice- que debe ejecutarse cuando se teje -weaving- en
      // el sistema en ejecución
      int tries = 0 ;
      string  userPassword = Password.Get ( tries ) ;
      while (tries < 3 && userPassword != thisUser.password ( ) )
      {
         // permite 3 intentos para ingresar la contraseña correcta
         tries = tries + 1 ;
         userPassword = Password.Get ( tries ) ;
      }
      if (userPassword != thisUser.password ( )) then
         //si la contraseña es equivocada, supone que el usuario olvidó su contraseña
         System.Logout (thisUser.uid) ;
      }
} // authentication
```

Figura 21.5
Un aspecto de la autenticación (aspect authentication)

logging (conexión lógica). O bien, el sistema podría modificarse de forma que cada vez que se solicite un método update, las llamadas del método se agreguen antes del llamado para hacer authentication, y después para registrar los cambios efectuados. Sin embargo, ninguna de éstas es una excelente solución para este problema:

1. El primer enfoque conduce a una implementación enredada. Lógicamente, actualizar una base de datos, autenticar la fuente de una actualización y registrar los detalles de la actualización son competencias (*concerns*) no relacionadas. Tal vez usted quiera incluir authentication en alguna otra parte del sistema sin ingresar al sistema o desee registrar las acciones separadas de la acción update (actualización). El mismo código authentication y logging debe incluirse dentro de varios métodos diferentes.
2. El enfoque alternativo conduce a una implementación dispersa. Si explícitamente comprende llamadas de método para hacer authentication y logging antes y después de cada llamada a los métodos update, entonces este código se incluye en el sistema en varios lugares diferentes.

Authentication y logging cortan transversalmente las competencias centrales del sistema y posiblemente deban incluirse en varios lugares diferentes. En un sistema orientado a aspectos, es posible representar estas competencias transversales como aspectos separados. Un aspecto incluye una especificación de dónde debe tejerse en el programa la competencia transversal, y el código para implementar dicha competencia. Esto se ilustra en la figura 21.5, que define un aspecto authentication. La notación que se usa en este ejemplo sigue el estilo de AspectJ, pero usa una sintaxis simplificada que debe ser comprensible sin conocimiento de Java o de AspectJ.

Los aspectos son completamente diferentes de otras abstracciones del programa en las que el aspecto en sí incluye una especificación en torno a dónde debe ejecutarse.

Con otras abstracciones, tales como los métodos, hay una separación clara entre la definición de la abstracción y su uso. Al examinar el método, no es posible decir de dónde se le llamará; las llamadas pueden ser desde cualquier parte que abarque el método. En contraste, los aspectos incluyen un "punto de corte" (*pointcut*): un enunciado que define dónde se tejerá el aspecto en el programa.

En este ejemplo, el punto de corte es un enunciado simple:

```
before: call (public void update* (..))
```

El significado de esto es que antes de la ejecución de cualquier método cuyo nombre comience con la cadena update, seguido por cualquier otra secuencia de caracteres, debe ejecutarse el código en el aspecto después de la definición de punto de corte. El carácter asterisco (*) se llama comodín (*wildcard*) y coincide con cualquier cadena de caracteres que se permite en los identificadores. El código a ejecutar se conoce como el "advice" (consejo) y es la implementación de la competencia transversal. En este caso, el consejo obtiene una contraseña de la persona que solicita el cambio y verifica que coincida con la contraseña del usuario que ingresó recientemente al sistema. Si no concuerda, se desconecta al usuario y no procede la actualización.

La habilidad para especificar, con puntos de corte, dónde debe ejecutarse el código es la característica distintiva de los aspectos. Sin embargo, para comprender qué significan los puntos de corte, es necesario entender otro concepto: la idea de un punto de enlace. Un punto de enlace (*join point*) es un evento que ocurre durante la ejecución de un programa; podría ser una llamada de método, la inicialización de una variable, el almacenamiento de una actualización, etcétera.

Existen muchos tipos posibles de eventos que pueden ocurrir durante la ejecución del programa. Un modelo de punto de enlace define el conjunto de eventos que se pueden referenciar en un programa orientado a aspectos. Los modelos de punto de enlace no son estandarizados y cada lenguaje de programación orientado a aspectos tiene su propio modelo de punto de enlace. Por ejemplo, en los eventos AspectJ, que son parte del modelo de punto de enlace, se incluyen:

- eventos de llamada: llamadas a un método o constructor;
- eventos de ejecución: ejecución de un método o constructor;
- eventos de inicialización: inicialización de clase u objeto;
- eventos de datos: acceso o actualización de un archivo;
- eventos de excepciones: manejo de una excepción.

Un punto de corte identifica el evento o eventos específicos (por ejemplo, una llamada a un procedimiento nombrado) con los que debe asociarse un consejo. Esto significa que es posible tejer consejos en un programa en muchos contextos diferentes, dependiendo del modelo de punto de enlace que soporte:

1. Los consejos pueden incluirse antes de la ejecución de un método específico, una lista de métodos nombrados o una lista de métodos cuyos nombres coincidan con una especificación de patrón (como update*).

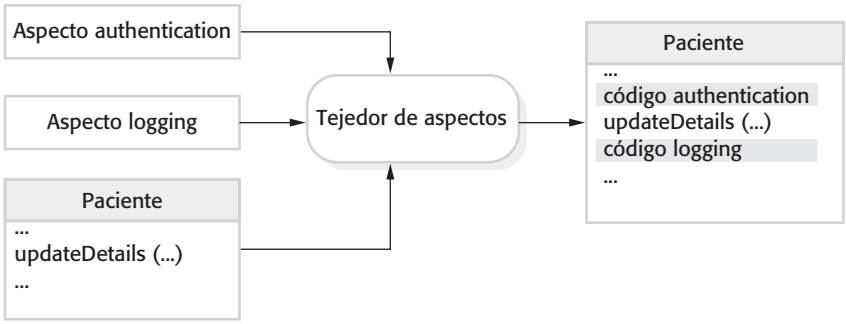


Figura 21.6 Tejido de aspectos

2. Los consejos pueden incluirse después de un regreso normal o excepcional de un método. En el ejemplo que se muestra en la figura 21.5, se podría definir un punto de corte que ejecute el código logging después de todas las llamadas a métodos update.

3. Los consejos pueden incluirse cuando se modifica un archivo en un objeto; se podrían incluir consejos para monitorizar o cambiar ese archivo.

La inclusión de consejos en los puntos de enlace especificados en los puntos de corte es responsabilidad de un tejedor de aspectos. Los tejedores de aspectos son extensiones de compiladores que procesan la definición de los aspectos y las clases de objetos y métodos que definen el sistema. El tejedor genera un nuevo programa con los aspectos incluidos en los puntos de enlace especificados. Los aspectos se integran de forma que las competencias transversales se ejecuten en los lugares correctos del sistema final.

La figura 21.6 ilustra este tejido de aspectos para los aspectos authentication y logging que deben incluirse en el MHC-PMS. Hay tres enfoques diferentes al tejido de aspectos:

1. Preprocesamiento de código fuente, en el que un tejedor toma entrada de código fuente y genera nuevo código fuente en un lenguaje como Java o C++, que luego pueden compilarse usando el compilador de lenguaje estándar. Este enfoque se adoptó para el lenguaje AspectX con su asociado XWeaver (Birrer *et al*., 2005).

2. Tejido de tiempo de vinculación, en el que el compilador se modifica para incluir un tejedor de aspectos. Un lenguaje orientado a aspectos, como AspectJ, se procesa y genera bytecode Java estándar. Entonces éste puede ejecutarse directamente mediante un intérprete Java o procesarse aún más para generar código de máquina nativo.

3. Tejido dinámico en tiempo de ejecución. En este caso, se monitorizan los puntos de enlace y, cuando ocurre un evento referenciado en un punto de corte, se integra el consejo correspondiente con el programa en ejecución.

El enfoque usado más comúnmente para el tejido de aspectos es el tejido de tiempo de vinculación, pues esto permite la implementación eficiente de los aspectos sin una gran carga de tiempo de ejecución. El tejido dinámico es el enfoque más flexible, pero puede incurrir en significativas penalizaciones de rendimiento durante la ejecución del programa. Actualmente el preprocesamiento de código fuente se usa poco.

21.3 Ingeniería de software con aspectos

Los aspectos se introdujeron originalmente con un lenguaje de programación de secuencias, pero, como se estudió, la noción de competencias es una que realmente proviene de los requerimientos del sistema. Por lo tanto, tiene sentido adoptar un enfoque orientado a aspectos en todas las etapas del proceso de desarrollo del sistema. En las primeras etapas de la ingeniería de software, adoptar un enfoque orientado a aspectos significa usar la noción de separación de competencias como base para considerar los requerimientos y el diseño del sistema. Identificar y modelar las competencias debe ser parte de la ingeniería de requerimientos y de diseño. Los lenguajes de programación orientados a aspectos, pues, ofrecen el soporte tecnológico para mantener la separación de las competencias en su implementación del sistema.

Cuando se diseña un sistema, Jacobson y Ng (2004) sugieren que debe considerarse que el sistema soporte diferentes competencias de las partes interesadas como un sistema central más extensiones. Esto se ilustra en la figura 21.7, donde se usan paquetes UML para representar tanto el núcleo como las extensiones. El sistema central es un conjunto de funciones del sistema que implementan el propósito esencial del sistema. Por consiguiente, si el propósito de un sistema particular es mantener información de los pacientes en un hospital, el sistema central en tal caso ofrece un medio para crear, editar, gestionar y acceder a una base de datos de registros de pacientes. Las extensiones del sistema central reflejan competencias adicionales de las partes interesadas, que deben integrarse con el sistema central. Por ejemplo, es importante que un sistema de información médica mantenga la confidencialidad de la información de los pacientes, de manera que una extensión podría ocuparse del control del acceso, otra de la encriptación, etcétera.

Existen algunos tipos diferentes de extensiones que se derivan de los distintos tipos de competencias que se estudiaron en la sección 21.1.

1. *Extensiones funcionales secundarias* Agregan capacidades adicionales a la funcionalidad que ofrece el sistema central. En este caso, con el ejemplo del MHC-PMS, la producción de informes sobre los medicamentos prescritos en el mes anterior sería una extensión funcional secundaria para un sistema de información de pacientes.

2. *Extensiones de política* Agregan capacidades funcionales para soportar políticas de la organización. Las extensiones que adicionan características de seguridad son ejemplos de extensiones de política.

3. *Extensiones QoS* Agregan capacidades funcionales para ayudar a alcanzar los requerimientos de calidad del servicio que se especificaron para el sistema. Por ejemplo, una extensión podría implementar una caché para reducir el número de accesos a la base de datos o automatizar los respaldos para recuperación en caso de una falla del sistema.

4. *Extensiones de infraestructura* Estas extensiones agregan capacidades funcionales para soportar la implementación de un sistema en alguna plataforma de implementación específica. Por ejemplo, en un sistema de información de pacientes, pueden usarse extensiones de infraestructura para implementar la interfaz al sistema de gestión de base de datos subyacente. Pueden hacerse cambios a esta interfaz modificando las extensiones de infraestructura asociadas.

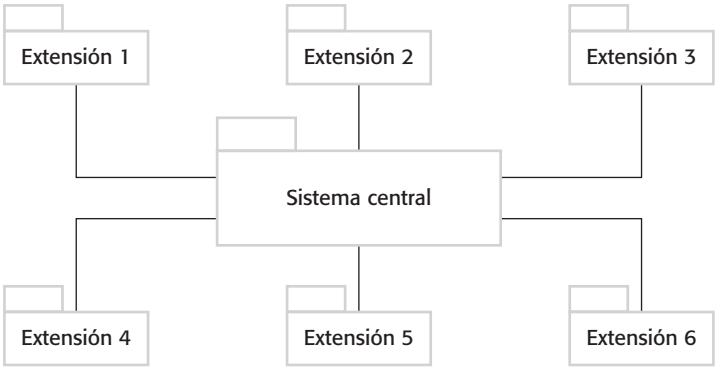


Figura 21.7 Sistema central con extensiones

Las extensiones siempre agregan algún tipo de funcionalidad o características adicionales al sistema central. Los aspectos son una forma de implementar dichas extensiones y pueden combinarse con la funcionalidad del sistema central mediante las instalaciones de tejido en el entorno de programación orientado a aspectos.

21.3.1 Ingeniería de requerimientos orientados a competencias

Como se sugirió en la sección 21.1, las competencias reflejan los requerimientos de las partes interesadas. Dichas competencias pueden reflejar la funcionalidad requerida por un participante, la calidad de servicio del sistema, políticas o conflictos de la organización que se relacionan con los atributos del sistema como un todo. Por lo tanto, tiene sentido adoptar un enfoque a la ingeniería de requerimientos que identifique y especifique las diferentes competencias de las partes interesadas. A veces se usa el término "aspectos tempranos" para referirse al uso de aspectos en etapas tempranas del ciclo de vida del software donde se enfatiza la separación de competencias.

Por muchos años se ha reconocido la importancia de separar las competencias durante la ingeniería de requerimientos. Los puntos de vista que representan diferentes perspectivas sistémicas se incorporan en algunos métodos de ingeniería de requerimientos (Easterbrook y Nuseibeh, 1996; Finkelstein *et al*., 1992; Kotonya y Sommerville, 1996). Estos métodos separan las competencias de diferentes partes interesadas. Los puntos de vista reflejan la distinta funcionalidad que requieren diferentes grupos de participantes.

Sin embargo, también existen requerimientos que atraviesan todos los puntos de vista, como se muestra en la figura 21.8. Este diagrama indica que los puntos de vista pueden ser de diferentes tipos, pero las competencias transversales (como regulación, confiabilidad y seguridad) generan requerimientos que pueden tener repercusiones en todos los puntos de vista del sistema. Ésta fue la principal consideración que hizo el autor en el trabajo para desarrollar el método PreView (Sommerville y Sawyer, 1997; Sommerville *et al*., 1998), que incluía pasos para identificar competencias transversales no funcionales.

Para desarrollar un sistema que esté organizado en el estilo que se presenta en la figura 21.7, hay que identificar los requerimientos para el sistema central además de los requerimientos para las extensiones del sistema. Un enfoque a la ingeniería de requerimientos orientado a puntos de vista, en el que cada punto de vista representa los requerimientos de grupos de participantes relacionados, es una forma de separar competencias centrales y

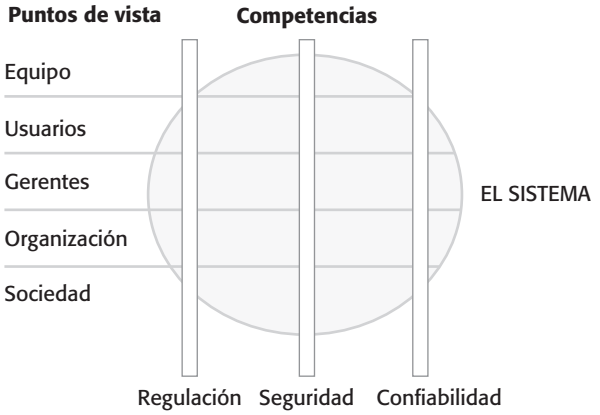


Figura 21.8 Puntos de vista y competencias

secundarias. Si los requerimientos se organizan de acuerdo con el punto de vista de las partes interesadas, entonces usted podrá analizarlos para descubrir requerimientos relacionados que aparezcan en todos o la mayoría de los puntos de vista. Éstos representan la funcionalidad central del sistema. Otros requerimientos de puntos de vista pueden ser requerimientos que sean específicos a dicho punto de vista. Éstos pueden implementarse como extensiones a la funcionalidad central.

Por ejemplo, imagine que usted desarrolla un sistema de software para rastrear equipo especializado que usan los servicios de emergencia. El equipo se ubica en diferentes lugares de una región o estado y, en caso de una emergencia, como una inundación o un terremoto, los servicios de emergencia emplean el sistema para descubrir qué equipo está disponible cerca del sitio del problema. La figura 21.9 muestra bosquejos de requerimientos de tres posibles puntos de vista para tal sistema.

En este ejemplo se observa que los participantes que representan los diferentes puntos de vista deben encontrar ítems de equipo específicos, navegar el equipo disponible en cada ubicación, e ingresar/sacar equipo del almacén. Por lo tanto, éstos son requerimientos para el sistema central. Los requerimientos secundarios soportan las necesidades más específicas de cada punto de vista. Existen requerimientos secundarios para extensiones del sistema que soportan el uso, la gestión y el mantenimiento del equipo.

Los requerimientos funcionales secundarios que se identifican a partir de cualquier punto de vista no atraviesan necesariamente los requerimientos de otros puntos de vista. Por ejemplo, sólo el punto de vista de mantenimiento está interesado por completar registros de mantenimiento. Dichos requerimientos reflejan las necesidades de ese punto de vista y tales competencias tal vez no sean compartidas por otros puntos de vista. Sin embargo, además de los requerimientos funcionales secundarios, existen competencias transversales que generan requerimientos de importancia para algunos o todos los puntos de vista. Con frecuencia, éstos reflejan requerimientos de política y calidad de servicio que se aplican al sistema como un todo. Según se estudió en el capítulo 4, se trata de requerimientos no funcionales, como los requerimientos de seguridad, rendimiento y costo.

En el sistema de inventario de equipo, un ejemplo de competencia transversal es la disponibilidad del sistema. Las emergencias pueden ocurrir con poca o ninguna advertencia. El salvar vidas puede requerir que el equipo especial se despliegue tan rápido como sea posible. Por lo tanto, los requerimientos de confiabilidad para el sistema de inventario

1. Usuarios del servicio de emergencia

1.1 Encontrar un tipo específico de equipo (por ejemplo, un elevador de cargas pesadas).
1.2 Ver equipo disponible en un almacén específico.
1.3 Retirar equipo.
1.4 Ingresar equipo.
1.5 Ordenar equipo para transportarlo a una emergencia.
1.6 Enviar reporte de daño.
1.7 Encontrar un almacén cerca de la emergencia.

2. Planificadores de emergencias

2.1 Encontrar un tipo específico de equipo.
2.2 Ver equipo disponible en una ubicación específica.
2.3 Ingresar/retirar equipo de un almacén.
2.4 Mover el equipo de un almacén a otro.
2.5 Ordenar nuevo equipo.

3. Personal de mantenimiento

3.1 Ingresar/retirar equipo para mantenimiento.
3.2 Ver equipo disponible en cada almacén.
3.3 Encontrar un tipo específico de equipo.
3.4 Ver calendario de mantenimiento para un ítem de equipo.
3.5 Completar registro de mantenimiento para un ítem de equipo.
3.6 Mostrar todos los ítems en un almacén que requieran mantenimiento.

Figura 21.9 Puntos de vista en un sistema de inventario de equipo

de equipo incluyen requerimientos para un alto nivel de disponibilidad del sistema. En la figura 21.10 se muestran algunos ejemplos de estos requerimientos de confiabilidad, con las razones asociadas. Al usar estos requerimientos es posible identificar las extensiones de la funcionalidad central para logging de transacción y reporte de estatus. Esto facilita la identificación de problemas y el cambio a un sistema de respaldo.

El resultado del proceso de ingeniería de requerimientos debe ser un conjunto de requerimientos que se estructuran en torno a la noción de un sistema central más extensiones. Por ejemplo, en el sistema de inventario, los ejemplos de requerimientos centrales pueden ser:

C.1 El sistema debe permitir que los usuarios autorizados vean la descripción de algún ítem de equipo en el inventario de servicios de emergencia.

Figura 21.10 Requerimientos relacionados con la disponibilidad para el sistema de inventario de equipo

DISP.1	Debe existir un sistema "de espera caliente" (*hot standby*) en una ubicación que esté geográficamente bien separada del sistema principal.
Razón:	La emergencia puede afectar la ubicación principal del sistema.
DISP.1.1	Todas las transacciones deben registrarse en el sitio del sistema principal y en el sitio de espera remoto.
Razón:	Esto permite que dichas transacciones se reproduzcan y que las bases de datos del sistema sean consistentes.
DISP.1.2	Cada cinco minutos el sistema debe enviar información de estatus al sistema de la sala de control de emergencias.
Razón:	Los operadores del sistema de la sala de control pueden cambiar a la posición de espera caliente cuando el sistema principal no esté disponible.

Puntos de vista

La noción de puntos de vista se introdujo en el capítulo 4, donde se explicó el modo en que pueden usarse los puntos de vista como una forma de estructurar los requerimientos de diferentes partes interesadas. Al usar puntos de vista, es posible identificar los requerimientos del sistema central de cada agrupamiento de participantes.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

C.2 El sistema debe incluir una instalación de búsqueda para permitir a los usuarios autorizados indagar inventarios individuales o el inventario completo de un ítem específico de equipo o un tipo específico de equipo.

El sistema también puede incluir una extensión que tenga el propósito de apoyar la procuración y sustitución de equipo. Los requerimientos para esta extensión pueden ser:

E1.1 Debe ser posible que usuarios autorizados realicen pedidos a proveedores acreditados para sustituir los ítems de equipo.

E1.1.1 Cuando se ordene un ítem de equipo, debe asignarse a un inventario específico y marcarlo en dicho inventario como "en pedido".

Como regla general, hay que evitar tener demasiadas competencias o extensiones al sistema. Esto simplemente confunde al lector y podría conducir a un diseño prematuro. Además, limita la libertad de los diseñadores y es posible que dé por resultado un diseño de sistema que no cumpla sus requerimientos de calidad de servicio.

21.3.2 Diseño y programación orientada a aspectos

El diseño orientado a aspectos es el proceso de diseñar un sistema que utilice los aspectos para implementar las competencias transversales y extensiones que se identificaron durante el proceso de ingeniería de requerimientos. En esta etapa es necesario traducir las competencias que se relacionan con el problema que hay que resolver a los aspectos correspondientes en el programa que implementará la solución. También es indispensable comprender cómo se combinarán estos aspectos con otros componentes de sistema y garantizar que no surjan ambigüedades de composición.

El enunciado de requerimientos de alto nivel brinda una base para identificar algunas extensiones del sistema que puedan implementarse como aspectos. Entonces es necesario desarrollarlos con mayor detalle para identificar más extensiones y comprender la funcionalidad solicitada. Una forma de hacerlo es identificar un conjunto de casos de uso (que se explicaron en los capítulos 4 y 5) asociados con cada punto de vista. Los modelos de caso de uso se enfocan en la interacción y son más detallados que los requerimientos de usuario. Puede considerarlos como un puente entre los requerimientos y el diseño. En

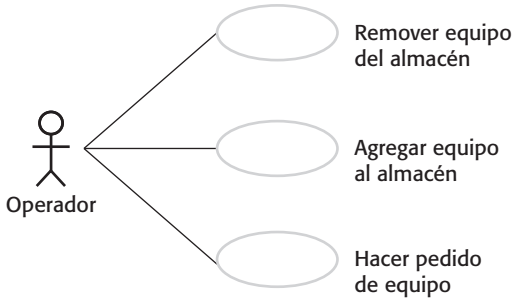


Figura 21.11 Casos de uso del sistema de gestión de inventario

un modelo de caso de uso se describen los pasos de cada interacción de usuario y así se comienza a identificar y definir las clases en el sistema.

Jacobson y Ng (2004) escribieron un libro que explica cómo pueden usarse los casos de uso en la ingeniería de software orientada a aspectos. Sugieren que cada caso de uso representa un aspecto, y proponen extensiones al enfoque de caso de uso para apoyar los puntos de enlace y puntos de corte. También introducen la noción de rebanadas de caso de uso y módulos de caso de uso. Ellos incluyen fragmentos de clases que implementan un aspecto. Pueden combinarse para crear el sistema completo.

La figura 21.11 muestra ejemplos de tres casos de uso que pueden ser parte del sistema de gestión de inventario. Éstos reflejan las competencias de agregar equipo a un inventario y ordenar equipo. La solicitud de equipo y la adición de equipo son competencias relacionadas. Una vez entregados los artículos solicitados, deben agregarse al inventario y entregarse a uno de los almacenes de equipo.

El UML incluye ya la noción de casos de uso de extensión. Un caso de uso de extensión extiende la funcionalidad hacia otro caso de uso. La figura 21.12 indica cómo la colocación de una petición de equipo extiende el caso de uso central para agregar equipo a un almacén específico. Si el equipo a agregar no existe, puede solicitarse y agregarse al almacén cuando se entregue el equipo. Durante el desarrollo de los modelos de caso de uso habrá que buscar las características comunes y, donde sea posible, estructurar los casos de uso como casos centrales más extensiones. Las características transversales, como el registro lógico de todas las transacciones, también pueden representarse como casos de uso de extensión. Jacobson y Ng discuten cómo las extensiones de este tipo pueden implementarse como aspectos.

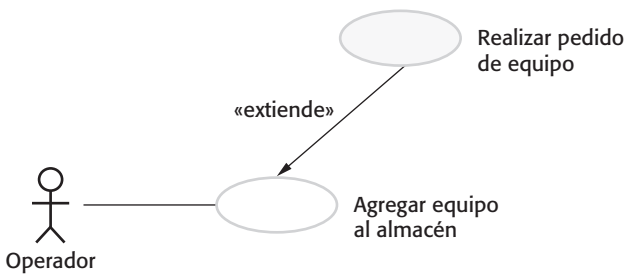


Figura 21.12 Casos de uso de extensión

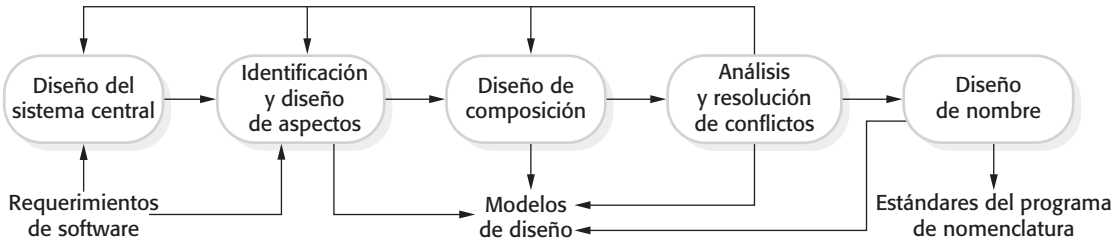


Figura 21.13 Proceso de diseño genérico orientado a aspectos

El desarrollo de un proceso efectivo para el diseño orientado a aspectos resulta esencial si el diseño orientado a aspectos debe aceptarse y usarse. Se sugiere que un proceso de diseño orientado a aspectos incluya las actividades que se muestran en la figura 21.13. Dichas actividades son:

1. *Diseño del sistema central* En esta etapa se diseña la arquitectura del sistema para soportar la funcionalidad central del sistema. La arquitectura debe considerar también los requerimientos de calidad de servicio, como los requerimientos de rendimiento y confiabilidad.
2. *Identificación y diseño de aspectos* A partir de las extensiones identificadas en los requerimientos del sistema, habrá que efectuar un análisis para ver si son aspectos en sí mismos o si deben descomponerse en varios aspectos. Una vez identificados los aspectos, pueden diseñarse por separado tomando en cuenta el diseño de las características del sistema central.
3. *Diseño de composición* En esta etapa se analiza el sistema central y los diseños de aspectos para descubrir dónde deben combinarse los aspectos con el sistema central. En esencia, se identifican los puntos de enlace en un programa donde se tejerán los aspectos.
4. *Análisis y resolución de conflictos* Un problema con los aspectos es que pueden interferir entre sí cuando se combinan con el sistema central. Los conflictos ocurren al haber un choque de puntos de corte con diferentes aspectos que especifican que deben combinarse en el mismo punto del programa. Sin embargo, puede haber conflictos más sutiles. Cuando los aspectos se diseñan de manera independiente, realizan suposiciones acerca de la funcionalidad del sistema central que debe modificarse. No obstante, cuando se combinan varios aspectos, un aspecto podría afectar la funcionalidad del sistema de una forma no anticipada por otros aspectos. Entonces el comportamiento global del sistema quizá no sea el esperado.
5. *Diseño de nombre* Ésta es una importante actividad de diseño que define estándares para nombrar las entidades del programa. Esto es importante para evitar el problema de puntos de corte accidentales. Éstos ocurren cuando, en algún punto de enlace del programa, el nombre coincide accidentalmente con el de un patrón de punto de corte. En consecuencia, el consejo se aplica de manera no intencional a dicho punto. Desde luego, esto es indeseable y puede conducir a un comportamiento inesperado del programa. Por lo tanto, habrá que diseñar un esquema de nomenclatura que minimice la probabilidad de que esto ocurra.

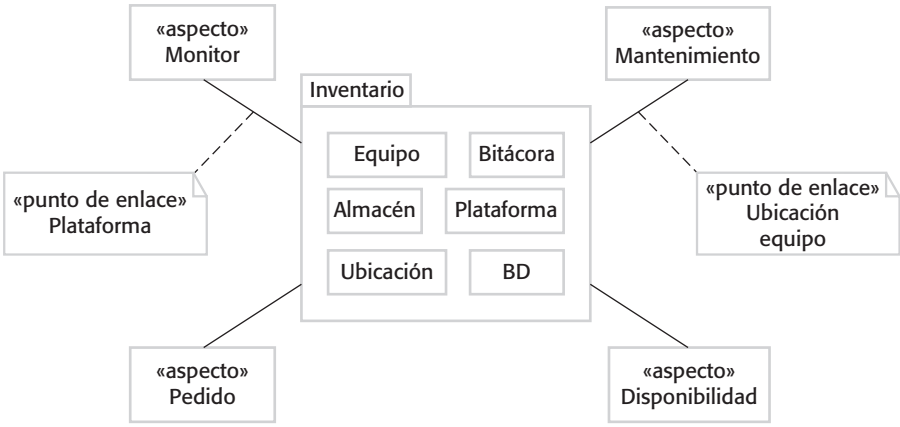


Figura 21.14 Modelo de diseño orientado a aspectos

Este proceso, naturalmente, es un proceso iterativo en el que usted hace propuestas de diseño iniciales y luego las refina conforme analiza y comprende los conflictos del diseño. Por lo general, esperaría refinar las extensiones identificadas en los requerimientos a un mayor número de aspectos.

El resultado del proceso de diseño orientado a aspectos es un modelo de diseño orientado a aspectos. Esto puede expresarse en una versión extendida del UML que incluye nuevas sentencias específicas de aspectos, tales como las propuestas por Clarke y Baniassad (2005) y Jacobson y Ng (2004). Los elementos esenciales del "UML de aspectos" son un medio para modelar aspectos y especificar los puntos de enlace donde deben combinarse los consejos de aspecto con el sistema central.

La figura 21.14 es un ejemplo de un modelo de diseño orientado a aspectos. Se usó el estereotipo UML para un aspecto propuesto por Jacobson y Ng. La figura 21.14 muestra el sistema central para un inventario de servicios de emergencia más algunos aspectos que pueden combinarse con dicho núcleo. Se exponen algunas clases del sistema central y algunos aspectos. Ésta es una imagen simplificada; un modelo completo incluiría más clases y aspectos. Observe cómo se usaron las notas UML para ofrecer información adicional acerca de las clases que son atravesadas (*cross-cut*) por algunos aspectos.

La figura 21.15 es un modelo más detallado de un aspecto. Desde luego, antes de diseñar aspectos, hay que tener un diseño del sistema central. Como aquí no hay espacio para mostrar esto, se hicieron algunas suposiciones acerca de las clases y los métodos en el sistema central.

La primera sección del aspecto establece los puntos de corte que especifican dónde se combinarán con el sistema central. Por ejemplo, el primer punto de corte especifica que el aspecto puede combinarse en el punto de enlace call getItemInfo(..). La siguiente sección define las extensiones que implementa el aspecto. En este ejemplo, el enunciado de la extensión puede leerse como:

> *En el método viewItem, después de llamar al método getItemInfo, debe incluirse un llamado al método displayHistory para mostrar el registro de mantenimiento.*

La programación orientada a aspectos (AOP, por las siglas de *aspect-oriented programming*) se inició en los laboratorios PARC de Xerox, en 1997, con el desarrollo del

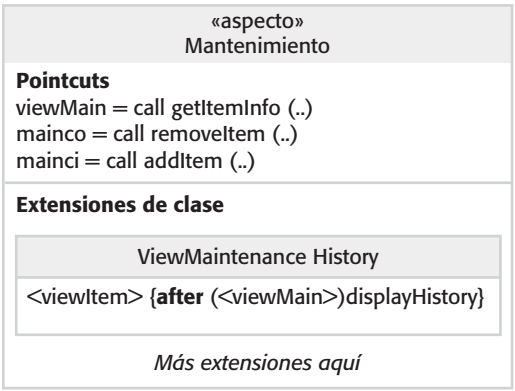


Figura 21.15 Parte de un modelo de un aspecto

lenguaje de programación AspectJ. Éste sigue siendo el lenguaje orientado a aspectos usado más ampliamente, aunque se han implementado también extensiones orientadas a aspectos de otros lenguajes, tales como C# y C++. Además, se han desarrollado otros lenguajes experimentales para soportar la separación explícita de las competencias y la composición de competencias, y existe implementación experimental de AOP en el framework .NET. La programación orientada a aspectos se cubre de manera extensa en otros libros (Colyer *et al*., 2005; Gradecki y Lezeiki, 2003; Laddad, 2003b).

Si usted siguió un enfoque orientado a aspectos para diseñar su sistema, ya identificó la funcionalidad central y las extensiones para que dicha funcionalidad se implemente como aspectos transversales. El enfoque del proceso de programación debe ser entonces escribir código que implemente la funcionalidad central y de extensión, y, de manera fundamental, especificar los puntos de corte en los aspectos para que el consejo del aspecto se teja en el código base en los lugares correctos.

Es muy importante especificar de manera correcta los puntos de corte, pues éstos definen dónde se combinarán los consejos de aspecto con la funcionalidad central. Si se comete un error en la especificación del punto de corte, entonces el consejo del aspecto se tejerá en el programa en el lugar equivocado. Esto podría conducir a un comportamiento inesperado e impredecible del programa. Es esencial respetar los estándares de nomenclatura establecidos durante el diseño del sistema. También es necesario revisar todos los aspectos para garantizar que no ocurra interferencia de aspectos si dos o más aspectos se tejen en el sistema central en el mismo punto de enlace. En general, es mejor evitar esto por completo, pero, en ocasiones, puede ser la mejor forma para implementar una competencia. En tales circunstancias, hay que asegurarse de que los aspectos son completamente independientes. El comportamiento del programa no debe depender del orden en que los aspectos se tejan en el programa.

21.3.3 Verificación y validación

Como se estudió en el capítulo 8, la verificación y la validación implican demostrar que un programa cumple su especificación (verificación) y satisface las necesidades reales de las partes interesadas (validación). Las técnicas de verificación estática se enfocan en análisis manual o automatizado del código fuente del programa. La validación o las pruebas dinámicas se usan para descubrir defectos en el programa o demostrar que el programa

cumple con sus requerimientos. Cuando la detección de defectos es el objetivo, el proceso de prueba puede guiarse mediante el conocimiento del código fuente del programa. Las métricas de cobertura de prueba muestran la efectividad de las pruebas para hacer que los enunciados del código fuente sean ejecutables.

Para los sistemas orientados a aspectos, los procesos de las pruebas de validación no son diferentes a los de cualquier otro sistema. El programa final ejecutable se trata como una caja negra y se diseñan pruebas para demostrar si el sistema cumple o no con los requerimientos. Sin embargo, el uso de aspectos causa problemas reales con las inspecciones de programa y las pruebas de caja blanca, mientras que el código fuente del programa se usa para identificar potenciales pruebas de defecto.

Las inspecciones de programa, que se describen en el capítulo 24, implican un equipo de lectores que observan el código fuente de un programa para descubrir defectos que haya introducido el programador. Es una técnica muy efectiva de descubrimiento de defectos. Sin embargo, los programas orientados a aspectos no pueden leerse secuencialmente (esto es, de arriba abajo). Por lo tanto, es más difícil que la gente los comprenda.

Un lineamiento general para la comprensibilidad del programa es que un lector pueda leer un programa de izquierda a derecha sin tener que cambiar la atención hacia otras partes del código. Esto lo hace más sencillo para los lectores y también hace menos probable que los programadores cometan errores mientras su atención se enfoca en una sola sección de código. Mejorar la legibilidad del programa fue una razón clave para la introducción de la programación estructurada (Dijkstra *et al*., 1972) y la eliminación de los enunciados de ramificación incondicional (go-to) de los lenguajes de programación de alto nivel.

En un sistema orientado a aspectos, es imposible la lectura de código secuencial. El lector debe examinar cada aspecto, comprender sus puntos de corte (que pueden ser patrones) y el modelo de punto de enlace del lenguaje orientado a aspectos. Al leer el programa, tiene que identificar todo punto de enlace potencial y entonces cambiar la atención al código de aspecto para ver si puede tejerse en dicho punto. Luego, su atención regresa al flujo principal de control del código base. En realidad, esto es cognitivamente imposible y la única forma de inspeccionar un programa orientado a aspectos es mediante el uso de herramientas de lectura de código.

Pueden escribirse herramientas de lectura de código que "aplanen" un programa orientado a aspectos y presenten un programa al lector con los aspectos "tejidos" en el programa en los puntos de enlace especificados. Sin embargo, ésta no es una solución completa al problema de lectura de código. El modelo de punto de enlace en un lenguaje de programación orientado a aspectos puede ser dinámico en vez de estático, y quizá sea imposible demostrar que el programa aplanado se comportará exactamente de la misma forma que el programa que ejecutará. Más aún, como es posible que diferentes aspectos tengan la misma especificación de punto de corte, la herramienta de lectura de programa debe saber cómo el tejedor de aspectos maneja dichos aspectos "en competencia" y cómo se ordenará la combinación.

Las pruebas de caja blanca o pruebas estructurales son un enfoque sistemático a las pruebas donde se usa el conocimiento del código fuente del programa para diseñar pruebas de defecto. La meta es diseñar pruebas que proporcionen algún nivel de cobertura de programa. Esto es, el conjunto de pruebas debe garantizar que se ejecute toda ruta lógica a través del programa, con la consecuencia de que cada enunciado de programa se efectúe al menos una vez. Pueden usarse analizadores de ejecución de programa para demostrar que se logra este nivel de cobertura de pruebas.

En un sistema orientado a aspectos, existen dos problemas con este enfoque:

1. ¿Cómo puede usarse el conocimiento del programa para derivar sistemáticamente pruebas del programa?
2. ¿Qué significa exactamente cobertura de prueba?

Para diseñar pruebas en un programa estructurado (por ejemplo, pruebas del código de un método) sin ramificaciones incondicionales, usted puede derivar un gráfico de flujo de programa, que revele toda la ruta de ejecución lógica a través de dicho programa. Entonces se examina el código y, para cada ruta a través del gráfico de flujo, se eligen valores de entrada que harán que la ruta se ejecute.

Sin embargo, un programa orientado a aspectos no es un programa estructurado. El flujo de control se interrumpe mediante enunciados "viene de" (Constantinos *et al*., 2004). En algún punto de enlace en la ejecución del código base, puede ejecutarse un aspecto. El autor no está seguro de que en tal situación sea posible construir un diagrama de flujo estructurado. Por lo tanto, es difícil diseñar sistemáticamente pruebas de programa que garanticen la ejecución de todas las combinaciones de código base y los aspectos.

En un programa orientado a aspectos, existe también el problema de decidir qué significa "cobertura de prueba". ¿Significa que el código de cada aspecto se ejecuta al menos una vez? Ésta es una condición muy débil, debido a la interacción entre los aspectos y el código base en los puntos de enlace donde se tejen los aspectos. ¿La idea de cobertura de prueba debe extenderse de manera que el código del aspecto se ejecute al menos una vez en cada punto de enlace especificado en el punto de corte del aspecto? En tales situaciones, ¿qué sucede si diferentes aspectos definen el mismo punto de corte? Se trata de problemas tanto teóricos como prácticos. Se necesitan herramientas para soportar las pruebas de los programas orientados a aspectos, que ayudarán a valorar la medida de la obertura de pruebas de un sistema.

Como se estudiará en el capítulo 24, los grandes proyectos tienen por lo general un equipo de aseguramiento de calidad separado, que establece los estándares de las pruebas y requiere un aseguramiento formal de que las revisiones y pruebas del sistema se completan bajo dichos estándares. Los problemas de inspeccionar y derivar pruebas para programas orientados a aspectos son una significativa barrera para la adopción del desarrollo de software orientado a aspectos en los proyectos de software grandes.

Además de los problemas con las inspecciones y las pruebas de caja blanca, Katz (2005) identificó problemas adicionales en las pruebas de los programas orientados a aspectos:

1. ¿Cómo deben especificarse los aspectos de manera que puedan derivarse pruebas para dichos aspectos?
2. ¿Cómo pueden probarse los aspectos independientemente del sistema base con el que deben tejerse?
3. ¿Cómo puede someterse a prueba la interferencia de aspectos? Como se estudió, la interferencia de aspectos ocurre cuando dos o más aspectos usan la misma especificación de punto de corte.
4. ¿Cómo pueden diseñarse pruebas de manera que se ejecuten todos los puntos de enlace de programa y se apliquen pruebas de aspecto adecuadas?

En esencia, estos problemas con las pruebas ocurren porque los aspectos están estrechamente integrados con el código base de un sistema. Por lo tanto, son difíciles de probar en aislamiento. Puesto que están tejidos en un programa en muchos lugares diferentes, no podemos estar seguros de que un aspecto que funciona con éxito en un punto de enlace funcionará necesariamente en todos los puntos de enlace. Todos éstos siguen siendo problemas de investigación para el desarrollo de software orientado a aspectos.

PUNTOS CLAVE

- El principal beneficio de un enfoque orientado a aspectos al desarrollo de software es que soporta la separación de competencias. Al representar las competencias transversales como aspectos, las competencias individuales pueden entenderse, reutilizarse y modificarse sin cambiar otras partes del programa.
- El enredo ocurre cuando un módulo en un sistema incluye código que implementa diferentes requerimientos del sistema. El fenómeno relacionado de dispersión ocurre cuando la implementación de una sola competencia se dispersa a través de varios componentes en un programa.
- Los aspectos incluyen un punto de corte (un enunciado que define dónde se tejerá el aspecto dentro del programa) y consejos (el código para implementar la competencia transversal). Los puntos de enlace son los eventos que se pueden referenciar en un punto de corte.
- Para garantizar la separación de las competencias, los sistemas pueden diseñarse como un sistema central que implementa las principales competencias de las partes interesadas, y un conjunto de extensiones que implementan competencias secundarias.
- Para identificar las competencias se puede usar un enfoque a la ingeniería de requerimientos orientado a puntos de vista, que permita recuperar los requerimientos de las partes interesadas e identificar la calidad de servicio transversal y las competencias de política.
- La transición de requerimientos a diseño puede hacerse identificando casos de uso, cada uno de los cuales representa una competencia de las partes interesadas. El diseño puede modelarse mediante una versión extendida del UML con estereotipos de aspectos.
- Los problemas de inspeccionar y derivar pruebas para programas orientados a aspectos son una barrera significativa para la adopción del desarrollo de software orientado a aspectos en los proyectos de software grandes.

LECTURAS SUGERIDAS

"Aspect-oriented programming". Este número especial del CACM cuenta con algunos artículos dirigidos a un público general, que son un buen punto de partida para leer acerca de la programación orientada a aspectos. (*Comm. ACM*, **44** (10), octubre de 2001.) http://dx.doi.org/10.1145/383845.383846.

Aspect-oriented Software Development. Un libro escrito por diferentes autores en conjunto, quienes presentan una gran variedad de ensayos sobre el desarrollo de software orientado a aspectos; entre sus autores se encuentran muchos de los investigadores líderes en el campo. (R. E. Filman, T. Elrad, S. Clarke y M. Aksit, Addison-Wesley, 2005.)

Aspect-oriented Software Development with Use cases. Éste es un libro práctico para diseñadores de software. Los autores analizan cómo usar casos de uso para gestionar la separación de competencias y utilizarlos como la base de un diseño orientado a aspectos. (I. Jacobson y P. Ng, Addison-Wesley, 2005.)

EJERCICIOS

21.1. ¿Cuáles son los diferentes tipos de competencias de las partes interesadas que pueden surgir en un sistema grande? ¿Cómo pueden los aspectos soportar la implementación de cada uno de estos tipos de competencias?

21.2. Resuma qué se entiende por enredos y dispersión. Con ejemplos, explique por qué los enredos y la dispersión pueden causar problemas cuando cambian los requerimientos del sistema.

21.3. ¿Cuál es la diferencia entre un punto de enlace y un punto de corte? Explique cómo éstos facilitan el tejido de código en un programa para manejar las competencias transversales.

21.4. ¿Qué suposiciones subyacen en la idea de que un sistema debe estar organizado como un sistema central que implemente los requerimientos esenciales, más extensiones que implementen funcionalidad adicional? ¿Puede pensar en sistemas donde este modelo no sería adecuado?

21.5. ¿Cuáles puntos de vista deben considerarse cuando se desarrollan especificaciones de requerimientos para el MHC-PMS? ¿Cuáles son probablemente las competencias transversales más importantes?

21.6. Considerando la funcionalidad para cada punto de vista mostrado en la figura 21.9, identifique seis casos de uso para el sistema de inventario de equipo, además de aquellos que se indican en la figura 21.11. Cuando sea adecuado, indique cómo algunos de éstos podrían organizarse como extensión de casos de uso.

21.7. Con la notación de estereotipo de aspectos que se ilustró en la figura 21.15, desarrolle con más detalle los aspectos Pedido y Monitor, que se muestran en la figura 21.14.

21.8. Explique cómo puede surgir interferencia de aspectos y sugiera lo que debería hacerse durante el proceso de diseño del sistema para reducir los problemas de interferencia de aspectos.

21.9. Explique por qué expresar las especificaciones de punto de corte como patrones aumenta los problemas de probar los programas orientados a aspectos. Para responder, piense en cómo las pruebas de los programas implican generalmente una comparación de la salida esperada con la salida real producida por un programa.

21.10. Sugiera cómo podría usar aspectos para simplificar la depuración de programas.

REFERENCIAS

Birrer, I., Pasetti, A. y Rohlik, O. (2005). "The XWeaver Project: Aspect-oriented Programming for On-Board Applications". http://control.ee.ethz.ch/index.cgi?page=publications;action=details;id=2361

Clark, S. y Baniassad, E. (2005). *Aspect-Oriented Analysis and Design: The Theme Approach*. Harlow, UK: Addison-Wesley.

Colyer, A. y Clement, A. (2005). "Aspect-oriented programming with AspectJ". *IBM Systems J.*, **44** (2), 301–8.

Colyer, A., Clement, A., Harley, G. y Webster, M. (2005). *Eclipse AspectJ*. Upper Saddle River, NJ: Addison-Wesley.

Constantinos, C., Skotiniotis, T. y Stoerzer, T. (2004). "AOP considered harmful". *European Interactive Workshop on Aspects in Software (EIWAS' 04)*, Berlín, Alemania.

Dijkstra, E. W., Dahl, O. J. y Hoare, C. A. R. (1972). *Structured Programming*. Londres: Academic Press.

Easterbrook, S. y Nuseibeh, B. (1996). "Using ViewPoints for inconsistency management". *BCS/IEE Software Eng. J.*, **11** (1), 31–43.

Finkelstein, A., Kramer, J., Nuseibeh, B. y Goedicke, M. (1992). "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development". *Int. J. of Software Engineering and Knowledge Engineering*, **2** (1), 31–58.

Gradecki, J. D. y Lezeiki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. Nueva York: John Wiley & Sons.

Jacobson, I. y Ng, P-W. (2004). *Aspect-oriented Software Development with Use Cases*. Boston: Addison-Wesley.

Katz, S. (2005). "A Survey of Verification and Static Analysis for Aspects". http://www.aosd-europe.net/documents/verificM81.pdf

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. y Griswold, W. G. (2001). "Getting Started with AspectJ". *Comm. ACM*, **44** (10), 59–65.

Kotonya, G. y Sommerville, I. (1996). "Requirements engineering with viewpoints". *BCS/IEE Software Eng. J.*, **11** (1), 5–18.

Laddad, R. (2003a). *AspectJ in Action*. Greenwich, Conn.: Manning Publications Co.

Laddad, R. (2003b). *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, Conn.: Manning Publications.

Sommerville, I. y Sawyer, P. (1997). "Viewpoints: principles, problems and a practical approach to requirements engineering". *Annals of Software Engineering*, **3** 101–30.

Sommerville, I., Sawyer, P. y Viller, S. (1998). "Viewpoints for requirements elicitation: a practical approach". *3rd Int. Conf. on Requirements Engineering*. Colorado: IEEE Computer Society Press, 74–81.

PARTE 4 Gestión de software

Algunas veces se ha dicho que la diferencia principal entre la ingeniería de software y otros tipos de programación radica en que la primera es un proceso de gestión. Debido a ello se entiende que el desarrollo de software tiene lugar dentro de una organización y está sujeto a una variedad de restricciones de tiempo, presupuesto y organizacionales. Por lo tanto, para la ingeniería de software, la gestión es muy importante. En esta parte del libro se presentan varios temas de administración con un enfoque en conflictos de gestión técnica y no en cuestiones administrativas "más suaves", como la gestión de personal o la gestión más estratégica de los sistemas empresariales.

El capítulo 22 introduce la gestión de proyectos de software, y su primera sección principal se refiere a la gestión del riesgo. Junto con la planeación de proyectos, la gestión del riesgo, en la que los gestores identifican aquello que puede salir mal y planean lo que pueden hacer al respecto, es una responsabilidad clave de la gestión del proyecto. Este capítulo incluye también secciones acerca de gestión de personal y trabajo en equipo.

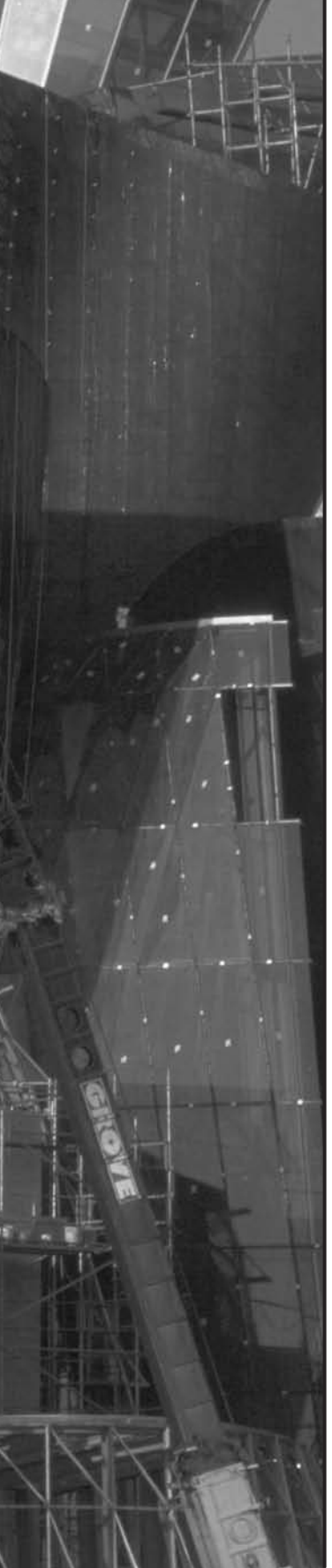
El capítulo 23 comprende la planeación y estimación de proyectos. Se exponen las gráficas de barras como herramientas fundamentales de planeación y se explica por qué el desarrollo dirigido por un plan seguirá siendo un importante enfoque de desarrollo, a pesar del éxito de los

métodos ágiles. También se analizan los conflictos que influyen en el precio que se cobra por un sistema y las técnicas de estimación de costo de software. Se usa la familia de modelos de costo COCOMO para describir el modelado algorítmico de costos y explicar los beneficios y las desventajas de este enfoque.

Los últimos tres capítulos (del 24 al 26) se ocupan de cuestiones referentes a la gestión de la calidad. Ésta tiene que ver con los procesos y las técnicas para garantizar y mejorar la calidad del software, un tema que se presenta en el capítulo 24. Se estudia la importancia de los estándares en la gestión de calidad, el uso de revisiones e inspecciones en el proceso de aseguramiento de la calidad y el papel de la medición del software en la gestión de la calidad.

El capítulo 25 explica la administración de la configuración. Éste es un tema relevante para todos los sistemas grandes que desarrollan los equipos. Sin embargo, la necesidad de administración de la configuración no siempre es clara para los estudiantes, quienes sólo se interesan por el desarrollo de software personal; por ello, se describen aquí los diversos aspectos de este tema, incluida la planeación de la configuración, la gestión de versiones, la construcción de sistemas y la administración del cambio.

Finalmente, el capítulo 26 se ocupa del mejoramiento del proceso de software: ¿Cómo pueden modificarse los procesos para que mejoren tanto los atributos del producto como del proceso? Se estudian las etapas de un proceso de mejoramiento de un proceso genérico, esto es, medición, análisis y cambio de proceso. Posteriormente, se analiza el enfoque basado en capacidades de SEI para el mejoramiento de procesos y se describen brevemente los modelos de capacidad de madurez.



22

Gestión de proyectos

Objetivos

El objetivo de este capítulo es introducirlo a la gestión de proyectos de software y a dos importantes actividades relacionadas, esto es, la gestión del riesgo y de personal. Al estudiar este capítulo:

- conocerá las principales tareas de los administradores de proyectos de software;
- se introducirá a la noción de gestión del riesgo y a algunos de los riesgos que pueden surgir en los proyectos de software;
- comprenderá los factores que influyen en la motivación personal y qué significan para los administradores de proyectos de software;
- entenderá los conflictos clave que influyen en el trabajo en equipo, tales como la composición del equipo, la organización y la comunicación.

Contenido

22.1 Gestión del riesgo
22.2 Gestión de personal
22.3 Trabajo en equipo

La gestión de proyectos de software es una parte esencial de la ingeniería de software. Los proyectos necesitan administrarse porque la ingeniería de software profesional está sujeta siempre a restricciones organizacionales de presupuesto y fecha. El trabajo del administrador del proyecto es asegurarse de que el proyecto de software cumpla y supere tales restricciones, además de que entregue software de alta calidad. La buena gestión no puede garantizar el éxito del proyecto. Sin embargo, la mala gestión, por lo general, da como resultado una falla del proyecto: el software puede entregarse tarde, costar más de lo estimado originalmente o no cumplir las expectativas de los clientes.

Desde luego, los criterios de éxito para la gestión del proyecto varían de un proyecto a otro, pero, para la mayoría de los proyectos, las metas importantes son:

1. Entregar el software al cliente en el tiempo acordado.
2. Mantener costos dentro del presupuesto general.
3. Entregar software que cumpla con las expectativas del cliente.
4. Mantener un equipo de desarrollo óptimo y con buen funcionamiento.

Tales metas no son únicas para la ingeniería de software, pero sí lo son para todos los proyectos de ingeniería. Sin embargo, la ingeniería de software es diferente en algunas formas a otros tipos de ingeniería que hacen a la gestión del software particularmente desafiante. Algunas de estas diferencias son:

1. *El producto es intangible* Un administrador de un astillero o un proyecto de ingeniería civil pueden ver el producto conforme se desarrolla. Si hay retraso en el calendario, es visible el efecto sobre el producto: es evidente que algunas partes de la estructura no están terminadas. El software es intangible. No se puede ver ni tocar. Los administradores de proyectos de software no pueden constatar el progreso con sólo observar el artefacto que se construye. Más bien, ellos se apoyan en otros para crear la prueba que pueden utilizar al revisar el progreso del trabajo.
2. *Los grandes proyectos de software con frecuencia son proyectos excepcionales* Los grandes proyectos de software se consideran en general diferentes en ciertas formas de los proyectos anteriores. Por eso, incluso los administradores que cuentan con vasta experiencia pueden encontrar difícil anticiparse a los problemas. Aunado a esto, los vertiginosos cambios tecnológicos en computadoras y comunicaciones pueden volver obsoleta la experiencia de un administrador. Las lecciones aprendidas de proyectos anteriores pueden no ser aplicables a nuevos proyectos.
3. *Los procesos de software son variables y específicos de la organización* El proceso de ingeniería para algunos tipos de sistema, como puentes y edificios, es bastante comprendido. Sin embargo, los procesos de software varían considerablemente de una organización a otra. Aunque se ha producido un notorio avance en la estandarización y el mejoramiento de los procesos, no es posible predecir de manera confiable cuándo un proceso de software particular conducirá a problemas de desarrollo. Esto es especialmente cierto si el proyecto de software es parte de un proyecto de ingeniería de sistemas más amplio.

Debido a estos conflictos, no es sorpresivo que algunos proyectos de software se retrasen y excedan el presupuesto. A menudo, los sistemas de software son nuevos y

técnicamente innovadores. Los proyectos de ingeniería (como los nuevos sistemas de transporte) que son reformadores, normalmente también tienen problemas de calendario. Dadas las dificultades, quizá sea asombroso que tantos proyectos de software ¡se entreguen a tiempo y dentro del presupuesto!

Es imposible efectuar una descripción laboral estándar para un administrador de proyecto de software. La labor varía enormemente en función de la organización y el producto de software a desarrollar. No obstante, la mayoría de los administradores, en alguna etapa, toman la responsabilidad de varias o todas las siguientes actividades:

1. *Planeación del proyecto* Los administradores de proyecto son responsables de la planeación, estimación y calendarización del desarrollo del proyecto, así como de la asignación de tareas a las personas. Supervisan el trabajo para verificar que se realice de acuerdo con los estándares requeridos y monitorizan el avance para comprobar que el desarrollo esté a tiempo y dentro del presupuesto.
2. *Informes* Los administradores de proyectos por lo común son responsables de informar del avance de un proyecto a los clientes y administradores de la compañía que desarrolla el software. Deben ser capaces de comunicarse en varios niveles, desde codificar información técnica detallada hasta elaborar resúmenes administrativos. Deben redactar documentos concisos y coherentes que sinteticen información crítica de reportes detallados del proyecto. Es necesario que esta información se presente durante las revisiones de avance.
3. *Gestión del riesgo* Los administradores de proyecto tienen que valorar los riesgos que pueden afectar un proyecto, monitorizar dichos riesgos y emprender acciones cuando surjan problemas.
4. *Gestión de personal* Los administradores de proyecto son responsables de administrar un equipo de personas. Deben elegir a los integrantes de sus equipos y establecer formas de trabajar que conduzcan a desempeño efectivo del equipo.
5. *Redactar propuestas* La primera etapa en un proyecto de software puede implicar escribir una propuesta para obtener un contrato de trabajo. La propuesta describe los objetivos del proyecto y cómo se realizará. Por lo general, incluye estimaciones de costo y calendarización, además de justificar por qué el contrato del proyecto debería concederse a una organización o un equipo particular. La escritura de propuestas es una tarea esencial, pues la supervivencia de muchas compañías de software depende de contar con suficientes propuestas aceptadas y concesiones de contratos. Es posible que no haya lineamientos establecidos para esta tarea; la escritura de propuestas es una habilidad que se adquiere a través de práctica y experiencia.

En este capítulo nos centraremos en la gestión del riesgo y la gestión de personal. La planeación de proyectos es un tema importante por derecho propio, que se estudia en el capítulo 23.

22.1 Gestión del riesgo

La gestión del riesgo es una de las tareas más sustanciales para un administrador de proyecto. La gestión del riesgo implica anticipar riesgos que pudieran alterar el calendario del

proyecto o la calidad del software a entregar, y posteriormente tomar acciones para evitar dichos riesgos (Hall, 1998; Ould, 1999). Podemos considerar un riesgo como algo que es preferible que no ocurra. Los riesgos pueden amenazar el proyecto, el software que se desarrolla o a la organización. Por lo tanto, existen tres categorías relacionadas de riesgo:

1. *Riesgos del proyecto* Los riesgos que alteran el calendario o los recursos del proyecto. Un ejemplo de riesgo de proyecto es la renuncia de un diseñador experimentado. Encontrar un diseñador de reemplazo con habilidades y experiencia adecuadas puede demorar mucho tiempo y, en consecuencia, el diseño del software tardará más tiempo en completarse.
2. *Riesgos del producto* Los riesgos que afectan la calidad o el rendimiento del software a desarrollar. Un ejemplo de riesgo de producto es la falla que presenta un componente que se adquirió al no desempeñarse como se esperaba. Esto puede afectar el rendimiento global del sistema, de modo que es más lento de lo previsto.
3. *Riesgos empresariales* Riesgos que afectan a la organización que desarrolla o adquiere el software. Por ejemplo, un competidor que introduce un nuevo producto es un riesgo empresarial. La introducción de un producto competitivo puede significar que las suposiciones hechas sobre las ventas de los productos de software existentes sean excesivamente optimistas.

Desde luego, estos tipos de riesgos se traslapan. Si un programador experimentado abandona un proyecto, esto puede ser un riesgo del proyecto porque, incluso si se sustituye de manera inmediata, el calendario se alterará. Siempre se requiere tiempo para que un nuevo miembro del proyecto comprenda el trabajo realizado, de manera que no puede ser inmediatamente productivo. En consecuencia, la entrega del sistema podría demorarse. La salida de un miembro del equipo también puede ser un riesgo del producto, porque un sustituto tal vez no sea tan experimentado y, por lo tanto, podría cometer errores de programación. Finalmente, puede ser un riesgo empresarial, porque la experiencia de dicho programador es vital para obtener nuevos contratos.

Es necesario registrar los resultados del análisis del riesgo en el plan del proyecto, junto con un análisis de consecuencias, que establece las consecuencias del riesgo para el proyecto, el producto y la empresa. La gestión de riesgos efectiva facilita hacer frente a los problemas y asegurar que éstos no conduzcan a un presupuesto inaceptable o a retrasos en el calendario.

Los riesgos específicos que podrían afectar un proyecto dependen del proyecto y el entorno de la organización donde se desarrolla el software. Sin embargo, también existen riesgos comunes que no se relacionan con el tipo de software a desarrollar y que pueden ocurrir en cualquier proyecto. En la figura 22.1 se muestran algunos de estos riesgos comunes.

La gestión del riesgo es particularmente importante para los proyectos de software, debido a la incertidumbre inherente que enfrentan la mayoría de proyectos. Ésta se deriva de requerimientos vagamente definidos, cambios de requerimientos que obedecen a cambios en las necesidades del cliente, dificultades en estimar el tiempo y los recursos requeridos para el desarrollo de software, o bien, se deriva de diferencias en las habilidades individuales. Es necesario anticipar los riesgos; comprender el efecto de estos riesgos sobre el proyecto, el producto y la empresa; y dar los pasos adecuados para evitar dichos riesgos. Tal vez se necesite diseñar planes de contingencia de manera que, si ocurren los riesgos, se puedan tomar acciones inmediatas de recuperación.

Riesgo	Repercute en	Descripción
Rotación de personal	Proyecto	Personal experimentado abandonará el proyecto antes de que éste se termine.
Cambio administrativo	Proyecto	Habrá un cambio de gestión en la organización con diferentes prioridades.
Indisponibilidad de hardware	Proyecto	Hardware, que es esencial para el proyecto, no se entregará a tiempo.
Cambio de requerimientos	Proyecto y producto	Habrá mayor cantidad de cambios a los requerimientos que los anticipados.
Demoras en la especificación	Proyecto y producto	Especificaciones de interfaces esenciales no están disponibles a tiempo.
Subestimación del tamaño	Proyecto y producto	Se subestimó el tamaño del sistema.
Bajo rendimiento de las herramientas CASE	Producto	Las herramientas CASE, que apoyan el proyecto, no se desempeñan como se anticipaba.
Cambio tecnológico	Empresa	La tecnología subyacente sobre la cual se construye el sistema se sustituye con nueva tecnología.
Competencia de productos	Empresa	Un producto competitivo se comercializa antes de que el sistema esté completo.

Figura 22.1 Ejemplos de riesgos comunes para el proyecto, el producto y la empresa

En la figura 22.2 se ilustra una idea general del proceso de gestión del riesgo. Comprende varias etapas:

1. *Identificación del riesgo* Hay que identificar posibles riesgos para el proyecto, el producto y la empresa.
2. *Análisis de riesgos* Se debe valorar la probabilidad y las consecuencias de dichos riesgos.
3. *Planeación del riesgo* Es indispensable elaborar planes para enfrentar el riesgo, evitarlo o minimizar sus efectos en el proyecto.
4. *Monitorización del riesgo* Hay que valorar regularmente el riesgo y los planes para atenuarlo, y revisarlos cuando se aprenda más sobre el riesgo.

Es preciso documentar los resultados del proceso de gestión del riesgo en un plan de gestión del riesgo. Éste debe incluir un estudio de los riesgos que enfrenta el proyecto, un análisis de dichos riesgos e información de cómo se gestionará el riesgo cuando es probable que se convierta en un problema.

El proceso de gestión del riesgo es un proceso iterativo que continúa a lo largo del proyecto. Una vez desarrollado un plan de gestión del riesgo inicial, se monitoriza la situación para detectar riesgos emergentes. Conforme está disponible más información referente a los riesgos, habrá que volver a analizar los riesgos y decidir si cambió la prioridad del riesgo. Entonces tal vez sea necesario cambiar los planes para evitar el riesgo y gestionar la contingencia.

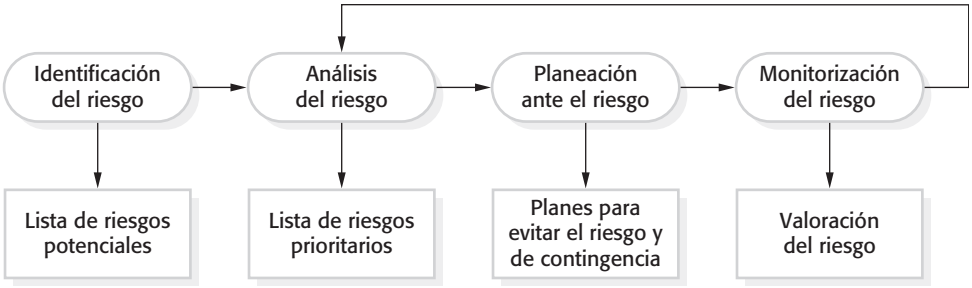


Figura 22.2 El proceso de gestión del riesgo

22.1.1 Identificación del riesgo

La identificación del riesgo es la primera etapa del proceso de gestión del riesgo. Se ocupa de identificar los riesgos que pudieran plantear una mayor amenaza al proceso de ingeniería de software, al software a desarrollar, o a la organización que lo desarrolla. La identificación del riesgo puede ser un proceso de equipo en el que este último se reúne para pensar en posibles riesgos. O bien, el administrador del proyecto, con base en su experiencia, identifica los riesgos más probables o críticos.

Como punto de partida para la identificación del riesgo, se recomienda utilizar una lista de verificación de diferentes tipos de riesgo. Existen al menos seis tipos de riesgos que pueden incluirse en una lista de verificación:

1. *Riesgos tecnológicos* Se derivan de las tecnologías de software o hardware usadas para desarrollar el sistema.
2. *Riesgos personales* Se asocian con las personas en el equipo de desarrollo.
3. *Riesgos organizacionales* Se derivan del entorno organizacional donde se desarrolla el software.
4. *Riesgos de herramientas* Resultan de las herramientas de software y otro software de soporte que se usa para desarrollar el sistema.
5. *Riesgos de requerimientos* Proceden de cambios a los requerimientos del cliente y del proceso de gestionarlos.
6. *Riesgos de estimación* Surgen de las estimaciones administrativas de los recursos requeridos para construir el sistema.

La figura 22.3 brinda algunos ejemplos de posibles riesgos en cada una de estas categorías. Al concluir el proceso de identificación de riesgos, se tendrá una larga lista de eventualidades que podrían ocurrir y afectar al producto, al proceso y a la empresa. Entonces se necesita reducir esta lista a un tamaño razonable. Si existen demasiados riesgos, será prácticamente imposible seguir la huella de todos ellos.

22.1.2 Análisis de riesgo

Durante el proceso de análisis de riesgos, hay que considerar cada riesgo identificado y realizar un juicio acerca de la probabilidad y gravedad de dicho riesgo. No hay una forma sencilla de hacer esto. Usted debe apoyarse en su propio juicio y en la experiencia

Tipo de riesgo	Riesgos posibles
Tecnológico	La base de datos que se usa en el sistema no puede procesar tantas transacciones por segundo como se esperaba. (1) Los componentes de software de reutilización contienen defectos que hacen que no puedan reutilizarse como se planeó. (2)
Personal	Es imposible reclutar personal con las habilidades requeridas. (3) El personal clave está enfermo e indispuesto en momentos críticos. (4) No está disponible la capacitación requerida para el personal. (5)
De organización	La organización se reestructura de modo que diferentes administraciones son responsables del proyecto. (6) Problemas financieros de la organización fuerzan reducciones en el presupuesto del proyecto. (7)
Herramientas	El código elaborado por las herramientas de generación de código de software es ineficiente. (8) Las herramientas de software no pueden trabajar en una forma integrada. (9)
Requerimientos	Se proponen cambios a los requerimientos que demandan mayor trabajo de rediseño. (10) Los clientes no entienden las repercusiones de los cambios a los requerimientos. (11)
Estimación	Se subestima el tiempo requerido para desarrollar el software. (12) Se subestima la tasa de reparación de defectos. (13) Se subestima el tamaño del software. (14)

Figura 22.3 Ejemplos de diferentes tipos de riesgos

obtenida en los proyectos anteriores y los problemas que surgieron en ellos. No es posible hacer valoraciones precisas y numéricas de la probabilidad y gravedad de cada riesgo. En vez de ello, habrá que asignar el riesgo a una de ciertas bandas:

1. La probabilidad del riesgo puede valorarse como muy baja (< 10%), baja (del 10 al 25%), moderada (del 25 al 50%), alta (del 50 al 75%) o muy alta (> 75%).
2. Los efectos del riesgo pueden estimarse como catastróficos (amenazan la supervivencia del proyecto), graves (causarían grandes demoras), tolerables (demoras dentro de la contingencia permitida) o insignificantes.

Luego hay que tabular los resultados de este proceso de análisis mediante una tabla clasificada de acuerdo con la gravedad del riesgo. La figura 22.4 ilustra esto para los riesgos que se identificaron en la figura 22.3. Desde luego, aquí la valoración de la probabilidad y seriedad son arbitrarias. Para hacer esta valoración, se necesita información detallada del proyecto, el proceso, el equipo de desarrollo y la organización.

Desde luego, tanto la probabilidad como la valoración de los efectos de un riesgo pueden cambiar conforme se disponga de más información acerca del riesgo y a medida que se implementen planes de gestión del riesgo. Por lo tanto, esta tabla se debe actualizar durante cada iteración del proceso de riesgo.

Una vez analizados y clasificados los riesgos, valore cuáles son los más significativos. Su juicio debe depender de una combinación de la probabilidad de que el riesgo surja junto con los efectos de dicho riesgo. En general, los riesgos catastróficos deben considerarse siempre, así como los riesgos graves con más de una probabilidad moderada de ocurrencia.

Riesgo	Probabilidad	Efectos
Problemas financieros de la organización fuerzan reducciones en el presupuesto del proyecto. (7)	Baja	Catastrófico
Es imposible reclutar personal con las habilidades requeridas. (3)	Alta	Catastrófico
El personal clave está enfermo e indispuesto en momentos críticos. (4)	Moderada	Grave
Los componentes de software de reutilización contienen defectos que hacen que no puedan reutilizarse como se planeó. (2)	Moderada	Grave
Se proponen cambios a los requerimientos que demandan mayor trabajo de rediseño. (10)	Moderada	Grave
La organización se reestructura de modo que diferentes administraciones son responsables del proyecto. (6)	Alta	Grave
La base de datos que se usa en el sistema no puede procesar tantas transacciones por segundo como se esperaba. (1)	Moderada	Grave
Se subestima el tiempo requerido para desarrollar el software. (12)	Alta	Grave
Las herramientas de software no pueden trabajar en una forma integrada. (9)	Alta	Tolerable
Los clientes no entienden las repercusiones de los cambios a los requerimientos. (11)	Moderada	Tolerable
No está disponible la capacitación requerida para el personal. (5)	Moderada	Tolerable
Se subestima la tasa de reparación de defecto. (13)	Moderada	Tolerable
Se subestima el tamaño del software. (14)	Alta	Tolerable
El código elaborado por las herramientas de generación de código de software es ineficiente. (8)	Moderada	Insignificante

Figura 22.4 Tipos de riesgos y ejemplos

Boehm (1988) recomienda identificar y monitorizar los 10 riesgos principales, pero considera que esta cifra es más bien arbitraria. El número correcto de riesgos a monitorizar debe depender del proyecto. Pueden ser cinco o 15. Sin embargo, el número de riesgos elegidos para monitorizar debe ser manejable. Un número de riesgos muy grande requeriría recopilar demasiada información. A partir de los riesgos identificados en la figura 22.4, es adecuado considerar los ocho riesgos que tienen consecuencias catastróficas o graves (figura 22.5).

22.1.3 Planeación del riesgo

El proceso de planeación del riesgo considera cada uno de los riesgos clave identificados y desarrolla estrategias para manejarlos. Para cada uno de los riesgos, usted debe considerar las acciones que puede tomar para minimizar la perturbación del proyecto si se produce el problema identificado en el riesgo. También debe pensar en la información que tal vez necesite recopilar mientras observa el proyecto para que pueda anticipar los problemas.

Riesgo	Estrategia
Problemas financieros de la organización	Prepare un documento informativo para altos ejecutivos en el que muestre cómo el proyecto realiza una aportación muy importante a las metas de la empresa y presente razones por las que los recortes al presupuesto del proyecto no serían efectivos en costo.
Problemas de reclutamiento	Alerte al cliente de dificultades potenciales y de la posibilidad de demoras; investigue la compra de componentes.
Enfermedad del personal	Reorganice los equipos de manera que haya más traslape de trabajo y, así, las personas comprendan las labores de los demás.
Componentes defectuosos	Sustituya los componentes potencialmente defectuosos con la compra de componentes de conocida fiabilidad.
Cambios de requerimientos	Obtenga información de seguimiento para valorar el efecto de cambiar los requerimientos; maximice la información que se oculta en el diseño.
Reestructuración de la organización	Prepare un documento informativo para altos ejecutivos en el que muestre cómo el proyecto realiza una aportación muy importante a las metas de la empresa.
Rendimiento de la base de datos	Investigue la posibilidad de comprar una base de datos de mayor rendimiento.
Subestimación del tiempo de desarrollo	Investigue los componentes comprados; indague el uso de un generador de programa.

Figura 22.5 Estrategias para ayudar a gestionar el riesgo

Nuevamente, no hay un proceso simple que pueda seguirse para la planeación de contingencias. Se apoya en el juicio y la experiencia del administrador del proyecto.

La figura 22.5 muestra posibles estrategias de gestión del riesgo que se identificaron como los principales riesgos (es decir, aquellos que son graves o intolerables) que se incluyen en la figura 22.4. Dichas estrategias se establecen en tres categorías:

1. *Estrategias de evitación* Seguir estas estrategias significa que se reducirá la probabilidad de que surja el riesgo. Un ejemplo de estrategia de evitación del riesgo es la estrategia de enfrentar los componentes defectuosos que se muestran en la figura 22.5.
2. *Estrategias de minimización* Seguir estas estrategias significa que se reducirá el efecto del riesgo. Un ejemplo de estrategia de minimización de un riesgo es la estrategia para las enfermedades del personal que se indica en la figura 22.5.
3. *Planes de contingencia* Seguir estas estrategias significa que se está preparado para lo peor y se tiene una estrategia para hacer frente a ello. Un ejemplo de estrategia de contingencia es la estrategia para los problemas financieros de la organización que se indica en la figura 22.5.

Aquí se observa una clara analogía con las estrategias utilizadas en los sistemas críticos para garantizar fiabilidad, seguridad y protección, cuando hay que evitar, tolerar o recuperarse de las fallas. Desde luego, es mejor usar una estrategia que evitar el riesgo. Si esto no es posible, se debe usar una estrategia que reduzca las posibilidades de que el riesgo cause graves efectos. Finalmente, se debe contar con estrategias para enfrentar el riesgo cuando éste surja. Tales estrategias deben reducir el efecto global de un riesgo en el proyecto o el producto.

Tipo de riesgo	Indicadores potenciales
Tecnológico	Entrega tardía de hardware o software de soporte; muchos problemas tecnológicos reportados.
Personal	Baja moral de personal; malas relaciones entre miembros del equipo; alta rotación de personal.
De organización	Chismes en la organización; falta de acción de los altos ejecutivos.
Herramientas	Renuencia de los miembros del equipo para usar herramientas; quejas acerca de las herramientas CASE; demandas por estaciones de trabajo mejor equipadas.
Requerimientos	Muchas peticiones de cambio de requerimientos; quejas de los clientes.
Estimación	Falla para cumplir con el calendario acordado; falla para corregir los defectos reportados.

Figura 22.6
Indicadores de riesgo

22.1.4 Monitorización del riesgo

La monitorización del riesgo es el proceso para comprobar que no han cambiado sus suposiciones sobre riesgos del producto, el proceso y la empresa. Hay que valorar regularmente cada uno de los riesgos identificados para decidir si este riesgo se vuelve más o menos probable. También se tiene que considerar si los efectos del riesgo han cambiado o no. Para hacer esto, observe otros factores, como el número de peticiones de cambio de requerimientos, lo que da pistas acerca de la probabilidad del riesgo y sus efectos. Dichos factores dependen claramente de los tipos de riesgos. La figura 22.6 proporciona algunos ejemplos de factores que pueden ser útiles para valorar estos tipos de riesgos.

Los riesgos deben monitorizarse comúnmente en todas las etapas del proyecto. En cada revisión administrativa, es necesario reflexionar y estudiar cada uno de los riesgos clave por separado. También hay que decidir si es más o menos probable que surja el riesgo, y si cambiaron la gravedad y las consecuencias del riesgo.

22.2 Gestión de personal

Las personas que trabajan en una organización de software son los activos más importantes. Cuesta mucho dinero reclutar y retener al buen personal, así que depende de los administradores de software garantizar que la organización obtenga el mejor aprovechamiento posible por su inversión. En las compañías y economías exitosas, esto se logra cuando la organización respeta a las personas y les asigna responsabilidades que reflejan sus habilidades y experiencia.

Es importante que los administradores de proyecto de software comprendan los conflictos técnicos que influyen en el trabajo del desarrollo de software. Sin embargo, por desgracia, los buenos ingenieros de software no necesariamente son buenos administradores de personal. Los ingenieros de software con frecuencia tienen grandes

habilidades técnicas, pero pueden carecer de habilidades más sutiles que les permitan motivar y dirigir a un equipo de desarrollo de proyecto. Como administrador de proyecto, usted deberá estar al tanto de los problemas potenciales de administrar personal y debe tratar de desarrollar habilidades de gestión de recursos humanos.

Desde la perspectiva del autor, existen cuatro factores críticos en la gestión de personal:

1. *Consistencia* Todas las personas en un equipo de proyecto deben recibir un trato similar. Nadie espera que todas las distinciones sean idénticas, pero las personas podrían sentir que sus aportaciones a la organización se menosprecian.
2. *Respeto* Las personas tienen distintas habilidades y los administradores deben respetar esas diferencias. Todos los miembros del equipo deben recibir una oportunidad para aportar. Desde luego, en algunos casos, usted encontrará que las personas simplemente no se ajustan al equipo y no pueden continuar, pero es importante no adelantar conclusiones sobre esto en una etapa temprana del proyecto.
3. *Inclusión* Las personas contribuyen efectivamente cuando sienten que otros las escuchan y que sus propuestas se toman en cuenta. Es importante desarrollar un ambiente laboral donde se consideren todas las visiones, incluso las del personal más joven.
4. *Honestidad* Como administrador, siempre debe ser honesto acerca de lo que está bien y lo que está mal en el equipo. También debe ser honesto respecto a su nivel de conocimiento técnico y voluntad para comunicar al personal más conocimiento cuando sea necesario. Si trata de encubrir la ignorancia o los problemas, con el tiempo, éstos saldrán a la luz y perderá el respeto del grupo.

La gestión de personal es algo que debe basarse en la experiencia, en lugar de aprenderse en un libro. El objetivo de esta sección, y la siguiente sobre el trabajo en equipo, es introducir de manera sencilla algunos de los problemas más importantes de la gestión de personal y del equipo que afectan la gestión de proyectos de software. Se espera que este material lo sensibilice a algunos de los problemas que pueden encontrar los administradores cuando se enfrentan con equipos de individuos técnicamente talentosos.

22.2.1 Motivación del personal

Como administrador de proyecto, usted necesitará motivar a las personas con quienes trabaja, de manera que éstas contribuyan con lo mejor de sus habilidades. Motivación significa organizar el trabajo y el ambiente laboral para alentar a los individuos a desempeñarse tan efectivamente como sea posible. Si las personas no están motivadas, no estarán interesadas en la actividad que realizan. Así que trabajarán con lentitud, y será más probable que cometan errores y que no contribuyan con las metas más amplias del equipo o la organización.

Para fomentar este ánimo, hay que saber un poco acerca de qué motiva a la gente. Maslow (1954) sugiere que las personas se sienten motivadas para cubrir sus necesidades, las cuales se ordenan en una serie de niveles, como se muestra en la figura 22.7. Los niveles más bajos de esta jerarquía representan necesidades fundamentales de alimentación, sueño, etcétera, y la necesidad de sentirse seguro en un ambiente. Las necesidades sociales se relacionan con el hecho de sentirse parte de un grupo social. Las necesidades de estima representan

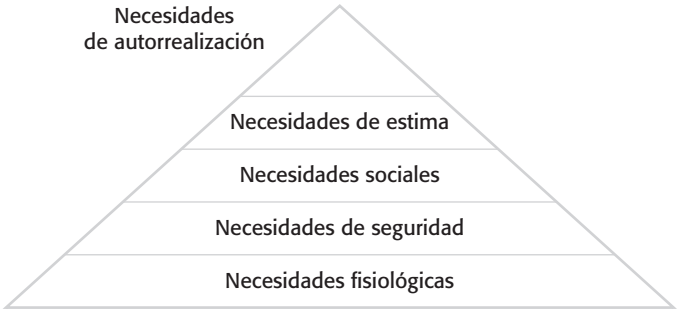


Figura 22.7 Jerarquía de necesidades humanas

la necesidad de sentirse respetado por otros, y las necesidades de autorrealización tienen que ver con el desarrollo personal. Las personas requieren cubrir las necesidades de nivel inferior, como el hambre, antes de las necesidades de nivel superior, que son más abstractas.

Las personas que trabajan en organizaciones de desarrollo de software, por lo general, no están hambrientas ni sedientas ni físicamente amenazadas por su ambiente. Por lo tanto, asegurarse de que se cubren las necesidades sociales, de estima y autorrealización de las personas es más importante desde un punto de vista administrativo.

1. Para satisfacer las necesidades sociales, es preciso dar a las personas tiempo para reunirse con sus compañeros de trabajo y proporcionarles lugares de socialización. Esto es relativamente sencillo cuando todos los miembros de un equipo de desarrollo trabajan en el mismo lugar. Aunque cada vez es más común que los miembros del equipo no laboren en el mismo edificio o ni siquiera en la misma ciudad o estado. Pueden trabajar para diferentes organizaciones o la mayor parte del tiempo desde casa.

   Los sistemas de redes sociales y las teleconferencias facilitan las comunicaciones, pero los sistemas electrónicos son más efectivos una vez que las personas se conocen entre sí. Por lo tanto, es necesario programar algunas reuniones cara a cara en etapas tempranas del proyecto, de manera que la gente pueda interactuar directamente con otros miembros del equipo. Mediante esta interacción directa, las personas se vuelven parte de un grupo social y aceptan las metas y prioridades de dicho grupo.
2. Para cubrir las necesidades de estima, es necesario demostrar a las personas que son valoradas por la organización. El reconocimiento público de los logros es una forma simple, aunque efectiva, de hacer esto. Desde luego, las personas también deben sentir que se les paga de acuerdo con sus habilidades y experiencia.
3. Finalmente, para cubrir las necesidades de autorrealización, es necesario dar responsabilidad a las personas por su trabajo, asignarles tareas demandantes (pero no imposibles) y ofrecer un programa de capacitación donde puedan desarrollar sus habilidades. La capacitación es una importante influencia motivadora, pues la mayoría de las personas desean adquirir nuevos conocimientos y aprender nuevas habilidades.

En la figura 22.8 se ilustra un problema de motivación que habitualmente deben enfrentar los administradores. En este ejemplo, un miembro competente del grupo

Estudio de caso: Motivación

Alice es administradora de un proyecto de software en una compañía que desarrolla sistemas de alarma. Esta compañía quiere ingresar al creciente mercado de tecnología de apoyo para ayudar a que los ancianos y las personas discapacitadas vivan de manera independiente. Se solicitó a Alice dirigir un equipo de seis desarrolladores que pueden diseñar nuevos productos basados en tecnología de alarma de la empresa.

El proyecto de tecnología de apoyo de Alice comienza bien. Dentro del equipo se desarrollan tanto buenas relaciones de trabajo como nuevas ideas creativas. El equipo decide desarrollar un sistema de mensajería entre pares usando televisores digitales vinculados a la red de alarma para comunicaciones. Sin embargo, meses después en el proyecto, Alice nota que Dorothy, una experta en diseño de hardware, comienza a llegar tarde al trabajo, que la calidad de su trabajo se deteriora y, cada vez más, no parece comunicarse con otros miembros del equipo.

Alice platica el problema informalmente con otros miembros del equipo para tratar de descubrir si cambiaron las circunstancias personales de Dorothy, y si esto pudiera afectar su trabajo. Como ninguno de ellos sabe algo al respecto, Alice decide hablar con Dorothy para tratar de entender el problema.

Después de negar al inicio que exista un problema, Dorothy admite que perdió interés en el trabajo. Ella esperaba desarrollar y usar sus habilidades de creación de interfaces de hardware. Sin embargo, debido a la gestión del producto que se eligió, tiene poca oportunidad de hacer esto. Básicamente, trabaja como programadora C con otros miembros del equipo.

Admite que aunque el trabajo es desafiante, ella está preocupada de no desarrollar sus habilidades de interfaces. Además, piensa que será difícil encontrar un trabajo relacionado con interfaces de hardware después de este proyecto. Puesto que no quiere alterar al equipo al revelar que piensa en el siguiente proyecto, decide que es mejor minimizar la conversación con ellos.

Figura 22.8
Motivación individual

pierde interés en el trabajo y en el grupo en su conjunto. La calidad de su trabajo declina y se vuelve inaceptable. Esta situación debe enfrentarse rápidamente. Si no se resuelve el problema, los otros miembros del grupo se sentirán insatisfechos y pensarán que están haciendo una parte del trabajo que no les corresponde.

En este ejemplo, Alice trata de descubrir si las circunstancias personales de Dorothy son el origen del problema. Comúnmente, las dificultades personales afectan la motivación, porque las personas no pueden concentrarse en su trabajo. Tal vez haya que darles tiempo y apoyo para resolver dichos conflictos, aunque también hay que dejar en claro que siguen teniendo una responsabilidad con su empleador.

El problema de motivación de Dorothy es bastante común cuando los proyectos se desarrollan en una dirección imprevista. Las personas que esperan hacer un tipo de trabajo pueden terminar por hacer algo completamente diferente. Esto se vuelve un problema cuando los miembros del equipo quieren desarrollar sus habilidades en una forma que es distinta al rumbo que tomó el proyecto. En tales circunstancias, el administrador podría decidir que un integrante debe abandonar el equipo y encontrar oportunidades en alguna otra parte. Sin embargo, en este ejemplo, Alice decide tratar de convencer a Dorothy de que ampliar su experiencia es un paso positivo en su carrera. Concede a Dorothy más autonomía de diseño y organiza cursos de capacitación en ingeniería de software que le darán más oportunidades después de terminado su proyecto actual.

Modelo de Madurez de Capacidad del Personal

El Modelo de Madurez de Capacidad del Personal (P-CMM) es un marco para valorar qué tan bien las organizaciones administran el desarrollo de su personal. Pone de relieve las mejores prácticas en la gestión de personal y ofrece una base para que las organizaciones mejoren los procesos de administración de los recursos humanos.

http://www.SoftwareEngineering-9.com/Web/Management/P-CMM.html

El modelo de motivación de Maslow es útil sólo hasta cierto punto, ya que adopta un punto de vista exclusivamente personal de la motivación. No considera de manera adecuada el hecho de que las personas se sienten parte de una organización, un grupo profesional y una o más culturas. Ésta no es tan sólo una cuestión de cubrir necesidades sociales: las personas pueden sentirse motivadas al ayudar a un grupo a lograr metas compartidas.

Ser miembro de un grupo cohesivo es enormemente motivador para la mayoría de la gente. Con frecuencia, las personas con trabajos satisfactorios disfrutan ir a trabajar, porque están motivadas por la gente con la que trabajan y por la actividad que realizan. Por lo tanto, además de pensar en la motivación individual, también hay que considerar cómo un grupo en su conjunto puede motivarse para lograr las metas de la organización. En la siguiente sección se estudian los conflictos de administrar grupos.

El tipo de personalidad también influye en la motivación. Bass y Dunteman (1963) clasifican a los profesionales en tres tipos:

1. Personas orientadas a las tareas, quienes están motivadas por el trabajo que realizan. En la ingeniería de software se trata de personas que están motivadas por el reto intelectual de desarrollar software.
2. Personas orientadas hacia sí mismas, quienes están motivadas principalmente por el éxito y el reconocimiento personales. Están interesadas en el desarrollo del software como un medio para lograr sus propias metas. Esto no significa que esos individuos sean egoístas y sólo piensen en sus propios intereses. En vez de ello, suelen tener metas a plazos más largos, como el avance profesional; de esta manera, se sienten motivados a tener éxito en su trabajo para conseguir dichas metas.
3. Personas orientadas a la interacción, quienes están motivadas por la presencia y las acciones de los compañeros de trabajo. Conforme el desarrollo de software se vuelve más centrado en el usuario, los individuos orientados a la interacción se involucran más en la ingeniería de software.

Las personas orientadas a la interacción comúnmente disfrutan al trabajar como parte de un grupo, mientras que quienes están orientadas a las tareas o hacia sí mismas prefieren actuar individualmente. Las mujeres tienen más probabilidad que los hombres de estar orientadas a la interacción. Con frecuencia son comunicadores más efectivos. En el estudio de caso de la figura 22.10 se muestra la mezcla de estos diferentes tipos de personalidad en grupos.

La motivación de cada individuo se constituye con elementos de cada clase, pero, por lo regular, un tipo de motivación domina en algún momento dado. Sin embargo, los individuos pueden cambiar. Por ejemplo, el personal técnico que siente que no se recompensa de manera adecuada puede volverse orientado hacia sí mismo y anteponer el interés personal a los asuntos técnicos. Si un grupo trabaja particularmente bien, las personas orientadas hacia sí mismas pueden volverse más orientadas a la interacción.

22.3 Trabajo en equipo

La mayor parte del software profesional se desarrolla mediante equipos de proyecto, cuyo número de miembros varía entre dos y varios cientos de personas. Sin embargo, como es imposible que todos los integrantes de un grupo grande trabajen en conjunto en un solo problema, los equipos grandes habitualmente se dividen en grupos más pequeños. Cada grupo es responsable de desarrollar parte del sistema global. Como regla general, los grupos del proyecto de ingeniería de software no deben tener más de 10 miembros. Cuando se usan grupos pequeños se reducen los problemas de comunicación. Todos conocen a todos los demás, y el grupo en su conjunto puede reunirse en torno a una mesa para estudiar el proyecto y el software que desarrollan.

Conformar un grupo que tiene el equilibrio justo de habilidades técnicas, experiencia y personalidades es una tarea administrativa fundamental. Sin embargo, los grupos exitosos son mucho más que una colección de individuos con el equilibrio justo de habilidades. Un buen equipo es cohesivo y tiene espíritu de grupo. Las personas que participan están motivadas tanto por el éxito del grupo como por sus metas personales.

En un grupo cohesivo, los miembros piensan que el equipo es más importante que los individuos que lo integran. Los miembros de un grupo cohesivo bien liderado son leales al equipo. Se identifican con las metas del grupo y con los demás miembros. Tratan de proteger al grupo, como entidad, de cualquier interferencia externa. Esto hace que el grupo sea sólido y pueda enfrentar problemas y situaciones inesperadas.

Los beneficios de crear un grupo cohesivo son:

1. *El grupo puede establecer sus propios estándares de calidad* Puesto que dichos estándares se establecen por consenso, éstos tienen más probabilidad de respetarse que los estándares externos impuestos sobre el grupo.
2. *Los individuos aprenden de los demás y se apoyan mutuamente* Las personas en el grupo aprenden de los demás. Las inhibiciones causadas por la ignorancia se minimizan mientras se promueve el aprendizaje mutuo.
3. *El conocimiento se comparte* Puede mantenerse la continuidad si sale un miembro del grupo. Otros en el grupo pueden tomar el control de las tareas críticas para asegurar que el proyecto no se altere en forma considerable.
4. *Se alientan la refactorización y el mejoramiento continuo* Los miembros del grupo trabajan de manera colectiva para entregar resultados de alta calidad y corregir problemas, sin importar quiénes crearon originalmente el diseño o programa.

Estudio de caso: Espíritu de equipo

Alice, una experimentada administradora de proyecto, comprende la importancia de crear un grupo cohesivo. Conforme se desarrolla un nuevo producto, ella aprovecha la oportunidad de hacer participar a todos los miembros del grupo en la especificación y el diseño del producto, al hacer que analicen las posibles tecnologías con los miembros de sus familias de mayor experiencia. Alice también los alienta a llevar a esos familiares para reunirse con otros integrantes del grupo de desarrollo.

También organiza almuerzos mensuales para todos los integrantes del grupo. Dichos almuerzos son una oportunidad para que todos los miembros del equipo se reúnan de manera informal, platiquen acerca de los temas que les preocupan y se conozcan mutuamente. En el almuerzo, Alice comunica al grupo lo que sabe acerca de las noticias, políticas, estrategias, etcétera, de la organización. Luego, cada miembro del equipo explica brevemente lo que hace, y el grupo examina un tema general, como las ideas de un nuevo producto de los parientes mayores.

Cada determinado tiempo, Alice organiza un día fuera para el grupo, en que el equipo pasa dos días en actualización tecnológica. Cada miembro del equipo prepara una actualización sobre una tecnología relevante y la presenta al grupo. Ésta es una reunión fuera de sitio en un buen hotel y se programa bastante tiempo para las discusiones y la interacción social.

Figura 22.9 Cohesión grupal

Los buenos administradores de proyecto siempre tratan de alentar la cohesión grupal. Pueden organizar eventos sociales para los miembros del grupo y sus familias, además de establecer un sentido de identidad al ponerle nombre al grupo y establecer una compatibilidad y un territorio grupales, u organizar actividades explícitas de construcción grupal, como actividades deportivas y juegos.

Una de las formas más efectivas de fomentar la cohesión es ser comprensivo. Esto significa que hay que tratar a los miembros de los grupos como responsables y confiables, y poner la información a libre disposición. En ocasiones, los administradores sienten que no pueden revelar cierta información a todos los miembros del grupo. Esto invariablemente crea un clima de desconfianza. El simple intercambio de información es una forma efectiva de hacer que las personas se sientan valoradas y se reconozcan como parte de un grupo.

En el estudio de caso de la figura 22.9 se puede ver un ejemplo de esto. Alice concierta reuniones informales regulares donde informa a los otros miembros del grupo lo que sucede. Se asegura de involucrar a las personas en el desarrollo del producto al pedirles que proporcionen nuevas ideas derivadas de su propia experiencia familiar. Los días fuera también son buenas formas de promover la cohesión: las personas se relajan al reunirse mientras se ayudan mutuamente a aprender nuevas tecnologías.

Si un grupo es efectivo o no, en cierta medida, depende de la naturaleza del proyecto y la organización que realiza el trabajo. Si una organización se encuentra en un estado de turbulencia por reorganizaciones constantes e inseguridad laboral, es muy difícil que los miembros del equipo se enfoquen en el desarrollo de software. Sin embargo, aparte de los conflictos del proyecto y la organización, existen tres factores genéricos que afectan el trabajo en equipo:

1. *Las personas en el grupo* Se necesita una combinación de personas en un grupo de proyecto, puesto que el desarrollo de software implica diversas actividades, como negociación con clientes, programación, pruebas y documentación.

2. *La organización grupal* Un grupo debe organizarse de forma que los individuos puedan contribuir con sus mejores habilidades y completar las tareas como se esperaba.

3. *Comunicaciones técnicas y administrativas* Es esencial la óptima comunicación entre los miembros del grupo, y entre el equipo de ingeniería de software y otras partes interesadas en el proyecto.

Como en todos los conflictos administrativos, reunir al equipo correcto no garantiza el éxito del proyecto. Muchas otras cosas pueden salir mal, incluidos los cambios en los negocios y en el ambiente empresarial. Sin embargo, si no se presta la atención debida a la composición, la organización y las comunicaciones del grupo, aumenta la probabilidad de que el proyecto enfrente dificultades.

22.3.1 Selección de los miembros del grupo

La labor de un administrador o líder de equipo es crear un grupo cohesivo y organizar a los miembros del grupo para que puedan trabajar en conjunto de manera efectiva. Esto implica crear un grupo con el equilibrio correcto de habilidades técnicas y personalidades, así como organizarlo para que los miembros trabajen adecuadamente en conjunto. En ocasiones, se contrata a personas externas a la organización; no obstante, con más frecuencia, los grupos de ingeniería de software se componen de empleados actuales que tienen experiencia adquirida en otros proyectos. Con todo, los administradores pocas veces tienen absoluta libertad en la selección del equipo. Con frecuencia deben recurrir a las personas que estén disponibles en la compañía, aun cuando no sean ideales para el puesto.

Como se estudió en la sección 22.2.1, muchos ingenieros de software están motivados principalmente por su trabajo. Por lo tanto, los grupos de desarrollo a menudo están compuestos por personas que cuentan con ideas propias sobre qué problemas técnicos deben resolverse. Esto se refleja en los problemas que se reportan regularmente relacionados con estándares de interfaz ignorados, sistemas rediseñados conforme se codifican, arreglo innecesario del sistema, etcétera.

Un grupo con personalidades complementarias puede trabajar mejor que un grupo seleccionado exclusivamente por la habilidad técnica. Es probable que las personas que están motivadas por el trabajo sean las más fuertes técnicamente. Las personas que son orientadas hacia sí mismas tal vez serán mejores para impulsar el trabajo hacia delante para terminar la tarea. Las personas orientadas a la interacción ayudan a facilitar las comunicaciones dentro del grupo. Se considera que en el grupo es particularmente importante contar con personas orientadas a la interacción. A éstas les gusta hablar con los demás y son capaces de detectar tensiones y diferencias en una etapa temprana, antes de que tengan serias repercusiones sobre el grupo.

En el estudio de caso de la figura 22.10, se narró cómo Alice, la administradora del proyecto, trató de crear un grupo con personalidades complementarias. Este grupo particular tiene una buena combinación de personas orientadas a la interacción y a las tareas, pero, en la figura 22.8, ya se describió cómo la personalidad de Dorothy, quien está orientada hacia sí misma, causó problemas porque no realizó el trabajo esperado. También el rol de tiempo parcial de Fred, como experto de dominio, podría ser un problema del grupo. Él está principalmente interesado en los retos técnicos, así que posible-

Estudio de caso: Composición de grupo

Al crear un grupo para el desarrollo de tecnología de apoyo, Alice está consciente de la importancia de seleccionar miembros con personalidades complementarias. Cuando entrevista a miembros potenciales del grupo, trata de valorar si están orientados a las tareas, hacia sí mismos u orientados a la interacción. Ella siente que su personalidad está orientada hacia sí misma, porque considera que el proyecto es una forma de hacerse notar ante los altos ejecutivos y buscar una promoción. Por ende, busca una o quizá dos personalidades orientadas a la interacción, e individuos orientados a las tareas para completar el equipo. La valoración final a la que llegó fue:

Alice: orientada hacia sí misma
Brian: orientado a tareas
Bob: orientado a tareas
Carol: orientada a la interacción
Dorothy: orientada hacia sí misma
Ed: orientado a la interacción
Fred: orientado a tareas

Figura 22.10 Composición de grupo

mente no interactúe bien con otros miembros del grupo. El hecho de que no siempre es parte del equipo significa que puede no relacionarse bien con las metas del equipo.

En ocasiones es imposible elegir un grupo con personalidades complementarias. Si éste es el caso, el administrador del proyecto tiene que controlar al grupo de modo que las metas individuales no se antepongan a los objetivos de la organización y del grupo. Este control es más sencillo de lograr si todos los miembros del grupo participan en cada etapa del proyecto. La iniciativa individual es más factible cuando los miembros del grupo reciben instrucciones sin estar al tanto de la parte que desempeña su tarea en el proyecto global.

Por ejemplo, suponga que a un ingeniero de software se le asigna un diseño de programa para codificar, y observa lo que parecen ser posibles mejoras que podrían hacerse al diseño. Si implementa dichas mejoras sin comprender las razones del diseño original, cualquier cambio, aun cuando sea muy bien intencionado, puede tener implicaciones adversas para otras partes del sistema. Si todos los miembros del grupo participan en el diseño desde el principio, comprenderán por qué se tomaron las decisiones de diseño. Entonces, los miembros podrán identificarse con dichas decisiones en lugar de oponerse a ellas.

22.3.2 Organización del grupo

La forma en que se organiza un grupo influye en las decisiones que toma dicho grupo, las maneras como se intercambia la información y las interacciones entre el grupo de desarrollo y los participantes externos del proyecto. Las preguntas organizacionales importantes para los administradores de proyecto incluyen:

1. ¿El administrador del proyecto debe ser el líder técnico del grupo? El líder técnico o arquitecto del sistema es responsable de las decisiones técnicas críticas tomadas durante el desarrollo del software. En ocasiones, el administrador del proyecto tiene

Contratar a las personas correctas

Con frecuencia los administradores del proyecto son responsables de seleccionar al personal en la organización que se unirá a su equipo de ingeniería de software. Conseguir a las mejores personas posibles en este proceso es muy importante, pues las malas decisiones de selección implican un grave riesgo para el proyecto.

Los factores clave que deben influir en la selección de personal son: educación y capacitación, dominio de aplicación y experiencia tecnológica, habilidad de comunicación, adaptabilidad y habilidad para resolver problemas.

http://www.SoftwareEngineering-9.com/Web/Management/Selection.html

la habilidad y experiencia para desempeñar este papel. Sin embargo, en caso de grandes proyectos, es mejor asignar a un ingeniero con experiencia como el arquitecto del proyecto, quien tomará la responsabilidad del liderazgo técnico.

2. ¿Quién se encargará de tomar las decisiones técnicas críticas, y cómo se tomarán? ¿Las decisiones las tomará el arquitecto del sistema, el administrador del proyecto o se llegará a un consenso entre un rango más amplio de miembros del equipo?

3. ¿Cómo se manejarán las interacciones con los participantes externos y los altos directivos de la compañía? En muchos casos, el administrador del proyecto será el responsable de dichas interacciones, asistido, si acaso, por el arquitecto del sistema. Sin embargo, un modelo de organización alternativo incluye una función exclusiva para las relaciones externas, lo que supone asignar para dicha función a una persona con habilidades de interacción adecuadas.

4. ¿Cómo es posible que los grupos logren integrar a personas que no se localizan en el mismo lugar? Ahora es común que los grupos incluyan a miembros de diferentes organizaciones y personas que trabajan desde casa o en oficinas compartidas. Esto debe tomarse en cuenta en los procesos de toma de decisiones grupales.

5. ¿Cómo puede compartirse el conocimiento a través del grupo? La organización del grupo afecta el intercambio de información, pues determinadas formas de organización son mejores que otras para compartir. Sin embargo, conviene evitar demasiado intercambio de información, ya que las personas pueden sobresaturarse y la información excesiva podría distraerlos de sus labores.

Los grupos de programación pequeños, por lo general, están organizados en una forma bastante informal. El líder del grupo participa en el desarrollo de software con los otros miembros del grupo. En un grupo informal, todo el equipo analiza el trabajo a realizar, y las tareas se asignan según la habilidad y la experiencia. Los miembros del grupo con mayor jerarquía pueden ser responsables del diseño arquitectónico. No obstante, el diseño y la implementación detallados son compromisos del miembro del equipo que se asigna a una tarea particular.

Los grupos de programación extrema (Beck, 2000) siempre son grupos informales. Los apasionados de XP afirman que la estructura formal inhibe el intercambio de información. En XP, muchas decisiones que normalmente se consideran como decisiones administrativas (por ejemplo, las decisiones acerca del calendario) se delegan a los

miembros del grupo. Los programadores trabajan en pares para diseñar un código y asumen responsabilidad conjunta de los programas que desarrollaron.

Los grupos informales pueden ser muy exitosos, en particular cuando la mayoría de los miembros del grupo son experimentados y competentes. Tal grupo toma decisiones por consenso, lo que mejora la cohesión y el rendimiento. Sin embargo, si un grupo está compuesto principalmente por miembros inexpertos e incompetentes, la informalidad puede ser un obstáculo porque no existe autoridad definida para dirigir el trabajo, lo que causa una falta de coordinación entre los miembros del grupo y, posiblemente, una eventual falla del proyecto.

Los grupos jerárquicos son grupos que comparten una estructura jerárquica con el líder del grupo en la parte superior del escalafón. El líder tiene autoridad más formal que los miembros del grupo y así puede dirigir el trabajo. Existe una clara estructura organizacional, y las decisiones se toman hacia la parte superior de la jerarquía y se aplican por las personas que están más abajo en la jerarquía. Las comunicaciones, ante todo, son instrucciones del personal ejecutivo y existe relativamente poca comunicación ascendente, es decir, desde los niveles más bajos hacia los niveles superiores en la jerarquía.

Este enfoque funciona bien cuando un problema bien entendido puede descomponerse fácilmente en subproblemas en los que las soluciones se desarrollan en diferentes partes de la jerarquía. En dichas situaciones se requiere muy poca comunicación a través de la jerarquía. Sin embargo, tales situaciones, en proporción, son poco comunes en la ingeniería de software por las siguientes razones:

1. Los cambios al software requieren con frecuencia cambios en varias partes del sistema y esto conduce a una discusión y negociación en todos los niveles de la jerarquía.
2. Las tecnologías de software cambian tan rápido que muchas veces el personal más joven conoce más de la tecnología que el personal experimentado. Las comunicaciones descendentes pueden significar que el administrador del proyecto no vislumbra las oportunidades de usar nuevas tecnologías. El personal más joven puede frustrarse debido a que considera obsoletas las tecnologías usadas para el desarrollo.

Las organizaciones grupales democráticas y jerárquicas no reconocen formalmente que puede haber diferencias muy grandes de habilidad técnica entre los miembros del grupo. Los mejores programadores pueden ser hasta 25 veces más productivos que los peores programadores. Tiene sentido aprovechar las capacidades de los mejores elementos en la forma más efectiva y brindarles tanto apoyo como sea posible. Uno de los primeros modelos organizacionales que tenía la intención de ofrecer apoyo fue el llamado equipo programador jefe.

Para aprovechar de manera más efectiva a los programadores con mayor habilidad, Baker (1972) y otros (Aron, 1974; Brooks, 1975) sugieren que los equipos deben construirse en torno a un programador jefe individual con gran habilidad. El principio subyacente del equipo programador jefe es que el personal habilidoso y experimentado debe ser responsable de todo el desarrollo del software. Sus integrantes no deben preocuparse por cuestiones rutinarias y deben tener buen apoyo técnico y administrativo para realizar su trabajo. Deben enfocarse en el software a desarrollar y no perder mucho tiempo en reuniones externas.

El ambiente laboral físico

El ambiente donde trabajan las personas afecta tanto las comunicaciones grupales como la productividad individual. Los espacios de trabajo individuales son mejores para la concentración en el trabajo técnico detallado, pues las personas tienen menos probabilidad de distraerse por interrupciones. Sin embargo, los espacios de trabajo compartidos son mejores para las comunicaciones. Un ambiente laboral bien diseñado toma en consideración ambas necesidades.

http://www.SoftwareEngineering-9.com/Web/Management/workspace.html

No obstante, la organización en equipo programador jefe es, desde la perspectiva del autor, demasiado dependiente del programador en jefe y su asistente. Otros miembros del equipo a quienes no se dé suficiente responsabilidad pueden desmotivarse porque sienten que sus habilidades son desaprovechadas. No tienen la información para hacer frente si las cosas salen mal y no se les da la oportunidad de participar en la toma de decisiones. Existen riesgos significativos para el proyecto asociados con esta organización grupal y esto podría superar cualquier beneficio que aporte este tipo de organización.

22.3.3 Comunicaciones grupales

Es absolutamente esencial que los miembros del grupo se comuniquen efectiva y eficientemente entre sí y con otras partes interesadas en el proyecto. Los miembros del grupo deben intercambiar información acerca del estatus de su trabajo, las decisiones de diseño que se tomaron y los cambios a las decisiones de diseño previas. Tienen que resolver los problemas que surjan con otros interesados en el proyecto e informar a éstos sobre los cambios al sistema, grupo y planes de entrega. La buena comunicación ayuda también a fortalecer la cohesión del grupo. Los miembros del grupo llegan a entender las motivaciones, fortalezas y debilidades de otras personas en el grupo.

La efectividad y la eficiencia de las comunicaciones están influidas por:

1. *Tamaño del grupo* Conforme el grupo crece, se hace más difícil que los miembros se comuniquen de manera efectiva. El número de vínculos de comunicación de un canal es $n * (n - 1)$, donde n es el tamaño del grupo, de manera que, con un grupo de ocho miembros, existen 56 posibles rutas de comunicación. Esto significa que es muy posible que algunas personas rara vez se comuniquen entre sí. Las diferencias de estatus entre los miembros del grupo significan que las comunicaciones con frecuencia son unidireccionales. Los administradores e ingenieros experimentados tienden a dominar las comunicaciones con el personal menos experimentado, quienes pueden tener reticencias para iniciar una conversación o hacer puntualizaciones críticas.
2. *Estructura del grupo* Las personas en los grupos estructurados de manera informal se comunican más efectivamente que los individuos en grupos con una estructura jerárquica formal. En los grupos jerárquicos, las comunicaciones tienden a fluir hacia arriba y abajo de la jerarquía. Las personas en el mismo nivel tal vez no se

comuniquen entre sí. Éste es un problema particular en un proyecto grande con varios grupos de desarrollo. Si las personas que trabajan en diferentes subsistemas se comunican sólo a través de sus administradores, hay más probabilidad de demoras y malas interpretaciones.

3. *Composición del grupo* Las personas con los mismos tipos de personalidad (estudiados en la sección 22.2) pueden chocar y, como resultado, las comunicaciones se inhiben. Además, por lo regular, la comunicación es mejor en los grupos integrados por personas de uno y otro género (Marshall y Heslin, 1975) que en los grupos formados por miembros de un solo género. Con frecuencia, las mujeres son más orientadas a la interacción que los hombres y suelen actuar como controladoras y facilitadoras de la interacción para el grupo.

4. *El ambiente laboral físico* La organización del centro de trabajo es un factor importante para facilitar o inhibir las comunicaciones. Véase la página Web del libro para más información.

5. *Los canales de comunicación disponibles* Existen muchas formas diferentes de comunicación: cara a cara, correo electrónico, documentos formales, teléfono y tecnologías Web 2.0, como las redes sociales y los wikis. Conforme los equipos de proyecto se distribuyen cada vez más, con miembros de equipo que trabajan en lugares remotos, es necesario utilizar varias tecnologías para facilitar las comunicaciones.

Los administradores de proyecto trabajan por lo general bajo plazos estrechos y, en consecuencia, tratan de usar canales de comunicación que no consuman mucho tiempo. Por lo tanto, se apoyan en reuniones y documentos formales para transmitir la información al personal del proyecto y las partes interesadas. Aunque éste tal vez sea un enfoque eficiente para la comunicación desde la perspectiva de un administrador de proyecto, comúnmente no es muy efectivo. A menudo existen buenas razones por las que las personas no pueden asistir a las reuniones y, por lo tanto, no escuchan la presentación. Los documentos extensos casi nunca se leen, debido a que los lectores no saben si los documentos son relevantes. Al producirse varias versiones del mismo documento, los lectores encuentran difícil hacer un seguimiento de los cambios.

La comunicación efectiva se logra cuando las comunicaciones son bidireccionales, y las personas implicadas pueden discutir los conflictos y la información, y establecer una comprensión común de las proposiciones y los problemas. Esto se lograr mediante reuniones, aunque éstas suelen estar dominadas por las personalidades poderosas. En ocasiones no es práctico citar con escasa anticipación a reuniones. Cada vez más equipos de proyecto incluyen miembros ubicados a distancia, lo que dificulta las reuniones.

Para contrarrestar estos problemas y apoyar el intercambio de información, se puede recurrir a tecnologías Web, como wikis y blogs. Los wikis respaldan la creación y edición de documentos en colaboración, mientras que los blogs apoyan las discusiones generadas por preguntas y comentarios hechos por los miembros del grupo. Los wikis y blogs permiten a los miembros del proyecto y a los participantes externos intercambiar información, sin importar su ubicación. Ayudan a gestionar la información y a seguir la huella de los hilos de discusión, que con frecuencia se vuelven confusos cuando se realizan por correo electrónico. Para resolver conflictos que necesiten discusión, se puede recurrir a la mensajería instantánea y las teleconferencias, las cuales pueden organizarse fácilmente.

PUNTOS CLAVE

- La buena gestión de proyectos de software es esencial si los proyectos de ingeniería de software deben desarrollarse dentro del plazo y el presupuesto establecidos.
- La gestión del software es distinta de otras administraciones de ingeniería. El software es intangible. Los proyectos pueden ser novedosos o innovadores, así que no hay un conjunto de experiencias para orientar su gestión. Los procesos de software no son tan maduros como los procesos de ingeniería tradicionales.
- La gestión del riesgo se reconoce ahora como una de las tareas más importantes de la gestión de un proyecto.
- La gestión del riesgo implica la identificación y valoración de los grandes riesgos del proyecto para establecer la probabilidad de que ocurran; también supone identificar y valorar las consecuencias para el proyecto si dicho riesgo surge. Debe hacer planes para evitar, gestionar o enfrentar los posibles riesgos.
- Las personas se sienten motivadas por la interacción con otros individuos, el reconocimiento de la gestión y sus pares, y al recibir oportunidades de desarrollo personal.
- Los grupos de desarrollo de software deber ser bastante pequeños y cohesivos. Los factores clave que influyen en la efectividad de un grupo son sus integrantes, la forma en que está organizado y la comunicación entre los miembros.
- Las comunicaciones dentro de un grupo están influidas por factores como el estatus de los miembros del grupo, el tamaño del grupo, la composición por género del grupo, las personalidades y los canales de comunicación disponibles.

LECTURAS SUGERIDAS

The Mythical Man Month (Anniversary Edition). Los problemas de la gestión del software siguen en gran medida invariables desde la década de 1960, y este libro es uno de los mejores sobre el tema. Una interesante y clara explicación de la gestión de uno de los primeros y más grandes proyectos de software: el sistema operativo OS/360 de IBM. La edición de aniversario (publicada 20 años después de la edición original de 1975) incluye otros ensayos clásicos de Brooks. (F. P. Brooks, 1995, Addison-Wesley.)

Software Project Survival Guide. Ésta es una explicación bastante pragmática de la gestión del software que incluye buenos consejos prácticos para los administradores de proyecto con antecedentes de ingeniería de software. Es fácil de leer y entender. (S. McConnell, 1998, Microsoft Press.)

Peopleware: Productive Projects and Teams, 2nd edition. Ésta es una nueva edición del libro clásico acerca de la importancia de tratar a las personas de manera adecuada cuando se administran proyectos de software. Es uno de los pocos libros que reconocen la importancia del lugar donde trabajan las personas. Enormemente recomendable. (T. DeMarco y T. Lister, 1999, Dorset House.)

Waltzing with Bears: Managing Risk on Software Projects. Una introducción muy práctica y fácil de leer respecto a los riesgos y la gestión del riesgo. (T. DeMarco y T. Lister, 2003, Dorset House.)

EJERCICIOS

22.1. Explique por qué la intangibilidad de los sistemas de software plantea problemas especiales para la gestión de proyectos de software.

22.2. Explique por qué los mejores programadores no siempre son los mejores administradores de software. Tal vez le resulte útil basar su respuesta en la lista de actividades administrativas de la sección 22.1.

22.3. Con los casos de problemas de proyecto reportados en la literatura, mencione las dificultades y los errores administrativos que ocurrieron en dichos proyectos de programación fallidos. (Se sugiere que comience con *The Mythical Man Month*, de Fred Brooks).

22.4. Además de los riesgos que se muestran en la figura 22.1, identifique al menos otros seis riesgos posibles que pudieran surgir en los proyectos de software.

22.5. Los contratos de precio fijo, donde el contratista ofrece un precio fijo para completar un desarrollo de sistema, permiten desplazar los riesgos del proyecto del cliente al contratista. Si algo sale mal, el contratista debe pagar. Sugiera cómo el uso de tales contratos puede aumentar la probabilidad de que surjan riesgos del producto.

22.6. Explique por qué mantener informados a todos los miembros de un grupo acerca del progreso y de las decisiones técnicas en un proyecto ayuda a mejorar la cohesión del grupo.

22.7. ¿Qué problemas considera que surgirían en los equipos de programación extrema, donde muchas decisiones administrativas se delegan en los miembros del equipo?

22.8. Escriba un estudio de caso, con el estilo usado aquí, para ilustrar la importancia de las comunicaciones en un equipo de proyecto. Suponga que algunos miembros del equipo trabajan a distancia y no es posible reunir a todo el equipo en el corto plazo.

22.9. Su administrador le pide entregar software en un plazo que sólo podrá cumplir si pide a su equipo de proyecto trabajar tiempo extra sin remuneración. Todos los miembros del equipo tienen hijos pequeños. Discuta si debe aceptar esta demanda de su administrador o si debe persuadir a su equipo a ceder su tiempo a la organización en lugar de dedicarlo a sus familias. ¿Qué factores pueden ser significativos en su decisión?

22.10. Como programador, se le ofrece una promoción a un puesto de gestión de proyectos, pero siente que puede hacer una aportación más efectiva en un papel técnico más que administrativo. Discuta si debe aceptar la promoción.

REFERENCIAS

Aron, J. D. (1974). *The Program Development Process*. Reading, Mass.: Addison-Wesley.

Baker, F. T. (1972). "Chief Programmer Team Management of Production Programming". *IBM Systems J.*, **11** (1), 56–73.

Bass, B. M. y Dunteman, G. (1963). "Behaviour in groups as a function of self, interaction and task orientation". *J. Abnorm. Soc. Psychology*., **66** (4), 19–28.

Beck, K. (2000). *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, **21** (5), 61–72.

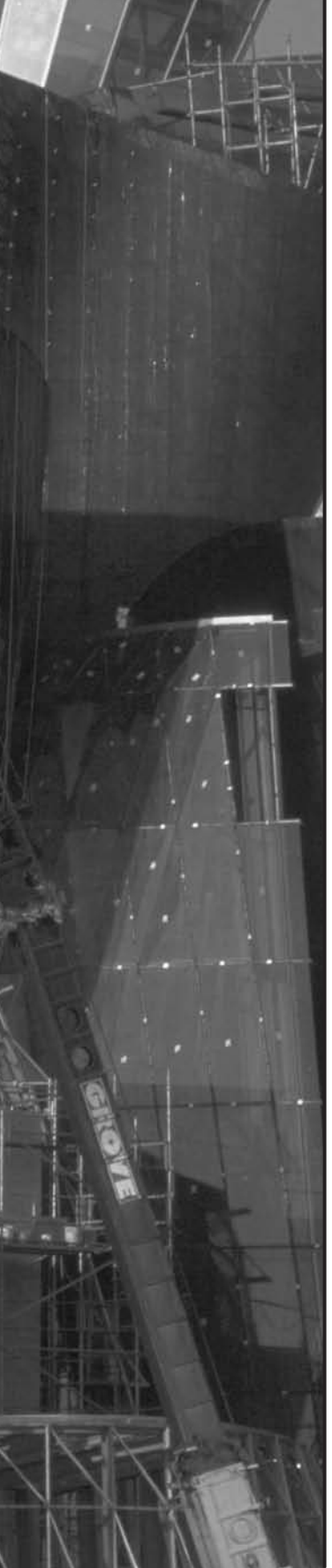
Brooks, F. P. (1975). *The Mythical Man Month*. Reading, Mass.: Addison-Wesley.

Hall, E. (1998). *Managing Risk: Methods for Software Systems Development*. Reading, Mass.: Addison-Wesley.

Marshall, J. E. y Heslin, R. (1975). "Boys and Girls Together. Sexual composition and the effect of density on group size and cohesiveness". *J. of Personality and Social Psychology*, **35** (5), 952–61.

Maslow, A. A. (1954). *Motivation and Personality*. Nueva York: Harper and Row.

Ould, M. (1999). *Managing Software Quality and Business Risk*. Chichester: John Wiley & Sons.



23

Planeación de proyectos

Objetivos

El objetivo de este capítulo es introducirlo a la planeación, calendarización y estimación de costos de proyectos. Al estudiar este capítulo:

- comprenderá los fundamentos del costo del software y las razones por las que el precio del software puede no relacionarse directamente con el costo de desarrollo;
- conocerá qué secciones debe incluir un plan de proyecto creado dentro de un proceso de desarrollo dirigido por un plan;
- entenderá lo que se contempla en la calendarización del proyecto y el uso de gráficas de barras para presentar un calendario de proyecto;
- se introducirá al "juego de planeación", utilizado para apoyar la planeación de proyectos en la programación extrema;
- analizará cómo puede usarse el modelo COCOMO II para la estimación algorítmica de costos.

Contenido

23.1 Fijación de precio al software
23.2 Desarrollo dirigido por un plan
23.3 Calendarización de proyectos
23.4 Planeación ágil
23.5 Técnicas de estimación

La planeación de proyectos es una de las labores más importantes de un administrador de proyectos de software. Como administrador, debe dividir el trabajo en partes y asignar éstas a los miembros del equipo del proyecto, anticipar los problemas que pudieran surgir y preparar posibles soluciones a tales inconvenientes. El plan creado al comienzo de un proyecto se usa para comunicar al equipo y los clientes cómo se realizará el trabajo, así como para ayudar a valorar el avance del proyecto.

La planeación se presenta durante tres etapas en un ciclo de vida del proyecto:

1. En la etapa de propuestas, cuando se presenta una licitación con vistas a obtener un contrato para desarrollar o proporcionar un sistema de software. En esta etapa es necesario un plan para ayudarle a decidir si cuenta con los recursos para completar el trabajo y a calcular el precio que debe cotizar al cliente.
2. Durante la fase de inicio, cuando debe determinar quién trabajará en el proyecto, cómo se dividirá el proyecto en incrementos, cómo se asignarán los recursos a través de su compañía, etcétera. Aquí, se cuenta con más información que en la etapa de la propuesta y, por lo tanto, se pueden afinar las estimaciones iniciales.
3. Periódicamente a lo largo del proyecto, cuando el plan se modifica a la luz de la experiencia obtenida y la información del monitoreo del avance del trabajo. Se aprende más acerca del sistema a implementar y de las capacidades del equipo de desarrollo. Esta información permite hacer estimaciones más precisas sobre cuánto tardará el trabajo. Más aún, es probable que los requerimientos del software cambien y esto, por lo general, significa que debe modificarse la división del trabajo y extenderse el plazo. Para proyectos de desarrollo tradicionales, quiere decir que el plan creado durante la fase de inicio debe modificarse. No obstante, cuando se usa un enfoque ágil, los planes son a plazo más corto y varían continuamente a medida que evoluciona el software. En la sección 23.4 se estudia la planeación ágil.

La planeación en la etapa de propuesta, inevitablemente, es especulativa, pues muchas veces no se cuenta con un conjunto completo de requerimientos para el software a desarrollar. En vez de ello, hay que responder a una solicitud de propuestas basada en una descripción de alto nivel sobre la funcionalidad del software que se requiere. Con frecuencia, se necesita un plan como parte de una propuesta, así que se debe diseñar un plan creíble para realizar el trabajo. Si se obtiene el contrato, entonces por lo general habrá que replantear el proyecto y tomar en cuenta los cambios desde que se hizo la propuesta.

Cuando se presenta una licitación para obtener un contrato, hay que calcular el precio que se propondrá al cliente para el desarrollo del software. Como punto de partida para calcular este precio, se requiere presentar una estimación de los costos para completar el trabajo del proyecto. La estimación incluye calcular cuánto esfuerzo se requiere para terminar cada actividad y, a partir de ello, calcular el costo total de las actividades. Siempre habrá que calcular los costos del software de manera objetiva, con la finalidad de predecir con precisión el costo para el desarrollo del software. Una vez que se tiene una estimación razonable de los probables costos, entonces es posible calcular el precio que se cotizará al cliente. Como se estudia en la siguiente sección, muchos factores influyen en la fijación del precio de un proyecto de software, no se trata simplemente de sumar el costo y la ganancia.

Costos generales

Al estimar los costos del esfuerzo de un proyecto de software, no sólo se multiplican los sueldos del personal involucrado por el tiempo invertido en el proyecto, sino que también hay que tener en cuenta todos los costos generales de la organización (espacio de oficinas, administración, etcétera) que deben cubrirse con el ingreso del proyecto. Los costos se estiman al calcular dichos costos generales y sumar una proporción a los costos de cada ingeniero que trabaja en un proyecto.

http://www.SoftwareEngineering-9.com/Web/Planning/overheadcosts.html

Existen tres principales parámetros que se deben usar al calcular los costos de un proyecto de desarrollo de software:

- Costos de esfuerzo (los costos de pagar a los ingenieros y administradores de software);
- costos de hardware y software, incluido el mantenimiento;
- costos de viajes y capacitación.

Para la mayoría de los proyectos, el mayor costo es el primer rubro. Debe estimarse el esfuerzo total (en meses-hombre) que es probable se requiera para completar el trabajo de un proyecto. Desde luego, se cuenta con datos limitados para realizar tal valoración, de manera que habrá que hacer la mejor evaluación posible y a continuación agregar contingencia significativa (tiempo y esfuerzo adicionales) en caso de que la estimación inicial sea optimista.

Para sistemas comerciales a menudo se usa hardware commodity, que es relativamente barato. Sin embargo, los costos de software pueden ser considerables si se debe licenciar el middleware y el software de plataforma. Es posible que se requieran viajes frecuentes cuando un proyecto se desarrolla en diferentes lugares. Aunque los costos de viaje, por lo regular, representan una pequeña fracción de los costos de esfuerzo, el tiempo invertido en viajar se desperdicia muchas veces y se agrega significativamente a los costos de esfuerzo del proyecto. Los sistemas de reunión electrónicos y otro software que apoya la colaboración remota pueden reducir la cantidad de viajes requeridos. El tiempo que se ahorra puede dedicarse a un trabajo del proyecto más productivo.

Una vez asignado un contrato para desarrollar un sistema, debe afinarse un bosquejo de plan de proyecto para crear un plan de inicio de proyecto. En esta etapa se debe saber más acerca de los requerimientos para este sistema. Sin embargo, es posible que no se cuente con una especificación completa de los requerimientos, en especial cuando se usa un enfoque ágil para el desarrollo. Su meta durante esta etapa debe ser elaborar un plan de proyecto que pueda usarse para apoyar la toma de decisiones acerca del personal y el presupuesto del proyecto. El plan sirve como base para asignar recursos al proyecto desde el interior de la organización y decidir si se requiere la contratación de nuevo personal.

El plan debe definir también los mecanismos de monitorización del proyecto. Es necesario hacer un seguimiento del avance del proyecto y comparar los avances y costos reales con el progreso planeado. Aunque la mayoría de las organizaciones tienen procedimientos formales para monitorizar, un buen administrador debe ser capaz de crear una

imagen clara de lo que sucede mediante comunicaciones informales con el personal del proyecto. La monitorización informal ayuda a pronosticar problemas potenciales de proyecto al revelar las dificultades conforme ocurren. Por ejemplo, los intercambios diarios con el personal del proyecto pueden revelar un problema particular para encontrar una falla del software. En vez de esperar el reporte de reducción del plazo, el administrador del proyecto podrá entonces asignar de inmediato un experto al problema, o decidir programar en torno a éste.

El plan del proyecto siempre evoluciona durante el proceso de desarrollo. La planeación del desarrollo pretende garantizar que el plan del proyecto siga siendo un documento útil para que el personal comprenda lo que debe lograrse y cuándo debe entregarse. Por lo tanto, el calendario y la estimación de costos y de riesgos deben revisarse a medida que se desarrolla el software.

Si se usa un método ágil, hay necesidad de un plan de inicio del proyecto pues, sin importar el enfoque empleado, la compañía necesita planear cómo se asignarán los recursos a un proyecto. Sin embargo, éste no es un plan detallado y sólo debe incluir información limitada sobre la división del trabajo y el calendario del proyecto. Durante el desarrollo, para cada entrega (*release*) del software, se define un plan de proyecto informal y las estimaciones del esfuerzo; todo el equipo participa en el proceso de planeación.

23.1 Fijación de precio al software

En principio, el precio de un producto de software a un cliente es simplemente el costo del desarrollo más las ganancias para el diseñador. Sin embargo, en la práctica, la relación entre el costo del proyecto y el precio cotizado al cliente no es tan simple. Cuando se calcula un precio, hay que hacer consideraciones más amplias de índole organizacional, económica, política y empresarial, como las mostradas en la figura 23.1. Debe pensarse en los intereses de la empresa, los riesgos asociados con el proyecto y el tipo de contrato que se firmará. Esto puede hacer que el precio se ajuste al alza o baja. Dadas las precisiones organizacionales implicadas, decidir sobre un precio de proyecto debe ser una actividad grupal que incluya al personal de marketing y ventas, así como altos ejecutivos y administradores de proyecto.

Para ilustrar algunos de los conflictos de fijación de precio del proyecto, tome en cuenta el siguiente escenario:

> *Una pequeña compañía de software, PharmaSoft, emplea a 10 ingenieros de software. Recientemente terminó un gran proyecto, pero sólo tiene contratos que requieren cinco personas de desarrollo. Sin embargo, licita para un contrato muy grande con una gran compañía farmacéutica que requiere 30 años-hombre de esfuerzo durante dos años. El proyecto no comenzará sino hasta dentro de 12 meses pero, si se realiza, transformará las finanzas de la compañía.*

> *PharmaSoft tiene una oportunidad para licitar en un proyecto que requiere seis personas y debe completarse en 10 meses. Los costos (incluidos los costos generales de este proyecto) se estiman en $1.2 millones. No obstante, para mejorar su posición competitiva, Pharmasoft decide ofrecer un precio al cliente de $0.8 millones.*

Factor	Descripción
Oportunidad de mercado	Una organización de desarrollo puede cotizar un precio bajo porque quiere moverse hacia un nuevo segmento del mercado de software. Aceptar una baja ganancia en un proyecto puede dar a la organización la oportunidad de obtener una mayor ganancia más adelante. La experiencia alcanzada podría ayudarle también a desarrollar nuevos productos.
Incertidumbre de estimación de costo	Si una organización no está segura de sus estimaciones de costos, puede aumentar su precio mediante una contingencia por arriba de su ganancia normal.
Términos contractuales	Un cliente puede permitir al desarrollador detener la propiedad del código fuente y reutilizarlo en otros proyectos. Entonces el precio podrá ser inferior al que se cobra si el código fuente se entrega al cliente.
Volatilidad de requerimientos	Si es probable que cambien los requerimientos, una organización puede reducir su precio para ganar un contrato. Una vez otorgado el contrato pueden cobrarse precios altos por cambios a los requerimientos.
Salud financiera	Los desarrolladores en dificultad financiera pueden reducir sus costos para obtener un contrato. Es mejor obtener una ganancia menor que lo normal o quedar en un punto de equilibrio, que salir del negocio. El flujo de efectivo es más importante que la ganancia en tiempos de problemas económicos.

Figura 23.1 Factores que afectan la fijación de precio del software

Esto significa que, aunque pierda dinero en este contrato, podrá retener al personal especializado para proyectos futuros más rentables que, probablemente, llegarán a raudales dentro de un año.

Como el costo de un proyecto sólo está débilmente relacionado con el precio cotizado a un cliente, "cotizar para ganar" es una estrategia usada comúnmente. Cotizar para ganar significa que una compañía tiene alguna idea del precio que el cliente espera pagar y hace una apuesta por el contrato con base en el precio esperado por el cliente. Esto puede parecer no ético y poco práctico en los negocios, pero tiene ventajas tanto para el cliente como para el proveedor del sistema.

Un costo de proyecto se acuerda sobre la base de un borrador de propuesta. Entonces las negociaciones tienen lugar entre cliente y consumidor para establecer la especificación detallada del proyecto. A esta especificación la restringe el costo acordado. El comprador y el vendedor deben convenir cuál es la funcionalidad aceptable del sistema. En muchos proyectos el factor fijo no está constituido por los requerimientos del proyecto, sino por el costo. Los requerimientos pueden cambiar para que el costo no se supere.

Por ejemplo, suponga que una compañía (OilSoft) licita por un contrato con el propósito de desarrollar un sistema de reparto de combustible para una compañía petrolera que programa entregas de gasolina a sus estaciones de servicio. Para este sistema no hay documento de requerimientos detallado, de manera que OilSoft estima que un precio de $900 000 (dólares estadounidenses) probablemente sea competitivo y esté dentro del presupuesto de la compañía petrolera. Después de que se le otorga el contrato, OilSoft negocia los requerimientos detallados del sistema, de manera que se entrega la funcionalidad básica. Entonces se estiman los costos adicionales para otros requerimientos. La compañía petrolera no necesariamente pierde aquí, pues otorga el contrato a una compañía en la

que puede confiar. Los requerimientos adicionales podrán financiarse a partir de un presupuesto futuro, así que el presupuesto de la compañía petrolera no resulta alterado por un alto costo inicial del software.

23.2 Desarrollo dirigido por un plan

El desarrollo dirigido por un plan o basado en un plan es un enfoque para la ingeniería de software donde el proceso de desarrollo se planea a detalle. Se elabora un plan de proyecto que registra el trabajo que se va a realizar, quién lo efectuará, el calendario de desarrollo y los productos de trabajo. Los administradores utilizan el plan para apoyar la toma de decisiones del proyecto y también como una forma de medir el progreso. El desarrollo dirigido por un plan se sustenta en técnicas de administración de proyectos de ingeniería y pueden considerarse como la manera tradicional de administrar grandes proyectos de desarrollo de software. Lo anterior contrasta con el desarrollo ágil, en el que muchas decisiones que afectan el desarrollo se retrasan y se hacen posteriormente, según se requiera, durante el proceso de desarrollo.

El principal argumento contra el desarrollo basado en un plan es que muchas decisiones tempranas deben revisarse debido a cambios al entorno en los que se desarrollará y usará el software. Retrasar las decisiones es una práctica sensata porque evita tener que volver a trabajar. Los argumentos en favor de un enfoque dirigido por un plan son que la planeación temprana permite que los asuntos de la organización (disponibilidad de personal, otros proyectos, etcétera) se tomen estrictamente en cuenta, y que los problemas potenciales y dependencias se descubran antes de que se inicie el proyecto, y no cuando ya esté en marcha.

Desde esta perspectiva, el mejor enfoque a la planeación del proyecto incluye una mezcla juiciosa de desarrollo basado en un plan y ágil. El equilibrio depende del tipo de proyecto y de las habilidades del personal que estén disponibles. Por un lado, los grandes sistemas críticos de seguridad y protección requieren un amplio análisis previo y quizá certificarse antes de utilizarse. Esto debe estar dirigido mediante un plan. Por el otro lado, los sistemas de información, de pequeños a medianos, que se usan en un entorno competitivo vertiginosamente cambiante, deben ser ágiles. Cuando muchas compañías están implicadas en un proyecto de desarrollo, con frecuencia se usa un enfoque basado en un plan para coordinar el trabajo a través de cada sitio de desarrollo.

23.2.1 Planes de proyecto

En un proyecto de desarrollo dirigido por un plan, en el plan de proyecto se establecen los recursos disponibles para el proyecto, la división del trabajo y un calendario para realizar el trabajo. El plan debe identificar los riesgos para el proyecto y el software en desarrollo, así como el enfoque que se toma para la gestión del riesgo. Aunque los detalles específicos de los planes de proyecto varían dependiendo del tipo de proyecto y organización, los planes incluyen por lo regular las siguientes secciones:

1. *Introducción* Ésta describe brevemente los objetivos del proyecto y establece las restricciones (por ejemplo, presupuesto, tiempo, etcétera) que afectan la administración del proyecto.

Plan	Descripción
Plan de calidad	Describe los procedimientos de calidad y estándares que se usarán en un proyecto.
Plan de validación	Describe el enfoque, los recursos y el calendario utilizados para la validación del sistema.
Configuración del plan de gestión	Describe la configuración de los procedimientos y las estructuras para la gestión.
Plan de mantenimiento	Predice los requerimientos, los costos y el esfuerzo de mantenimiento.
Plan de desarrollo de personal	Describe cómo se desarrollarán las habilidades y la experiencia de los miembros del equipo de proyecto.

Figura 23.2 Complementos de plan de proyecto

2. *Organización del proyecto* Ésta refiere la forma en que está organizado el equipo de desarrollo, las personas implicadas y sus roles en el equipo.
3. *Análisis de riesgo* Detalla los posibles riesgos del proyecto, la probabilidad de que surjan dichos riesgos y las estrategias propuestas para reducir el riesgo. La gestión del riesgo se trata en el capítulo 22.
4. *Requerimientos de recursos de hardware y software* Detallan el hardware y el software de soporte requeridos para realizar el desarrollo. Si hay que comprar hardware, pueden incluirse estimaciones de los precios y el calendario de entregas.
5. *División del trabajo* Establece la división del proyecto en actividades e identifica los plazos y las entregas asociados con cada actividad. Los plazos son las etapas clave del proyecto donde puede valorarse el avance; las entregas son productos de trabajo que se proporcionan al cliente.
6. *Calendario del proyecto* Indica las dependencias entre las actividades, el tiempo estimado requerido para alcanzar cada plazo y la asignación de personal a las actividades. Las formas en las que puede presentarse el calendario se estudian en la siguiente sección del capítulo.
7. *Mecanismos de monitorización y reporte* Esta sección define los informes administrativos que deben producirse, cuándo tienen que elaborarse y los mecanismos de monitorización del proyecto que se usarán.

Además del plan de proyecto principal, que debe enfocarse en los riesgos para los proyectos y el calendario, conviene desarrollar algunos planes complementarios para apoyar otras actividades del proceso, como las pruebas y la administración de la configuración. En la figura 23.2 se muestran ejemplos de posibles planes complementarios.

23.2.2 El proceso de planeación

La planeación del proyecto es un proceso iterativo que comienza cuando se diseña un plan de proyecto inicial durante la fase de arranque del proyecto. La figura 23.3 es

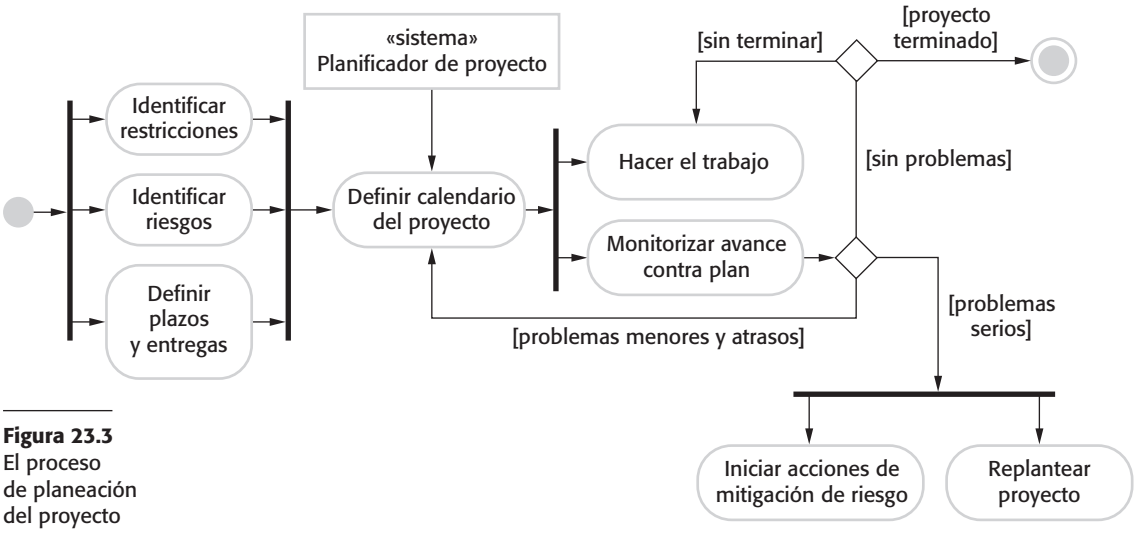


Figura 23.3
El proceso de planeación del proyecto

un diagrama de actividad UML que muestra un flujo de trabajo típico para un proceso de planeación de proyecto. Los cambios al plan son inevitables. Conforme más información sobre el sistema y el equipo esté disponible durante el proyecto, habrá que revisar regularmente el plan para reflejar los requerimientos, el calendario y los cambios en el riesgo. Modificar las metas de la empresa conduce también a cambios en los planes del proyecto. A medida que cambien las metas de la empresa, esto podría afectar a todos los proyectos, los cuales tal vez deban replantearse.

Al comienzo de un proceso de planeación, hay que valorar las restricciones que afectan el proyecto. Éstas son fecha de entrega requerida, personal disponible, presupuesto global, herramientas disponibles, etcétera. En conjunción con esto, también hay que identificar los hitos y entregables del proyecto. Los hitos son puntos en el calendario contra los que puede valorar el avance, por ejemplo, la transferencia del sistema para pruebas. Los entregables son productos de trabajo que se proporcionan al cliente (por ejemplo, un documento de requerimientos para el sistema).

Entonces el proceso entra en un ciclo. Se prepara un calendario estimado para el proyecto y se inician las actividades definidas en el calendario o se concede el permiso para continuarlas. Después de cierto tiempo (por lo general de dos a tres semanas), se debe revisar el avance y anotar las diferencias del calendario planeado. Puesto que las estimaciones iniciales de los parámetros del proyecto inevitablemente son aproximadas, es normal que se presenten atrasos menores y habrá que hacer modificaciones al plan original.

Es importante ser realista al elaborar un plan de proyecto. Los problemas de alguna descripción surgen casi siempre durante un proyecto y pueden conducir a demoras del mismo. En consecuencia, las suposiciones y la calendarización iniciales deben ser más pesimistas que optimistas. Tiene que haber suficiente contingencia acumulada en el plan, de modo que las restricciones y los hitos del plan no necesiten renegociarse cada vez que se revisa el ciclo de planeación.

Si existen graves problemas con el trabajo de desarrollo que conduzcan a demoras significativas, habrá que iniciar acciones de mitigación del riesgo para reducir los riesgos de falla del proyecto. Junto con dichas acciones, se debe también replantear el proyecto.

Esto puede incluir renegociar las restricciones del proyecto y entregables con el cliente. También es necesario establecer y acordar con el cliente un nuevo calendario sobre el tiempo en que se completará el trabajo.

Si esta renegociación no tiene éxito o si no son efectivas las acciones de mitigación del riesgo, se debe organizar entonces una revisión técnica formal del proyecto. Los objetivos de esta revisión son encontrar un enfoque alternativo que permita la continuación del proyecto, y comprobar si éste, así como las metas del cliente y el desarrollador de software, todavía están alineados.

El resultado de una revisión puede ser una decisión para cancelar un proyecto. Esto podría obedecer a un efecto de los fracasos técnicos o administrativos pero, a menudo, es consecuencia de cambios externos que afectan al proyecto. Durante este tiempo, los objetivos y las prioridades de la compañía cambian inevitablemente. Tales cambios pueden significar que el software ya no se requiere más o que los requerimientos del proyecto original resultan inadecuados. Entonces la administración puede decidir detener el desarrollo del software o realizar grandes cambios al proyecto para que éstos reflejen los cambios en los objetivos de la organización.

23.3 Calendarización de proyectos

La calendarización de proyectos es el proceso de decidir cómo se organizará el trabajo en un proyecto como tareas separadas, y cuándo y cómo se ejecutarán dichas tareas. Se estima el tiempo calendario para completar cada tarea, el esfuerzo requerido y quién trabajará en las tareas identificadas. También hay que estimar los recursos necesarios para completar cada tarea (como el espacio de disco requerido en un servidor), el tiempo que se necesitará el hardware especializado (como un simulador) y cuál será el presupuesto de viajes. En términos de las etapas de planeación estudiadas en la introducción de este capítulo, un calendario de proyecto inicial se elabora por lo general durante la fase de arranque del proyecto. Luego, durante la planeación del desarrollo, este calendario se afina y modifica.

Tanto los procesos basados en un plan como los ágiles precisan de un calendario de proyecto inicial, aunque el nivel de detalle puede ser menor en un plan de proyecto ágil. Este calendario inicial se utiliza para planear cómo se asignarán las personas al proyecto y comprobar el avance de éste frente a los compromisos contractuales. En los procesos tradicionales de desarrollo, el calendario completo se elabora inicialmente y enseguida se modifica conforme avanza el proyecto. En los procesos ágiles debe existir un calendario global que identifique el tiempo en que se completarán las principales fases del proyecto. Entonces, se usa un enfoque iterativo de calendarización para planear cada fase.

La calendarización en los proyectos dirigidos por un plan (figura 23.4) implica dividir el trabajo total de un proyecto en tareas separadas y estimar el tiempo requerido para completar cada tarea. Por lo general, las tareas deben durar al menos una semana, pero no más de dos meses. Una subdivisión más fina significa que una cantidad desproporcionada de tiempo debe emplearse para volver a planear y actualizar el plan del proyecto. La cantidad máxima de tiempo para cualquier tarea debe durar alrededor de ocho a 10 semanas. Si tarda más que esto, la tarea debe subdividirse para la planeación y calendarización del proyecto.

Gráficas de actividad

Una gráfica de actividad es una representación del calendario del proyecto que muestra cuáles tareas pueden realizarse en forma paralela y las que deben ejecutarse en secuencia, debido a su dependencia respecto a actividades anteriores. Si una tarea depende de muchas otras, entonces todas éstas deben terminarse antes de comenzar aquella. La ruta crítica a través de la gráfica de actividad es la secuencia más larga de tareas dependientes. Esto define la duración del proyecto.

http://www.SoftwareEngineering-9.com/Web/Planning/activities.html

Algunas de estas tareas se realizan paralelamente, con distintas personas que trabajan en diferentes componentes del sistema. Es necesario coordinar las tareas paralelas y organizar las actividades para que la fuerza de trabajo se desempeñe de manera óptima y no introduzca entre las tareas dependencias innecesarias. Es importante evitar una situación en la que todo el proyecto se demore debido a que una tarea crítica no está terminada.

Si un proyecto está técnicamente avanzado, las estimaciones iniciales seguramente serán optimistas aun cuando se trate de considerar todas las eventualidades. En este aspecto, la calendarización del software no es diferente de la de algún otro tipo de gran proyecto avanzado. La construcción de las nuevas aeronaves, los puentes y los nuevos modelos de automóviles con frecuencia sufre retrasos, debido a problemas no previstos. Por lo tanto, los calendarios deben actualizarse continuamente conforme se disponga de mejor información sobre el avance. Si el proyecto a calendarizar es similar a un proyecto anterior, pueden reutilizarse las estimaciones previas. Sin embargo, los proyectos pueden usar diferentes métodos de diseño y lenguajes de implementación, de modo que la experiencia de proyectos anteriores tal vez no sea aplicable en la planeación de un proyecto nuevo.

Como ya se sugirió, al evaluar calendarios hay que tomar en cuenta la posibilidad de que las cosas salgan mal. Las personas que trabajan en un proyecto podrían enfermar o cambiar de trabajo, el hardware puede fallar y la entrega del software o hardware de apoyo esencial quizá se demore. Si el proyecto es nuevo y técnicamente avanzado, partes de éste pueden resultar más difíciles y tardar más que lo previsto originalmente.

Una buena regla empírica es estimar que nada saldrá mal, luego ampliar la estimación para enfrentar problemas anticipados. También puede añadirse a la estimación un factor de contingencia para hacer frente a problemas no anticipados. Este factor de contingencia adicional depende del tipo de proyecto, los parámetros del proceso (plazo, estándares, etcétera) y la calidad y experiencia de los ingenieros de software que trabajan en el proyecto. Las estimaciones de contingencia pueden agregar entre 30 y 50% al esfuerzo y tiempo requeridos para el proyecto.

23.3.1 Representación del calendario

Los calendarios de proyecto pueden representarse simplemente en una tabla u hoja de cálculo que indique las tareas, el esfuerzo, la duración esperada y las dependencias entre las tareas (figura 23.5). Sin embargo, este modo de representación dificulta visualizar las relaciones y dependencias entre las diferentes actividades. Por esta razón se han

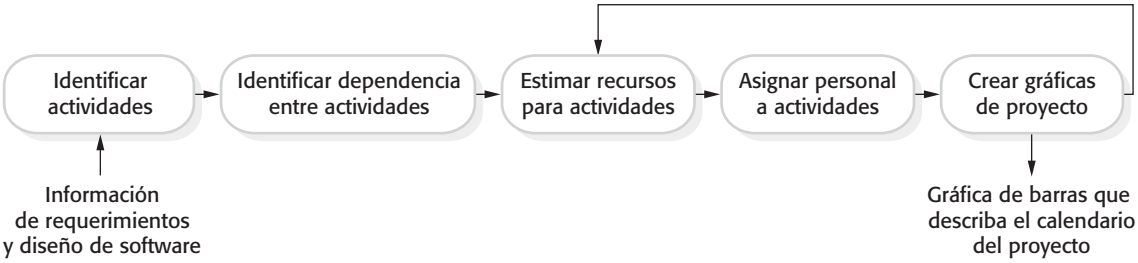


Figura 23.4
El proceso de calendarización de proyecto

desarrollado representaciones gráficas alternativas de los calendarios de proyecto, que con frecuencia son más fáciles de leer y entender. Existen dos tipos de representación que se usan comúnmente:

1. Gráficas de barras, basadas en el calendario, las cuales señalan al responsable de cada actividad, el tiempo transcurrido previsto y la fecha en que se programó el inicio y el fin de la actividad. En ocasiones, las gráficas de barras se llaman gráficas de Gantt, en honor de su inventor, Henry Gantt.
2. Redes de actividad, son diagramas de red que muestran las dependencias entre las diferentes actividades que constituyen un proyecto.

Por lo general, una herramienta de planeación se usa para gestionar la información del calendario del proyecto. Dichas herramientas esperan a menudo que se introduzca la información en una tabla y luego crearán una base de datos de información del proyecto. Entonces, a partir de esta base de datos, se generan automáticamente gráficas de barras y de actividad.

Las actividades de proyecto son el elemento de planeación básico. Cada actividad cuenta con:

1. Una duración en días o meses calendario.
2. Una estimación del esfuerzo, la cual refleja el número de días-hombre o meses-hombre para completar el trabajo.
3. Un plazo dentro del cual debe completarse la actividad.
4. Un punto final definido. Éste representa el resultado tangible de completar la actividad. También podría ser un documento, la realización de una junta de revisión, una ejecución exitosa de todas las pruebas, etcétera.

Cuando planee un proyecto, también deberá definir los hitos; esto es, cada etapa del proyecto en la que puede realizarse una valoración del avance. Cada hito debe documentarse mediante un breve reporte que compendie el avance realizado y el trabajo efectuado. Los hitos pueden asociarse con una sola tarea o con grupos de actividades relacionadas. Por ejemplo, en la figura 23.5, el hito M1 se asocia con la tarea T1, mientras que el hito M3 se asocia con un par de tareas, T2 y T4.

Un tipo especial de hito es la producción de un entregable del proyecto. Un entregable es un producto de trabajo que se entrega al cliente. Es el resultado de una fase significativa del proyecto, como la especificación o el diseño. Por lo general, los entregables

Tarea	Esfuerzo (días-hombre)	Duración (días)	Dependencias
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figura 23.5 Tareas, duraciones y dependencias

requeridos se especifican en el contrato del proyecto, y la visión del cliente del avance del proyecto depende de dichos entregables.

Para ilustrar cómo se usan las gráficas de barras se creó un conjunto hipotético de tareas, que se presenta en la figura 23.5. Esta tabla indica tareas, esfuerzo estimado, duración e interdependencias de tareas. En la figura 23.5 se observa que la tarea T3 depende de la tarea T1. Por lo tanto, la tarea T1 debe completarse antes de comenzar la tarea T3. Por ejemplo, T1 puede ser la preparación del diseño de un componente, y T3 la implementación de ese diseño. Antes de comenzar la implementación, debe completarse el diseño. Observe que la duración estimada para algunas tareas es mayor que el esfuerzo requerido y viceversa. Si el esfuerzo es menor que la duración, significa que las personas asignadas a dicha tarea no trabajan en ella de tiempo completo. Si el esfuerzo supera la duración, quiere decir que muchos miembros del equipo trabajan al mismo tiempo en la tarea.

La figura 23.6 toma la información de la figura 23.5 y presenta el calendario del proyecto en formato gráfico. Es una gráfica de barras que muestra un calendario de proyecto y las fechas de inicio y terminación de las actividades. Al leerse de izquierda a derecha, la gráfica de barras señala claramente cuándo comienzan y terminan las tareas. Los hitos (M1, M2, etcétera) se muestran también en la gráfica de barras. Observe que las tareas que son independientes se realizan paralelamente (por ejemplo, las tareas T1, T2 y T4 se inician desde el principio del proyecto).

Además de planear el calendario de entregas para el software, los administradores de proyecto deben asignar recursos a las tareas. Desde luego, el recurso clave son los ingenieros de software que harán el trabajo, y ellos deben asignarse a las actividades del proyecto. También, la asignación de recursos puede ingresarse a las herramientas de administración del proyecto y a una gráfica de barras generada, que muestra cuándo el personal trabaja en el proyecto (figura 23.7). Las personas pueden trabajar en más de una

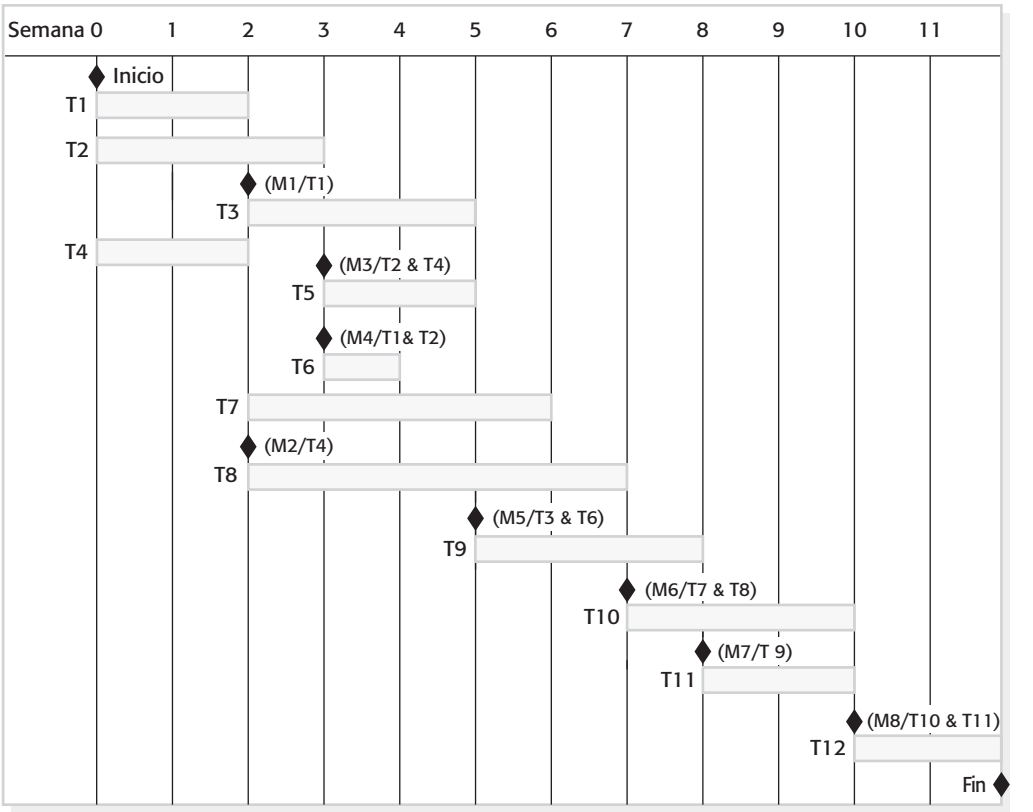


Figura 23.6 Gráfica de barras de actividad

tarea al mismo tiempo y, en ocasiones, no laborar en el proyecto. Pueden estar de vacaciones, ocupados en otros proyectos, en cursos de capacitación o inmersos en alguna otra actividad. Las asignaciones de tiempo parcial se indican con una línea diagonal que atraviesa la barra.

Las grandes organizaciones, por lo general, emplean a varios especialistas que trabajan en un proyecto cuando se requiere. En la figura 23.7 se observa que Mary es una especialista que trabaja sólo en una tarea del proyecto. Esto puede generar problemas de calendarización. Si un proyecto se demora mientras un especialista trabaja en él, esto podría tener un efecto dominó sobre otros proyectos donde también se requiera al especialista. Entonces este proyecto puede demorarse porque el especialista no está disponible.

Si una tarea se retrasa, sin duda, afectará el desarrollo de tareas posteriores que dependen de ella. No es posible iniciar éstas antes de completar la tarea retrasada. Las demoras pueden causar graves problemas con la asignación de personal, en especial cuando los individuos trabajan al mismo tiempo en varios proyectos. Si una tarea (T) se demora, las personas seleccionadas pueden asignarse a otro trabajo (W). Completar este último tal vez tarde más que la demora; sin embargo, el personal, una vez asignado, no puede simplemente reasignarse a la tarea original (T). Entonces esto conducirá a más aplazamientos en T mientras se completa W.

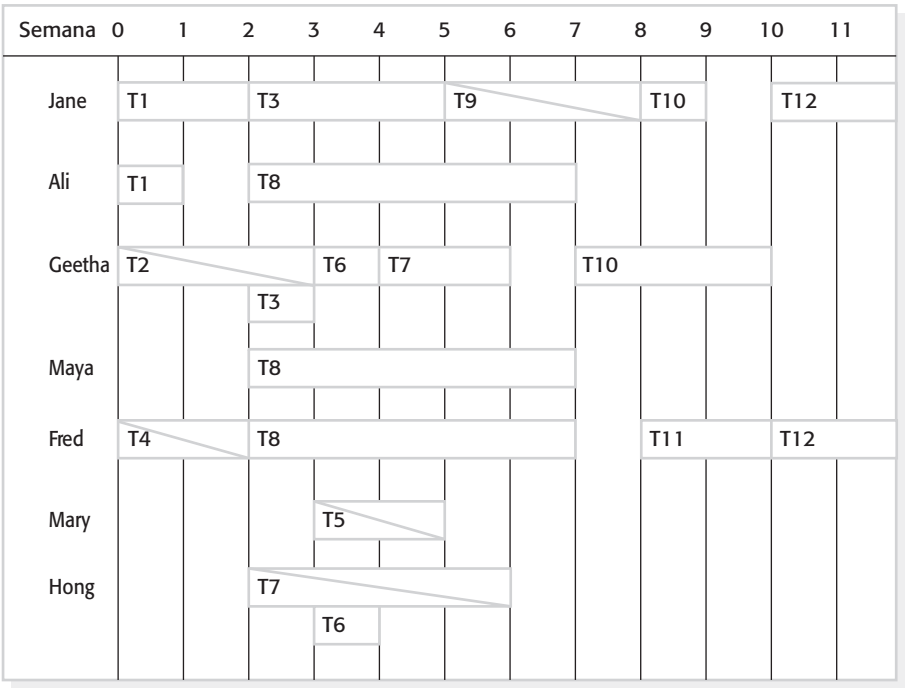


Figura 23.7 Gráfica de asignación de personal

23.4 Planeación ágil

Los métodos ágiles de desarrollo de software son enfoques iterativos donde el software se desarrolla y entrega a los clientes en incrementos. A diferencia de los enfoques dirigidos por un plan, la funcionalidad de dichos incrementos no se planea por anticipado, sino que se decide durante el desarrollo. La decisión acerca de qué incluir en un incremento depende del progreso y las prioridades del cliente. El argumento para este enfoque es que las prioridades y requerimientos del cliente cambian, de manera que tiene sentido tener un plan flexible que pueda acomodar dichos cambios. El libro de Cohn (Cohn, 2005) es un análisis exhaustivo de los conflictos de planeación en los proyectos ágiles.

Los enfoques ágiles de uso más común, como Scrum (Schwaber, 2004) y la programación extrema (Beck, 2000), tienen un enfoque de dos etapas para la planeación, las cuales corresponden a la fase de arranque en el desarrollo dirigido por un plan y la planeación del desarrollo:

1. Planeación de la entrega (*release*), que prevé con muchos meses de antelación y decide sobre las características que deben incluirse en una entrega de un sistema.
2. Planeación de la iteración, la cual tiene un panorama a corto plazo y se enfoca en la planeación del siguiente incremento de un sistema. Esto, para el equipo, generalmente representa de dos a cuatro semanas de trabajo.

En el capítulo 3 se estudió el enfoque Scrum a la planeación, de modo que aquí el enfoque estará en la planeación de la programación extrema (XP). A esto se le llama

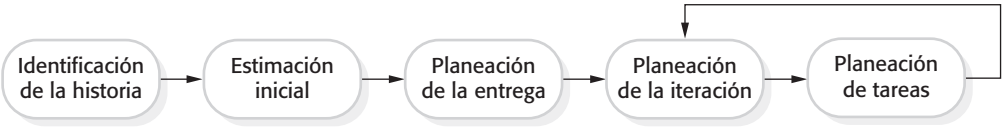


Figura 23.8
Planeación en XP

"juego de planeación" y por lo general implica a todo el equipo de desarrollo, incluidos los representantes del cliente. La figura 23.8 muestra las etapas del juego de planeación.

La especificación del sistema en XP se basa en historias del usuario, las cuales reflejan las características que deben incluirse en el sistema. Al inicio del proyecto, el equipo y el cliente tratan de identificar un conjunto de historias que comprendan toda la funcionalidad que se incluirá en el sistema final. Inevitablemente se perderá cierta funcionalidad, pero en esta etapa esto no es importante.

La siguiente fase es una etapa de estimación. El equipo del proyecto lee y discute las historias y las clasifica de acuerdo con la cantidad de tiempo que consideran que se tardará implementar la historia. Esto puede implicar la división de las historias grandes en más pequeñas. La estimación relativa con frecuencia es más sencilla que la estimación absoluta. Por lo regular, las personas encuentran difícil evaluar cuánto esfuerzo o tiempo se requiere para hacer algo. Sin embargo, cuando se les presentan muchas actividades por hacer, pueden emitir juicios acerca de cuáles historias requerirán más tiempo y más esfuerzo. Una vez completada la clasificación, entonces el equipo asigna puntos de esfuerzo hipotéticos a las historias. Una historia compleja puede tener 8 puntos y una historia sencilla 2. Esto se hace para todas las historias en la lista clasificada.

Una vez estimadas las historias, el esfuerzo relativo se traduce en la primera estimación del esfuerzo total requerido usando la noción de velocidad. En XP, velocidad es el número de puntos de esfuerzo implementados por el equipo, por día. Esto puede valuarse a partir de la experiencia previa o al desarrollar una o dos historias para ver el tiempo que se requiere. La estimación de la velocidad es aproximada, pero se afina durante el proceso de desarrollo. Una vez evaluada la velocidad, es posible calcular el esfuerzo total en días-hombre para implementar el sistema.

La planeación de entrega implica seleccionar y afinar las historias que reflejarán las características a aplicar en una entrega de un sistema y el orden en el que deben implementarse las historias. El cliente tiene que participar en este proceso. Entonces se elige una fecha de entrega y las historias se examinan para ver si la estimación del esfuerzo es congruente con dicha fecha. Si no lo es, las historias se agregan o eliminan de la lista.

La planeación de iteración es la primera etapa en el proceso de desarrollo de iteración. Se eligen las historias a implementar para dicha iteración; el número de historias refleja el tiempo para entregar una iteración (por lo general dos o tres semanas) y la velocidad del equipo. Cuando se alcanza la fecha de entrega de la iteración, ésta se completa, incluso si no se han implementado todas las historias. El equipo considera las historias que se implementaron y suma sus puntos de esfuerzo. Entonces puede calcularse nuevamente la velocidad y ésta se considera en la planeación de la siguiente entrega del sistema.

Al inicio de cada iteración hay una etapa de planeación más detallada en que los desarrolladores dividen las historias en tareas de desarrollo. Una tarea de desarrollo debe tardar de cuatro a 16 horas. Se mencionan todas las tareas que deben completarse para implementar todas las historias en dicha iteración. Entonces los desarrolladores indivi-

duales se comprometen a cumplir las tareas específicas que implementarán. Cada desarrollador conoce su rapidez individual, de manera que no se comprometerá para más tareas de las que puede implementar en el tiempo.

Existen dos beneficios importantes a partir de este enfoque a la asignación de tareas:

1. Todo el equipo consigue un panorama de las tareas a completar en una iteración. Por lo tanto, todos tienen una comprensión de lo que hacen otros miembros del equipo y saben a quién dirigirse si se identifican dependencias de tarea.
2. Los desarrolladores individuales eligen las tareas a implementar; no son simplemente tareas asignadas por un administrador de proyecto. En consecuencia, tienen un sentido de propiedad sobre dichas actividades y es probable que esto los motive a completar la tarea.

A la mitad de una iteración se revisa el avance. En esta etapa deben estar completos la mitad de los puntos de esfuerzo de la historia. De este modo, si una iteración implica 24 puntos de historia y 36 tareas, 12 puntos de historia y 18 tareas deben estar completos. Si éste no es el caso, se debe consultar al cliente y eliminar algunas historias de la iteración.

Este enfoque de la planeación tiene la ventaja de que el software siempre se entrega como se planeó y no hay atraso en el calendario. Si el trabajo no puede completarse en el tiempo asignado, la filosofía XP es reducir el alcance del trabajo en lugar de extender el calendario. Sin embargo, en algunos casos, el incremento puede no ser suficiente para ser útil. Reducir el ámbito podría generar trabajo adicional para los clientes si deben usar un sistema incompleto o cambiar sus prácticas laborales entre una entrega del sistema y otra.

Una gran dificultad en la planeación ágil es que depende del involucramiento y la disponibilidad del cliente. En la práctica, esto suele ser difícil de organizar, pues los representantes del cliente en ocasiones deben dar prioridad a otros trabajos. Los clientes tal vez estén más familiarizados con planes de proyecto tradicionales y encuentren difícil participar en un proyecto de planeación ágil.

La planeación ágil funciona bien con equipos de desarrollo pequeños y estables, que pueden reunirse y discutir las historias a implementar. No obstante, cuando los equipos son grandes y/o están geográficamente distribuidos, o cuando cambia con frecuencia la conformación del equipo, es casi imposible que todos colaboren en la planeación, lo cual es esencial para la administración de proyecto ágil. Por consiguiente, los proyectos grandes se planean generalmente usando enfoques tradicionales a la administración de proyectos.

23.5 Técnicas de estimación

Es difícil la estimación del calendario del proyecto. Probablemente haya que hacer estimaciones iniciales sobre la base de una definición de requerimientos de usuario de alto nivel. El software puede ejecutarse en computadoras no familiares o usar nueva tecnología de desarrollo. Quizá no lleguen a conocerse las personas involucradas en el proyecto

y sus habilidades. Existe tanta incertidumbre que es imposible estimar con precisión los costos de desarrollo del sistema durante las primeras etapas de un proyecto.

Incluso existe una dificultad fundamental en la valoración de la precisión de diferentes enfoques a la estimación del costo y esfuerzo. Con frecuencia, las estimaciones del proyecto se autosatisfacen. La estimación se utiliza para definir el presupuesto del proyecto, y el producto se ajusta para que se cumpla la cifra del presupuesto. Un proyecto que está dentro de presupuesto puede lograr esto a expensas de las características en el software a desarrollar.

No se conocen experimentos controlados con el costo de proyectos donde los costos estimados no se usen para sesgar el experimento. Un experimento controlado no revelaría la estimación de costo al administrador del proyecto. Los costos reales se compararían entonces con los costos de proyecto estimados. No obstante, las organizaciones necesitan hacer evaluaciones de esfuerzo y costo del software. Existen dos tipos de técnicas para ello:

1. *Técnicas basadas en la experiencia* La estimación de los requerimientos de esfuerzo futuro se basan en la experiencia del administrador con proyectos anteriores y el dominio de aplicación. En esencia, el administrador emite un juicio informado de cuáles serán los requerimientos de esfuerzo.
2. *Modelado algorítmico de costo* En este caso se usa un enfoque formulista para calcular el esfuerzo del proyecto con base en estimaciones de atributos del producto (por ejemplo, el tamaño), así como las características del proceso (por ejemplo, la experiencia del personal implicado).

En ambos casos es necesario usar el juicio para evaluar el esfuerzo directamente, estimar el proyecto y las características del producto. En la fase de arranque de un proyecto, dichas estimaciones tienen un amplio margen de error. Con base en datos recopilados de un gran número de proyectos, Boehm y sus colaboradores (1995) descubrieron que las estimaciones de arranque varían significativamente. Si la estimación inicial del esfuerzo requerido es de x meses de esfuerzo, el rango puede ser de $0.25x$ a $4x$ del esfuerzo real, medido cuando el sistema se entregó. Durante la planeación del desarrollo, las estimaciones se vuelven cada vez más precisas conforme avanza el proyecto (figura 23.9).

Las técnicas basadas en la experiencia dependen de la experiencia del administrador de proyectos anteriores y el esfuerzo real empleado en dichos proyectos en actividades que se relacionan con el desarrollo del software. Por lo general, se identifican los entregables que hay que producir en un proyecto y los diferentes componentes de software o sistemas a desarrollar. Esto se documenta en una hoja de cálculo, se les estima de manera individual y se calcula el esfuerzo total requerido. Por lo general, ayuda a que un grupo de personas se involucre en la estimación del esfuerzo y a pedir a cada miembro del grupo que explique sus estimaciones. Con frecuencia, esto revela factores que otros no consideraron y entonces se itera hacia una estimación grupal consensuada.

La dificultad con las técnicas basadas en la experiencia es que un nuevo proyecto de software puede no tener mucho en común con proyectos anteriores. El desarrollo de software cambia muy rápidamente y con frecuencia un proyecto usará técnicas no familiares, tales como servicios Web, desarrollo basado en COTS o AJAX. Si usted no ha trabajado con estas técnicas, su experiencia previa puede no ser de ayuda para estimar el esfuerzo requerido, lo que dificultará producir estimaciones precisas de costo y calendario.

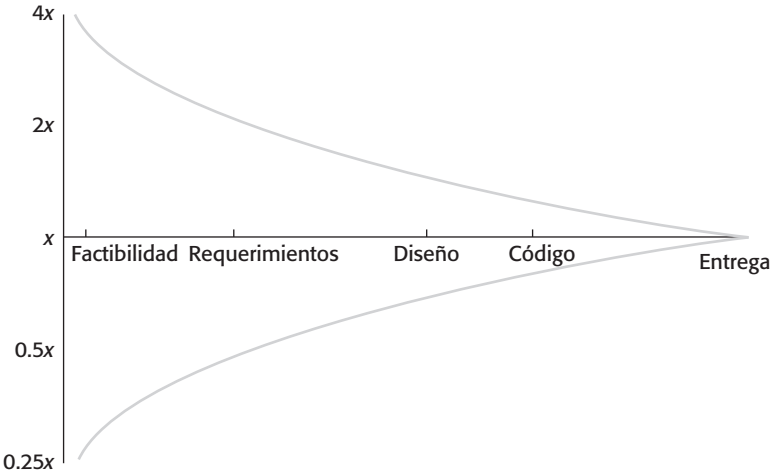


Figura 23.9
Incertidumbre de estimación

23.5.1 Modelado algorítmico de costos

El modelado algorítmico de costos utiliza una fórmula matemática para predecir los costos del proyecto con base en estimaciones del tamaño del proyecto, el tipo de software a desarrollar, y otros factores de equipo, proceso y producto. Un modelo algorítmico de costo puede elaborarse al analizar los costos y atributos de los proyectos completados, y encontrar la fórmula de ajuste más cercana a la experiencia real.

Los modelos algorítmicos de costo se usan principalmente para hacer estimaciones de los costos de desarrollo de software. Sin embargo, Boehm y sus colaboradores (2000) examinan una variedad de otros usos para dichos modelos, como la elaboración de estimaciones para inversionistas en compañías de software, estrategias alternativas para ayudar a valorar los riesgos, y decisiones informadas acerca de reutilización, replaneación o subcontratación.

Los modelos algorítmicos para estimar el esfuerzo en un proyecto de software se basan principalmente en una fórmula sencilla:

$$\text{Esfuerzo} = A \times \text{Tamaño}^B \times M$$

A es un factor constante que depende de las prácticas locales de la organización y el tipo de software que se desarrolla. El tamaño puede ser una valoración del tamaño del código del software o una estimación de la funcionalidad expresada en puntos de función o de aplicación. El valor del exponente B se encuentra por lo general entre 1 y 1.5. M es un multiplicador que se integra al combinar atributos de procesos, producto y desarrollo, tales como los requerimientos de confiabilidad para el software y la experiencia del equipo de desarrollo.

El número de líneas de código fuente (SLOC) en el sistema entregado es la métrica de tamaño fundamental que se utiliza en muchos modelos algorítmicos de costo. La estimación del tamaño puede implicar estimación por analogía con otros proyectos, estimación al convertir los puntos de función o aplicación al tamaño del código, estimación al clasificar los tamaños de los componentes del sistema y uso de un componente de referencia conocido para estimar el tamaño del componente, o simplemente puede ser una cuestión de juicio de ingeniería.

La mayoría de los modelos de estimación algorítmica tienen un componente exponencial (B en la ecuación anterior) que se relaciona con el tamaño y la complejidad del sistema. Esto refleja el hecho de que los costos no aumentan con regularidad linealmente con el tamaño del proyecto. Conforme se incrementa el tamaño y la complejidad del software, se incurre en costos adicionales debido a los costos generales de comunicación de los equipos más grandes, la administración de configuración más compleja, la integración de sistemas más difícil, etcétera. Cuanto más complejo sea el sistema, más afectarán al costo estos factores. Por lo tanto, el valor de B aumenta normalmente con el tamaño y la complejidad del sistema.

Todos los modelos algorítmicos tienen problemas similares:

1. Con frecuencia es difícil estimar el Tamaño en una etapa temprana del proyecto, cuando sólo está disponible la especificación. Las estimaciones de punto de función y punto de aplicación (véase más adelante) son más fáciles de producir que las estimaciones de tamaño del código, pero por lo general aún son imprecisas.
2. Las estimaciones de los factores que contribuyen a B y M son subjetivas. Las estimaciones varían de una persona a otra, dependiendo de sus antecedentes y experiencia con el tipo de sistema que se desarrolla.

La estimación precisa del tamaño del código es difícil en una etapa temprana de un proyecto, porque el tamaño del programa final depende de decisiones de diseño que pueden no haberse hecho cuando se requirió la estimación. Por ejemplo, una aplicación que requiere gestión de datos de alto rendimiento puede implementar su propio sistema de gestión de datos o usar un sistema de base de datos comercial. En la estimación inicial del costo, no es probable que se conozca si existe un sistema de base de datos comercial que se desempeñe bastante bien para cumplir los requerimientos de rendimiento. Por lo tanto, se desconoce cuánto código de gestión de datos se incluirá en el sistema.

El lenguaje de programación usado para desarrollar el sistema afecta también el número de líneas de código a desarrollar. Un lenguaje como Java puede significar que se necesiten más líneas de código que si se usa C (por ejemplo). Sin embargo, este código adicional permite más comprobación de tiempo de compilación, de manera que es probable que se reduzcan los costos de validación. ¿Cómo debe tomarse esto en cuenta? Más aún, se puede reutilizar una cantidad significativa de código de proyectos anteriores, así que tendrá que ajustarse la estimación del tamaño para tomar esto en cuenta.

Los modelos algorítmicos de costo son una forma sistemática de estimar el esfuerzo requerido para desarrollar un sistema, aunque dichos modelos son complejos y difíciles de usar. Existen muchos atributos y un margen considerable para la incertidumbre al estimar sus valores. Esta complejidad desalienta a los usuarios potenciales y, por lo tanto, la aplicación práctica del modelado algorítmico de costos se limita a un número pequeño de compañías.

Otra barrera que desalienta el uso de los modelos algorítmicos es la necesidad de la calibración. Los usuarios del modelo deben calibrar sus modelos y valores de los atributos con sus datos históricos de proyecto, pues ello refleja la práctica y la experiencia locales. Sin embargo, muy pocas organizaciones recaban suficientes datos de proyectos anteriores en una forma que soporte la calibración del modelo. En consecuencia, el uso práctico de modelos algorítmicos debe comenzar con los valores publicados para los parámetros del modelo. Es casi imposible que un modelador sepa qué tan estrechamente se relacionan éstos con su organización.

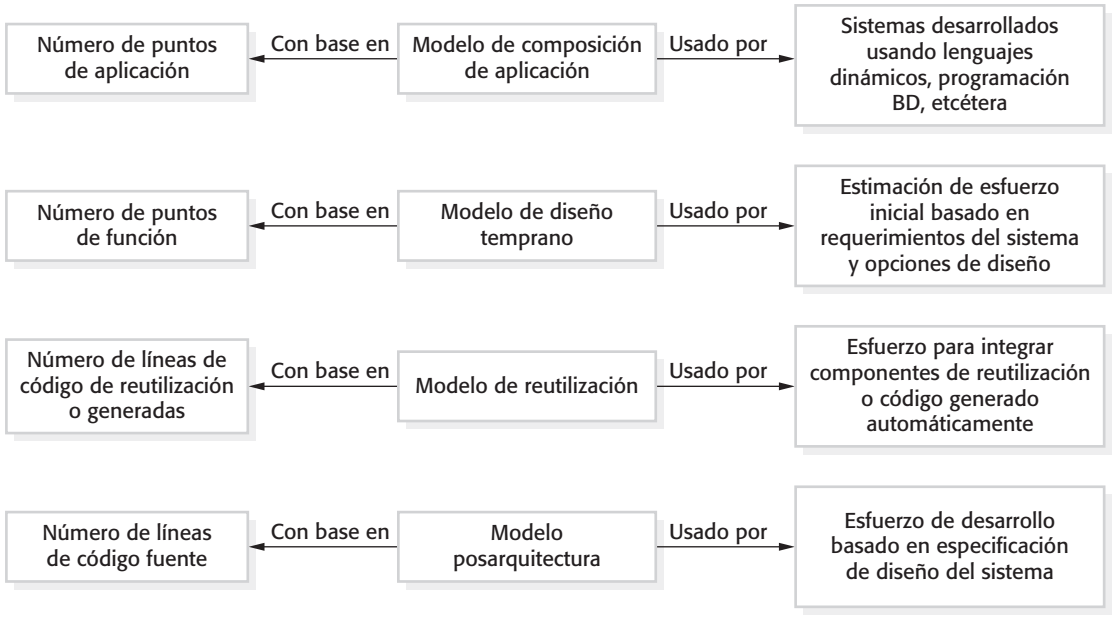


Figura 23.10 Modelos de estimación COCOMO

Si se usa un modelo algorítmico de estimación de costos, hay que desarrollar un rango de estimaciones (peor, esperado y mejor) en lugar de una sola estimación y aplicar la fórmula de costo a todas ellas. Es más probable que las estimaciones sean precisas cuando se comprende el tipo de software que se desarrolla, se calibró el modelo de costeo usando datos locales, o cuando se predefinen el lenguaje de programación y las opciones de hardware.

23.5.2 El modelo COCOMO II

Se han propuesto algunos modelos similares para coadyuvar a estimar el esfuerzo, el calendario y los costos de un proyecto de software. El modelo que se estudia aquí es el COCOMO II. Éste es un modelo empírico que se derivó al recopilar datos a partir de un gran número de proyectos de software. Dichos datos se analizaron para descubrir qué fórmulas se ajustaban mejor con las observaciones. Dichas fórmulas vinculan el tamaño del sistema y los factores del producto, proyecto y equipo, con el esfuerzo para desarrollar el sistema. COCOMO II es un modelo de estimación bien documentado y no registrado.

COCOMO II se desarrolló a partir de los primeros modelos de estimación de costos COCOMO, que se basaron principalmente en el desarrollo de código original (Boehm, 1981; Boehm y Royce, 1989). El modelo COCOMO II toma en cuenta enfoques más modernos para el desarrollo de software, tales como el desarrollo rápido que usa lenguajes dinámicos, el desarrollo mediante composición de componentes y el uso de programación de base de datos. COCOMO II soporta el modelo en espiral de desarrollo, descrito en el capítulo 2, e incrusta submodelos que producen estimaciones cada vez más detalladas.

Los submodelos (figura 23.10) que se consideran parte del modelo COCOMO II son:

1. *Un modelo de composición de aplicación* Éste modela el esfuerzo requerido para desarrollar sistemas que se crean a partir de componentes de reutilización, escritura de

Productividad del software

La productividad del software es una estimación de la cantidad promedio de trabajo de desarrollo que los ingenieros de software completan en una semana o un mes. Por lo tanto, se expresa como líneas de código/mes, puntos de función/mes, etcétera.

Sin embargo, en tanto que la productividad puede medirse fácilmente donde existe un resultado tangible (por ejemplo, un oficinista que procesa *N* facturas por día), la productividad del software es más difícil de definir. Diferentes personas pueden implementar la misma funcionalidad de distintas formas, mediante un número diferente de líneas de código. La calidad del código también es importante pero, en cierta medida, se considera subjetiva. Por consiguiente, las comparaciones de productividad entre ingenieros de software son poco fiables y, por consiguiente, no son muy útiles para la planeación del proyecto.

http://www.SoftwareEngineering-9.com/Web/Planning/productivity.html

guiones o programación de base de datos. Las estimaciones del tamaño de software se basan en puntos de aplicación, y para estimar el esfuerzo requerido se usa una simple fórmula tamaño/productividad. El número de puntos de aplicación en un programa es una estimación ponderada del número de pantallas separadas que se despliegan, el número de informes que se producen, el número de módulos en lenguajes de programación imperativa (como Java) y el número de líneas de lenguaje de escritura de guiones (*scripting*) o código de programación de base de datos.

2. *Un modelo de diseño temprano* Este modelo se usa durante etapas tempranas del diseño del sistema después de establecer los requerimientos. La estimación se basa en la fórmula de estimación estándar que se discutió en la introducción, con un conjunto simplificado de siete multiplicadores. Las estimaciones se basan en puntos de función, que luego se convierten a número de líneas de código fuente. Los puntos de función son una forma independiente de lenguaje para cuantificar la funcionalidad del programa. El número total de puntos de función en un programa se calcula al medir o estimar el número de entradas y salidas externas, las interacciones de usuario, las interfaces externas y las tablas de archivos o bases de datos que usa el sistema.

3. *Un modelo de reutilización* Este modelo se emplea para calcular el esfuerzo requerido al integrar los componentes de reutilización y/o código de programa generado automáticamente. Muchas veces se utiliza en conjunto con el modelo posarquitectónico.

4. *Un modelo posarquitectónico* Una vez diseñada la arquitectura del sistema, puede hacerse una estimación más precisa del tamaño del software. Nuevamente, este modelo usa la fórmula estándar para estimación de costo discutida líneas arriba. Sin embargo, incluye un conjunto más extenso de 17 multiplicadores que reflejan características de capacidad personal, del producto y del proyecto.

Desde luego, en sistemas grandes pueden desarrollarse diferentes partes del sistema mediante distintas tecnologías y es posible que no se tenga que estimar todas las partes del sistema con el mismo nivel de precisión. En tales casos se puede usar el submodelo

Experiencia y habilidad del desarrollador	Muy bajo	Bajo	Nominal	Alto	Muy alto
Madurez y capacidad ICASE	Muy bajo	Bajo	Nominal	Alto	Muy alto
PROD (NAP/mes)	4	7	13	25	50

Figura 23.11 Productividad de punto de aplicación

adecuado a cada parte del sistema y combinar los resultados para crear una estimación compuesta.

El modelo de composición de aplicación

El modelo de composición de aplicación se introdujo en COCOMO II para apoyar la estimación del esfuerzo requerido para proyectos de creación de prototipos y proyectos en que el software se desarrolla mediante la composición de los componentes existentes. Se basa en una estimación de puntos de aplicación ponderados (en ocasiones llamados puntos de objeto), divididos entre una estimación estándar de productividad de puntos de aplicación. Entonces la estimación se ajusta de acuerdo con la dificultad al desarrollar cada punto de aplicación (Boehm *et al*., 2000). La productividad depende de la experiencia y habilidad del desarrollador, así como de las capacidades de las herramientas de software (ICASE) usadas para apoyar el desarrollo. La figura 23.11 muestra los niveles de productividad de punto de aplicación sugeridos por los desarrolladores de COCOMO (Boehm *et al*., 1995).

Por lo general, la composición de aplicaciones incluye una significativa reutilización de software. Es casi seguro que ciertos puntos de aplicación en el sistema se implementen a través de componentes de reutilización. En consecuencia, habrá que ajustar la estimación para tomar en cuenta el porcentaje de reutilización previsto. Por lo tanto, la fórmula final para calcular el esfuerzo del prototipo de sistema es:

$$\text{PM} = (\text{NAP} \times (1 - \%\text{reutilización}/100)) / \text{PROD}$$

PM es la estimación del esfuerzo en meses-hombre. NAP es el número de puntos de aplicación en el sistema entregado. "%reutilización" es una estimación de la cantidad de código de reutilización en el desarrollo. PROD es la productividad del punto de aplicación, tal como se muestra en la figura 23.11. El modelo produce una estimación aproximada, pues no toma en cuenta el esfuerzo adicional incluido en la reutilización.

El modelo de diseño temprano

Este modelo puede usarse durante las primeras etapas de un proyecto, antes de que esté disponible un diseño arquitectónico detallado para el sistema. Las estimaciones de diseño temprano son más útiles para la exploración de opciones en que es necesario comparar diferentes formas de implementar los requerimientos del usuario. El modelo de diseño temprano supone que se acordaron los requerimientos del usuario y que están en marcha las etapas

iniciales del proceso de diseño del sistema. La meta en esta etapa debe ser elaborar una estimación rápida y aproximada de los costos. Por lo tanto, habrá que hacer suposiciones simplificadoras, por ejemplo, que el esfuerzo implicado en la integración del código de reutilización es cero.

Las estimaciones generadas en esta etapa se basan en la fórmula estándar para modelos algorítmicos, esto es:

$$\text{Esfuerzo} = A \times \text{Tamaño}^{B} \times M$$

Con base en su propio gran conjunto de datos, Boehm propuso que el coeficiente **A** debe ser 2.94. El tamaño del sistema se expresa en KSLOC, que es el número de miles de líneas de código fuente. Las KSLOC se calculan al estimar el número de puntos de función en el software. Entonces se usan tablas estándar que relacionan el tamaño del software con puntos de función para diferentes lenguajes de programación, con la finalidad de hacer una estimación inicial del tamaño del sistema en KSLOC.

El exponente **B** refleja el esfuerzo creciente requerido conforme aumenta el tamaño del proyecto. Esto puede variar de 1.1 a 1.24, dependiendo de la novedad del proyecto, flexibilidad del desarrollo, procesos de resolución de riesgos utilizados, cohesión del equipo de desarrollo y nivel de madurez del proceso (véase el capítulo 26) de la organización. En la descripción del modelo posarquitectónico COCOMO II se estudia cómo calcular el valor de este exponente usando dichos parámetros.

Esto da por resultado el siguiente cálculo de esfuerzo:

$$\text{PM} = 2.94 \times \text{Tamaño}^{(1.1\ -\ 1.24)} \times M$$

donde

$$M = \text{PERS} \times \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{PREX} \times \text{FCIL} \times \text{SCED}$$

El multiplicador **M** se basa en siete atributos de proyecto y proceso que aumentan o disminuyen la estimación. Los atributos que se usan en el modelo de diseño temprano son fiabilidad y complejidad del producto (**RCPX**), reutilización requerida (**RUSE**), dificultad de plataforma (**PDIF**), habilidad personal (**PERS**), experiencia personal (**PREX**), calendario (**SCED**) y facilidades de soporte (**FCIL**). Estos atributos se explican en las páginas Web del libro. Los valores para dichos atributos se estiman mediante una escala de seis puntos, donde 1 corresponde a "muy bajo" y 6 corresponde a "muy alto".

El modelo de reutilización

Como se explicó en el capítulo 16, ahora es común la reutilización de software. La mayoría de los grandes sistemas incluyen una cantidad significativa de código que se reutilizó de proyectos de desarrollo previos. El modelo de reutilización se emplea para estimar el esfuerzo requerido al integrar código de reutilización o generado.

COCOMO II considera dos tipos de código de reutilización. El código "caja negra" es un código que puede reutilizarse sin comprender el código o hacerle cambios. Se considera que el esfuerzo de desarrollo para el código caja negra es cero. El código "caja blanca" tiene que adaptarse para integrarlo en un código nuevo u otros componentes de

reutilización. Para la reutilización se requiere esfuerzo de desarrollo, pues el código tiene que entenderse y modificarse antes de que pueda trabajar correctamente en el sistema.

Muchos sistemas incluyen código generado automáticamente de modelos de sistema, como se estudió en el capítulo 5. Se analiza un modelo (con frecuencia en UML) y se genera el código para implementar los objetos especificados en el modelo. El modelo de reutilización COCOMO II incluye una fórmula para estimar el esfuerzo requerido al integrar este código generado:

PM_{Auto} = (ASLOC × AT/100) / ATPROD // Estimación para código generado

ASLOC es el número total de líneas de código de reutilización, incluido el código que se genera automáticamente.

AT es el porcentaje de código de reutilización que se genera automáticamente.

ATPROD es la productividad de los ingenieros para integrar tal código.

Boehm y sus colaboradores (2000) midieron la **ATPROD** en aproximadamente 2,400 enunciados fuente por mes. Por lo tanto, si existe un total de 20 000 líneas de código fuente de reutilización en un sistema y el 30% de éste se genera automáticamente, entonces el esfuerzo que se requiere para integrar el código generado es:

(20,000 × 30/100) / 2400 = 2.5 meses-hombre // Código generado

Para estimar el esfuerzo requerido al integrar el código de reutilización de otros sistemas, se realiza un cálculo de esfuerzo separado. El modelo de reutilización no calcula directamente el esfuerzo a partir de una estimación del número de componentes de reutilización. En vez de ello, con base en el número de líneas de código que se reutilizan, el modelo ofrece una base para calcular el número equivalente de líneas de código nuevo (**ESLOC**). Éste se basa en el número de líneas de código de reutilización que deben cambiarse y un multiplicador que refleja la cantidad de trabajo que es necesario hacer para reutilizar los componentes. La fórmula que calcula **ESLOC** toma en cuenta el esfuerzo requerido para comprender el software, hacer cambios al código de reutilización y al sistema para integrar dicho código.

La siguiente fórmula se usa para calcular el número de líneas equivalentes de código fuente:

ESLOC = ASLOC × AAM

ESLOC es el número equivalente de líneas de nuevo código fuente.

ASLOC es el número de líneas de código en los componentes que deben cambiarse.

AAM es un Multiplicador de Ajuste de Adaptación, como se estudia a continuación.

La reutilización nunca es gratuita y se incurre en algunos costos aun si ninguna reutilización es posible. Sin embargo, los costos de reutilización disminuyen conforme aumenta la cantidad de código de reutilización. Los costos fijos de comprensión y valoración se dispersan a través de más líneas de código. El Multiplicador de Ajuste y Adaptación

Controladores de costos COCOMO II

Los controladores de costos (*cost drivers*) COCOMO II son atributos que reflejan algunos de los factores del producto, equipo, proceso y organización que afectan la cantidad de esfuerzo necesario para desarrollar un sistema de software. Por ejemplo, si se requiere un alto nivel de fiabilidad, será preciso un esfuerzo adicional; si hay demanda de entrega rápida, se requerirá esfuerzo adicional; si cambian los miembros del equipo, se solicitará esfuerzo adicional.

En el modelo COCOMO II hay 17 de estos atributos, a los que los desarrolladores del modelo asignaron valores.

http://www.SoftwareEngineering-9.com/Web/Planning/costdrivers.html

(AAM) ajusta la estimación para reflejar el esfuerzo adicional requerido para reutilizar el código. De manera simplista, AAM es la suma de tres componentes:

1. Un componente de adaptación (conocido como AAF) que representa los costos de hacer cambios al código de reutilización. El componente de adaptación incluye subcomponentes que toman en cuenta cambios al diseño, código e integración.
2. Un componente de comprensión (conocido como SU) que representa los costos para comprender el código a reutilizar y la familiaridad del ingeniero con el código. SU varía de 50 para código complejo no estructurado, a 10 para código orientado a objetos bien escrito.
3. Un factor de valoración (conocido como AA) que representa los costos de tomar la decisión de reutilizar. Esto es, siempre se requiere algún análisis para decidir si el código puede reutilizarse o no, y esto se incluye en el costo como AA. El factor AA varía de 0 a 8, dependiendo de la cantidad de esfuerzo de análisis requerido.

Si alguna adaptación del código puede hacerse automáticamente, esto reduce el esfuerzo requerido. Por lo tanto, la estimación se ajusta al evaluar el porcentaje de código adaptado automáticamente (AT) y usar esto para ajustar ASLOC. En consecuencia, la fórmula final es:

$$\text{ESLOC} = \text{ASLOC} \times (1 - \text{AT}/100) \times \text{AAM}$$

Una vez calculado ESLOC, se aplica la fórmula de estimación estándar para calcular el esfuerzo total requerido, en que el parámetro Tamaño = ESLOC. Entonces se suma esto al esfuerzo para integrar el código generado automáticamente ya calculado, lo que permite calcular el esfuerzo total requerido.

El nivel posarquitectónico

El modelo posarquitectónico es el más detallado de los modelos COCOMO II. Se usa una vez que está disponible un diseño arquitectónico inicial para el sistema, de manera que se conoce la estructura del subsistema. Entonces es posible hacer estimaciones para cada parte del sistema.

El punto de partida para las estimaciones producidas en el nivel posarquitectónico es la misma fórmula básica usada en las estimaciones de diseño temprano:

$$\text{PM} = \text{A} \times \text{Tamaño}^{\text{B}} \times \text{M}$$

Para esta etapa del proceso, hay que hacer una estimación más precisa del tamaño del proyecto conforme se conoce cómo se dividirá el sistema en objetos o módulos. Estas estimaciones del tamaño del código se hacen mediante tres parámetros:

1. Una estimación del número total de líneas de código nuevo a desarrollar (SLOC).
2. Una estimación de los costos de reutilización, con base en un número equivalente de líneas de código fuente (ESLOC), calculadas mediante el modelo de reutilización.
3. Una estimación del número de líneas de código que es probable se modifiquen debido a cambios a los requerimientos del sistema.

Los valores de estos parámetros se suman para calcular el tamaño de código total, en KSLOC, que se emplea en la fórmula de cálculo de esfuerzo. El componente final en la estimación, el número de líneas de código modificado, refleja el hecho de que los requerimientos de software siempre cambian. Esto conduce a la reelaboración y el desarrollo de código adicional, lo que debe tomarse en cuenta. Desde luego, con frecuencia habrá incluso más incertidumbre en esta cifra que en las estimaciones de código nuevo a desarrollar.

El término exponente (B) en la fórmula de cálculo de esfuerzo se relaciona con los niveles de complejidad del proyecto. Conforme los proyectos se hacen más complejos, los efectos de aumentar el tamaño del sistema se hacen más significativos. Sin embargo, las prácticas y los procedimientos organizacionales adecuados pueden controlar la deseconomía de escala que surge como consecuencia de aumentar la complejidad. Por lo tanto, el valor del exponente B se basa en cinco factores, como se muestra en la figura 23.12. Dichos factores se clasifican en una escala de seis puntos, de 0 a 5, donde 0 significa "extra alto" y 5 significa "muy bajo". Para calcular B, se suman las clasificaciones, se dividen entre 100 y los resultados se suman a 1.01 para obtener el exponente que debe usarse.

Por ejemplo, imagine que una organización acepta un proyecto en un dominio en el que tiene poca experiencia. El cliente del proyecto no tiene definido el proceso a usar ni ha asignado tiempo en el calendario del proyecto para el análisis del riesgo significativo. Un nuevo equipo de desarrollo debe reunirse para implementar este sistema. La organización dispuso recientemente un programa de mejoramiento de procesos y se clasificó como una organización de nivel 2 de acuerdo con la valoración de capacidad SEI, que se estudiará en el capítulo 26. Por consiguiente, los posibles valores para las clasificaciones empleadas en el cálculo del exponente son:

1. *Precedencia*, calificada baja (4). Éste es un proyecto nuevo para la organización.
2. *Flexibilidad de desarrollo*, clasificada muy alta (1). No hay involucramiento del cliente en el proceso de desarrollo, de manera que hay pocos cambios impuestos desde el exterior.

Factor de escala	Explicación
Precedencia	Refleja la experiencia previa de la organización con este tipo de proyectos. Muy bajo significa ninguna experiencia; extra alto significa que la organización está completamente familiarizada con este dominio de aplicación.
Flexibilidad de desarrollo	Refleja el grado de flexibilidad en el proceso de desarrollo. Muy bajo significa que se usa un proceso establecido; extra alto significa que el cliente sólo establece las metas generales.
Resolución arquitectura/ riesgo	Refleja la extensión de análisis de riesgos realizado. Muy bajo significa poco análisis; extra alto significa un análisis de riesgos completo y a profundidad.
Cohesión de equipo	Refleja cuán bien el equipo de desarrollo se conoce y trabaja en conjunto. Muy bajo significa interacciones muy difíciles; extra alto significa un equipo integrado y efectivo sin problemas de comunicación.
Madurez del proceso	Refleja la madurez del proceso de la organización. El cálculo de este valor depende del Cuestionario de Madurez CMM, pero puede lograrse una estimación al restar el nivel de madurez del proceso CMM de 5.

Figura 23.12 Factores de escala empleados en el cálculo de exponente en el modelo posarquitectónico

3. *Resolución de arquitectura/riesgo*, clasificada muy baja (5). No se ha realizado el análisis de riesgos.
4. *Cohesión del equipo*, clasificada nominal (3). Éste es un equipo nuevo, así que no hay información disponible acerca de la cohesión.
5. *Madurez del proceso*, clasificada nominal (3). Existe cierto control del proceso.

La suma de estos valores es 16. Entonces el exponente se calcula al dividir esto entre 100 y sumar el resultado a 0.01. Por lo tanto, el valor ajustado de B es 1.17.

La estimación del esfuerzo global se perfecciona usando un conjunto extenso de 17 atributos (controladores de costos) del producto, el proceso y la organización, en lugar de los siete atributos usados en el modelo de diseño temprano. Usted puede estimar los valores de estos atributos porque tiene más información acerca del software en sí, sus requerimientos no funcionales, el equipo de desarrollo y el proceso de desarrollo.

La figura 23.13 muestra cómo los atributos del controlador de costos pueden influir en las estimaciones del esfuerzo. En este libro se tomó un valor de 1.17 para el exponente, como se discutió en el ejemplo anterior, y se supone que RELY, CPLX, STOR, TOOL y SCED son los controladores de costos clave en el proyecto. Todos los otros controladores de costos tienen un valor nominal de 1, así que no afectan el cálculo del esfuerzo.

En la figura 23.13 se asignaron valores máximo y mínimo a los controladores de costos clave para mostrar cómo influye la estimación del esfuerzo. Los valores tomados son los del manual de referencia COCOMO II (Boehm, 2000). Como se observa, valores altos para los controladores de costos conducen a una estimación del esfuerzo que es más de tres veces la estimación inicial, mientras que los valores bajos reducen la estimación a aproximadamente un tercio del original. Esto destaca las diferencias significativas entre distintos tipos de proyectos y las dificultades de transferir la experiencia de un dominio de aplicación a otro.

Valor del exponente	1.17
Tamaño del sistema (incluidos factores para reutilización y volatilidad de requerimientos)	128 000 DSI
Estimación COCOMO inicial sin controladores de costos	**730 meses-hombre**
Fiabilidad	Muy alto, multiplicador = 1.39
Complejidad	Muy alto, multiplicador = 1.3
Restricción de memoria	Alto, multiplicador = 1.21
Uso de herramientas	Bajo, multiplicador = 1.12
Calendario	Acelerado, multiplicador = 1.29
Estimación COCOMO ajustada	**2,306 meses-hombre**
Fiabilidad	Muy bajo, multiplicador = 0.75
Complejidad	Muy bajo, multiplicador = 0.75
Restricción de memoria	Ninguno, multiplicador = 1
Uso de herramientas	Muy alto, multiplicador = 0.72
Calendario	Normal, multiplicador = 1
Estimación COCOMO ajustada	**295 meses-hombre**

Figura 23.13 Efecto de los controladores de costos sobre las estimaciones del esfuerzo

23.5.3 Duración del proyecto y asignación de personal

Además de estimar los costos globales de un proyecto y el esfuerzo que se requiere para desarrollar un sistema de software, los administradores de proyecto también deben estimar cuánto tardará el software en desarrollarse, y cuándo el personal necesitará trabajar en el proyecto. Cada vez más, las organizaciones demandan calendarios de desarrollo más cortos, de forma que sus productos puedan llegar al mercado antes que los de sus competidores.

El modelo COCOMO incluye una fórmula para estimar el tiempo calendario requerido para completar un proyecto:

$$TDEV = 3 \times (PM)^{(0.33 + 0.2*(B - 1.01))}$$

TDEV es el calendario nominal para el proyecto, en meses calendario, que ignora cualquier multiplicador relacionado con el calendario del proyecto.

PM es el esfuerzo calculado por el modelo COCOMO.

B es el exponente relacionado con la complejidad, como se estudió en la sección 23.5.2.

Si B = 1.17 y PM = 60, entonces

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ meses}$$

Sin embargo, no necesariamente son lo mismo el calendario de proyecto nominal predicho por el modelo COCOMO y el calendario requerido por el plan del proyecto. Puede haber un requerimiento para entregar el software más pronto o (muy rara vez) después de la fecha sugerida por el calendario nominal. Si el calendario se comprime, aumenta el esfuerzo requerido para el proyecto. El multiplicador SCED toma en cuenta esto en el cálculo de estimación del esfuerzo.

Suponga que un TDEV estimado del proyecto es de 13 meses, como se sugirió anteriormente, pero el calendario real requerido fue de 11 meses. Esto representa una compresión del calendario de aproximadamente 25 por ciento. Al usar los valores para el multiplicador SCED derivados por el equipo de Boehm, el multiplicador del esfuerzo para tal compresión de calendario es 1.43. Por lo tanto, el esfuerzo real que se requerirá si este calendario acelerado se cumple es casi 50% mayor que el esfuerzo requerido para entregar el software de acuerdo con el calendario nominal.

Existe una compleja relación entre el número de personas que trabajan en un proyecto, el esfuerzo que se dedicará a éste y el calendario de entrega del proyecto. Si cuatro personas logran completar un proyecto en 13 meses (es decir, 52 meses-hombre de esfuerzo), entonces se puede considerar que, al agregar una persona más, es posible completar el trabajo en 11 meses (55 meses-hombre de esfuerzo). Sin embargo, el modelo COCOMO sugiere que, de hecho, se necesitarán seis personas para terminar el trabajo en 11 meses (66 meses-hombre de esfuerzo).

La razón para esto es que agregar una persona reduce en realidad la productividad de los miembros del equipo existente y, por ende, el incremento real de esfuerzo agregado es menor que el que representa una persona. Conforme aumenta el tamaño del equipo del proyecto, los miembros del equipo pasan más tiempo comunicándose y definiendo las interfaces entre las partes del sistema desarrollado por otras personas. Por lo tanto, duplicar el número de personal (por ejemplo) no significa que la duración del proyecto se reducirá a la mitad. Si el equipo de desarrollo es grande, en ocasiones el hecho de agregar más personas al proyecto extiende, en vez de reducir, el calendario de desarrollo. Myers (1989) discute los problemas de aceleración del calendario. Él sugiere que es probable que los proyectos operen hacia problemas significativos si tratan de desarrollar software sin permitir suficiente tiempo de calendario para completar el trabajo.

No se puede simplemente estimar el número de personas requerido para un equipo de proyecto al dividir el esfuerzo total entre el tiempo contemplado en el calendario del proyecto requerido. Por lo general, al inicio del proyecto se necesita un pequeño número de personas para realizar el diseño inicial. Entonces el equipo crece hasta un nivel pico durante el desarrollo y las pruebas del sistema, y luego declina en tamaño conforme el sistema se prepara para su despliegue. Se ha demostrado que una acumulación muy rápida de personal del proyecto se correlaciona con atrasos en el calendario de éste. Por lo tanto, los administradores deben evitar agregar demasiado personal a un proyecto en etapas tempranas de su ciclo de vida.

Esta acumulación de esfuerzo puede modelarse con la curva de Rayleigh (Londeix, 1987). El modelo de estimación de Putnam (1978), que incorpora un modelo de dotación de personal a un proyecto, se basa en estas curvas de Rayleigh. Este modelo también incluye el tiempo de desarrollo como un factor clave. Conforme se reduce el tiempo de desarrollo, el esfuerzo requerido para desarrollar el sistema crece exponencialmente.

PUNTOS CLAVE

- El precio que se cobra por un sistema no depende sólo de sus costos de desarrollo estimados y la ganancia requerida por la compañía de desarrollo. Factores organizacionales pueden significar que el precio aumente para compensar el riesgo creciente o disminuya para conseguir ventaja competitiva.
- Con frecuencia al software se le asigna un precio para obtener un contrato y entonces la funcionalidad del sistema se ajusta para satisfacer el precio estimado.
- El desarrollo dirigido por un plan se organiza en torno a un plan de proyecto completo que define las actividades del proyecto, el esfuerzo planeado, el calendario de actividades y quién es responsable de cada actividad.
- La calendarización del proyecto implica la elaboración de varias representaciones gráficas de parte del plan del proyecto. Las gráficas de barras, las cuales muestran la duración de la actividad y los cronogramas de dotación de personal, son las representaciones de calendario que se usan con mayor frecuencia.
- Un hito de proyecto es un resultado predecible de una actividad o un conjunto de actividades. En cada hito debe presentarse a la administración un reporte formal de avance. Un entregable es un producto de trabajo que se entrega al cliente del proyecto.
- El juego de planeación XP implica a todo el equipo en la planeación del proyecto. El plan se desarrolla incrementalmente y, si surgen problemas, se ajustan de modo que la funcionalidad del software se reduzca en lugar de que demore la entrega de un incremento.
- Las técnicas de estimación para el software pueden basarse en la experiencia (en la que los administradores juzgan el esfuerzo requerido) o ser algorítmicas (el esfuerzo requerido se calcula a partir de otros parámetros de proyecto estimados).
- El modelo de costos COCOMO II es un modelo de costos algorítmico maduro que toma en cuenta atributos de proyecto, producto, hardware y personal cuando se formula una estimación del costo.

LECTURAS SUGERIDAS

Software Cost Estimation with COCOMO II. Éste es el libro fundamental sobre el modelo COCOMO II. Ofrece una descripción completa del modelo, con muchos ejemplos, e incluye software que implementa el modelo. Es ampliamente detallado y no una lectura ligera. (B. Boehm et al., Prentice Hall, 2000.)

"Ten unmyths of project estimation". Un artículo pragmático que analiza las dificultades prácticas de la estimación del proyecto y desafía algunas suposiciones fundamentales en esta área. (P. Armour, *Comm. ACM*, **45** (11), noviembre de 2002.)

Agile Estimating and Planning. Este libro es una descripción comprensible de la planeación basada en historias, como se usa en XP; además, ofrece fundamentos para usar un enfoque ágil en la planeación de proyectos. Además, incluye una buena introducción general de los conflictos en la planeación de proyectos. (M. Cohn, Prentice Hall, 2005.)

"Achievements and Challenges in Cocomo-based Software Resource Estimation". Este artículo presenta una historia de los modelos COCOMO y las influencias sobre dichos modelos, y examina las variantes de estos modelos que se han desarrollado. También identifica posibles desarrollos ulteriores en el enfoque COCOMO. (B. W. Boehm y R. Valeridi, *IEEE Software*, **25** (5), septiembre/octubre de 2008.) http://dx.doi.org/10.1109/MS.2008.133.

EJERCICIOS

23.1. ¿En qué circunstancias una compañía puede cobrar justificadamente por un sistema de software un precio mucho mayor que la estimación de costo del software más un margen de ganancia razonable?

23.2. Explique por qué el proceso de planeación de proyecto es iterativo y por qué un plan debe revisarse de manera continua durante un proyecto de software.

23.3. Explique brevemente el propósito de cada una de las secciones en un plan de proyecto de software.

23.4. Las estimaciones de costos son inherentemente riesgosas, sin importar la técnica de estimación empleada. Sugiera cuatro formas en las que pueda reducirse el riesgo en una estimación de costos.

23.5. La figura 23.14 establece algunas tareas, duraciones y dependencias. Dibuje una gráfica de barras que muestre el calendario del proyecto.

23.6. La figura 23.14 muestra las duraciones de las tareas para actividades de un proyecto de software. Suponga que ocurre un grave inconveniente no anticipado y que la tarea T5, en vez de tardar 10 días, tarda 40 días. Dibuje nuevas gráficas de barras que muestren cómo puede reorganizarse el proyecto.

23.7. El juego de planeación XP se basa en la noción de planeación para implementar las historias que representan los requerimientos del sistema. Explique los problemas potenciales con este enfoque cuando el software tiene alto rendimiento o requerimientos de confiabilidad.

23.8. Un administrador de software está encargado del desarrollo de un sistema de software crítico de protección, que está diseñado con la finalidad de controlar una máquina de radioterapia para tratar a pacientes que sufren de cáncer. Este sistema está embebido en la máquina y debe operar en un procesador de propósito especial con una cantidad fija de memoria (256 Mbytes). La máquina se comunica con un sistema de base de datos de pacientes para obtener los detalles del paciente y, después del tratamiento, registrar automáticamente en la base de datos la dosis de radiación administrada y otros detalles de tratamiento.

Se usa el método COCOMO para estimar el esfuerzo requerido al desarrollar este sistema y se calcula una estimación de 26 meses-hombre. Todos los multiplicadores de controlador de costos se establecieron en 1 cuando se hizo esta estimación.

Explique por qué debe ajustarse esta estimación para tomar en cuenta factores del proyecto, el personal, el producto y la organización. Sugiera cuatro factores que puedan tener efectos significativos sobre la estimación COCOMO inicial y proponga posibles valores para estos factores. Justifique por qué debe incluir cada factor.

Tarea	Duración (días)	Dependencias
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T2, T14
T16	10	T15

Figura 23.14 Ejemplo de calendarización

23.9. Algunos proyectos de software muy grandes implican escribir millones de líneas de código. Explique por qué los modelos de estimación de esfuerzo, como el COCOMO, pueden no funcionar bien cuando se aplican a sistemas muy grandes.

23.10. ¿Es ético que una compañía cotice un precio bajo para un contrato de software, a sabiendas de que los requerimientos son ambiguos y que pueden cobrar un precio alto por cambios posteriores solicitados por el cliente?

REFERENCIAS

Beck, K. (2000). *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. 2000. "COCOMO II Model Definition Manual". Center for Software Engineering, University of Southern California.
http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_ modelman2000.0.pdf.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. y Selby, R. (1995). "Cost models for future life cycle processes: COCOMO 2". *Annals of Software Engineering*, **1** 57–94.

Boehm, B. y Royce, W. (1989). "Ada COCOMO and the Ada Process Model". *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh: Software Engineering Institute.

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

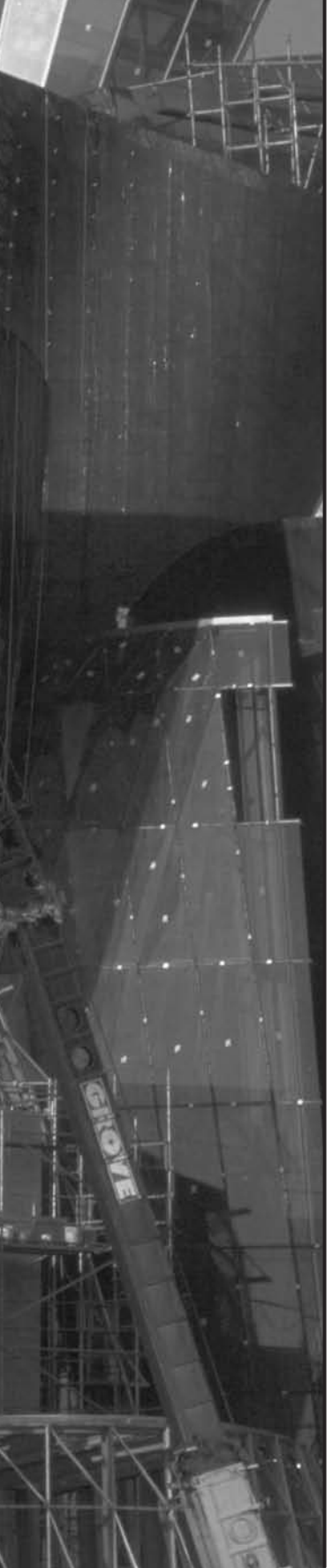
Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. y Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.

Londeix, B. (1987). *Cost Estimation for Software Development*. Wokingham: Addison-Wesley.

Myers, W. (1989). "Allow Plenty of Time for Large-Scale Software". *IEEE Software*, **6** (4), 92–9.

Putnam, L. H. (1978). "A General Empirical Solution to the Macro Software Sizing and Estimating Problem". *IEEE Trans. on Software Engineering*., **SE-4** (3), 345–61.

Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.



24

Gestión de la calidad

Objetivos

El objetivo de este capítulo es introducirlo a la gestión de calidad y la medición del software. Al estudiar este capítulo:

- estará al tanto del proceso de gestión de calidad y sabrá por qué es importante la planeación de calidad;
- comprenderá que la calidad del software se ve afectada por el proceso de desarrollo del software utilizado;
- conocerá la importancia de los estándares en el proceso de gestión de calidad y aprenderá cómo se usan los estándares en el aseguramiento de calidad;
- distinguirá la forma en que se utilizan las revisiones e inspecciones como mecanismo para garantizar la calidad del software;
- identificará cómo pueden ser útiles las mediciones en la valoración de algunos atributos de calidad del software y las limitaciones actuales de medición del software.

Contenido

24.1 Calidad del software
24.2 Estándares de software
24.3 Revisiones e inspecciones
24.4 Medición y métricas del software

Los problemas de calidad del software se descubrieron inicialmente en la década de 1960 con el desarrollo de los primeros grandes sistemas de software, y han continuado invadiendo la ingeniería de software a partir de esa década. El software entregado era lento y poco fiable, difícil de mantener y de reutilizar. El descontento con esta situación condujo a la adopción de técnicas formales de gestión de calidad del software, desarrolladas a partir de métodos usados en la industria manufacturera. Estas técnicas de gestión de calidad, en conjunto con nuevas tecnologías y mejores pruebas de software, llevaron a progresos significativos en el nivel general de calidad del software.

La gestión de calidad del software para los sistemas de software tiene tres intereses fundamentales:

1. A nivel de organización, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad. Esto supone que el equipo de gestión de calidad debe tener la responsabilidad de definir los procesos de desarrollo del software a usar, los estándares que deben aplicarse al software y la documentación relacionada, incluyendo los requerimientos, el diseño y el código del sistema.
2. A nivel del proyecto, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados; además, se ocupa de garantizar que los resultados del proyecto estén en conformidad con los estándares aplicables a dicho proyecto.
3. A nivel del proyecto, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto. El plan de calidad debe establecer metas de calidad para el proyecto y definir cuáles procesos y estándares se usarán.

Los términos *aseguramiento de calidad* y *control de calidad* se utilizan ampliamente en la industria manufacturera. El aseguramiento de calidad (QA, por las siglas de *quality assurance*) es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad y, en el proceso de fabricación, a la introducción de procesos de calidad. El control de calidad es la aplicación de dichos procesos de calidad para eliminar aquellos productos que no cuentan con el nivel requerido de calidad.

En la industria de software, diversas compañías y sectores industriales interpretan de maneras diferentes el aseguramiento de calidad y el control de calidad. En ocasiones, el aseguramiento de calidad representa simplemente la definición de procedimientos, procesos y estándares cuyo objetivo es asegurar el logro de calidad del software. En otros casos, el aseguramiento de calidad incluye también todas las actividades de gestión de configuración, verificación y validación aplicadas después de que un equipo de desarrollo entrega un producto. En este capítulo se usa el término *aseguramiento de calidad* para incluir verificación y validación, y los procesos de que la comprobación de procedimientos de calidad se aplicó de manera adecuada. Se evita el término "control de calidad", puesto que esta expresión no se usa mucho en la industria del software.

En la mayoría de las compañías, el equipo QA es el responsable de administrar el proceso de pruebas de liberación. Como se explicó en el capítulo 8, esto significa que se aplican las pruebas del software antes de que éste se libere a los clientes. El equipo es responsable de comprobar que las pruebas del sistema cubran los requerimientos y de mantener los registros adecuados del proceso de pruebas. Como en el capítulo 8 se estudiaron las pruebas de liberación, en este apartado no se trata este aspecto del aseguramiento de calidad.

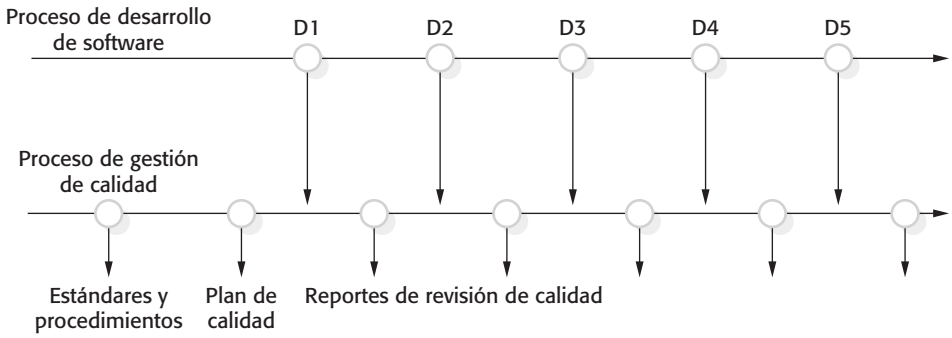


Figura 24.1
Gestión de calidad y desarrollo de software

La gestión de calidad proporciona una comprobación independiente sobre el proceso de desarrollo de software. El proceso de gestión de calidad verifica los entregables del proyecto para garantizar que sean consistentes con los estándares y las metas de la organización (figura 24.1). El equipo QA debe ser independiente del equipo de desarrollo para que pueda tener una perspectiva objetiva del software. Esto les permite reportar la calidad del software sin estar influidos por los conflictos de desarrollo del software.

De preferencia, el equipo de gestión de calidad no debe asociarse con algún grupo de desarrollo particular; sin embargo, tiene la responsabilidad ante toda la organización por la administración de la calidad. El equipo debe ser independiente y reportarse ante la administración ubicada sobre el nivel del administrador del proyecto. La razón es que los administradores de proyecto tienen que mantener el presupuesto y el calendario del proyecto. Si surgen problemas, pueden estar tentados a comprometer la calidad del producto para cumplir con el calendario. Un equipo de gestión de calidad independiente garantiza que las metas de calidad de la organización no estén comprometidas a corto plazo por consideraciones de presupuesto y calendario. Sin embargo, en compañías más pequeñas, esto es prácticamente imposible. La gestión de calidad y el desarrollo de software están inevitablemente vinculados con las personas que tienen responsabilidades tanto de desarrollo como de calidad.

La planeación de calidad es el proceso de desarrollar un plan de calidad para un proyecto. El plan de calidad debe establecer las cualidades deseadas de software y describir cómo se valorarán. Por lo tanto, define lo que realmente significa software de "alta calidad" para un sistema particular. Sin esta definición, los ingenieros pueden hacer diferentes suposiciones, algunas veces conflictivas, sobre cuáles atributos del producto reflejan las características de calidad más importantes. La planeación de calidad formalizada es parte integral de los procesos de desarrollo basados en un plan. No obstante, los métodos ágiles adoptan un enfoque menos formal para la gestión de calidad.

Humphrey (1989), en su clásico libro referente a la gestión del software, sugiere un bosquejo de estructura para un plan de calidad. Éste incluye:

1. *Introducción del producto* Una descripción del producto, la pretensión de su mercado y las expectativas de calidad para el producto.
2. *Planes del producto* Indican las fechas de entrega críticas y las responsabilidades para el producto, junto con planes para distribución y servicio al producto.

3. *Descripciones de procesos* Describen los procesos y estándares de desarrollo y servicio que deben usarse para diseño y gestión del producto.
4. *Metas de calidad* Las metas y los planes de calidad para el producto, incluyendo una identificación y justificación de los atributos esenciales de calidad del producto.
5. *Riesgos y gestión del riesgo* Los riesgos clave que pueden afectar la calidad del producto y las acciones a tomar para enfrentar dichos riesgos.

Los planes de calidad, que se desarrollan como parte del proceso de planeación general del proyecto, difieren en detalle dependiendo del tamaño y tipo de sistema que se desarrolló. Sin embargo, cuando escriba planes de calidad, debe tratar de mantenerlos tan breves como sea posible. Si el documento es demasiado amplio, las personas no lo leerán y, en consecuencia, se anulará el propósito de generar un plan de calidad.

Algunas personas consideran que la calidad del software puede lograrse mediante procesos establecidos basados en estándares de organización y procedimientos de calidad asociados que verifican el seguimiento de dichos estándares mediante el equipo de desarrollo de software. Su argumento es que los estándares se dirigen a la buena práctica de ingeniería de software y que seguir esta buena práctica conducirá a productos de alta calidad. No obstante, en la práctica, se considera que en la gestión de calidad hay mucho más que estándares y burocracia asociados para asegurar que éstos se sigan.

Aunque los estándares y procesos son importantes, los administradores de calidad deben enfocarse también a desarrollar una "cultura de calidad" en la que todo responsable del desarrollo del software se comprometa a lograr un alto nivel de calidad del producto. Deben exhortar a los equipos a asumir la responsabilidad de la calidad de su trabajo y desarrollar nuevos enfoques para el mejoramiento de la calidad. A pesar de que los estándares y procedimientos son la base de la gestión de calidad, los buenos administradores de calidad reconocen que existen aspectos intangibles a la calidad del software (elegancia, legibilidad, etcétera) que no pueden expresarse en estándares. Deben apoyar a la gente interesada en aspectos intangibles de calidad e impulsar el comportamiento profesional en todos los miembros del equipo.

La gestión de la calidad formalizada es particularmente importante para los equipos que diseñan grandes sistemas de larga duración, los cuales tardan varios años en desarrollarse. La documentación de la calidad es un registro de lo que cada subgrupo realizó en el proyecto. Ayuda a las personas a comprobar que no se olvidaron tareas importantes o que un grupo no hizo suposiciones incorrectas acerca de lo que hicieron otros grupos. La documentación de la calidad también es un medio de comunicación durante la vida del sistema. Permite a los grupos responsables de la evolución del sistema encontrar las pruebas y comprobaciones que el equipo de desarrollo debe implementar.

Para sistemas más pequeños, la gestión de calidad es aún importante, aunque puede adoptarse un enfoque más informal. No se necesita tanto papeleo, puesto que un equipo de desarrollo pequeño puede comunicarse de manera informal. El aspecto de calidad clave para el desarrollo de sistemas pequeños es establecer una cultura de calidad y asegurarse de que todos los miembros del equipo tienen un enfoque positivo sobre la calidad del software.

24.1 Calidad del software

La industria manufacturera estableció los fundamentos de la gestión de calidad para mejorar ésta en los productos que se fabricaban. Como parte de ello se desarrolló una definición de calidad, que se basa en la conformidad con una especificación de producto detallada (Crosby, 1979) y la noción de tolerancia. La suposición subyacente era que los productos podían especificarse por completo y establecerse procedimientos que comprobaran si un producto manufacturado cumplía o no con su especificación. Desde luego, los productos nunca cumplirán exactamente una especificación, pues se permite cierta tolerancia. Si el producto era "casi bueno", se clasificaba como aceptable.

La calidad del software no es directamente comparable con la calidad en la fabricación. La idea de tolerancia no es aplicable a los sistemas digitales y es prácticamente imposible llegar a una conclusión objetiva sobre si un sistema de software cumple o no su especificación, por las siguientes razones:

1. Como se explicó en el capítulo 4, referente a la ingeniería de requerimientos, es difícil escribir especificaciones de software completas y sin ambigüedades. Los desarrolladores y clientes de software pueden interpretar los requerimientos de diferentes formas y tal vez sea imposible llegar a acuerdos acerca de si el software se desarrolló conforme a su especificación.
2. Por lo general, las especificaciones integran requerimientos de varias clases de participantes. Dichos requerimientos son un compromiso ineludible y tal vez no incluyan los requerimientos de todos los grupos de participantes. Por lo tanto, las partes interesadas excluidas quizá perciban el sistema como uno de mala calidad, a pesar de que implementa los requerimientos acordados.
3. Es imposible medir de manera directa ciertas características de calidad (por ejemplo, mantenibilidad) y, por ende, no pueden especificarse plenamente sin ambigüedades. En la sección 24.4 se estudian las dificultades de la medición.

Debido a estos problemas, la valoración de calidad del software es un proceso subjetivo en que el equipo de gestión de calidad tiene que usar su juicio para decidir si se logró un nivel aceptable de calidad. El equipo de gestión de calidad debe considerar si el software se ajusta o no a su propósito pretendido. Esto implica responder preguntas sobre las características del sistema. Por ejemplo:

1. ¿En el proceso de desarrollo se siguieron los estándares de programación y documentación?
2. ¿El software se verificó de manera adecuada?
3. ¿El software es suficientemente confiable para utilizarse?
4. ¿El rendimiento del software es aceptable para uso normal?

Protección	Comprensibilidad	Portabilidad
Seguridad	Comprobabilidad	Usabilidad
Fiabilidad	Adaptabilidad	Reusabilidad
Flexibilidad	Modularidad	Eficiencia
Robustez	Complejidad	Facilidad para que el usuario aprenda a utilizarlo

Figura 24.2 Atributos de calidad del software

5. ¿El software es utilizable?
6. ¿El software está bien estructurado y es comprensible?

Existe la suposición general en la gestión de calidad del software de que el sistema se pondrá a prueba en contra de sus requerimientos. La decisión acerca de si entregar o no la funcionalidad requerida debe basarse en los resultados de dichas pruebas. Por lo tanto, el equipo QA debe revisar las pruebas que se desarrollaron y examinar los registros de pruebas para verificar que éstas se hayan realizado de manera apropiada. En algunas organizaciones el equipo de gestión de calidad es responsable de las pruebas del sistema, pero, en ocasiones, un grupo de pruebas de sistema separado es responsable de esto.

La calidad subjetiva de un sistema de software se basa principalmente en sus características no funcionales. Esto refleja la experiencia práctica del usuario: Si la funcionalidad del software no es lo que se esperaba, entonces los usuarios con frecuencia sólo le darán la vuelta a este asunto y encontrarán otras formas de hacer lo que quieren. Sin embargo, si el software no es fiable o resulta muy lento, entonces es prácticamente imposible que los usuarios logren sus metas.

Por consiguiente, la calidad del software no sólo se trata de si la funcionalidad de éste se implementó correctamente, sino también depende de los atributos no funcionales del sistema. Boehm y sus colaboradores (1978) indican que existen 15 importantes atributos de calidad de software, los cuales se listan en la figura 24.2. Dichos atributos se relacionan con la confiabilidad, usabilidad, eficiencia y mantenibilidad del software. Como se estudió en el capítulo 11, por lo general se considera que los atributos de confiabilidad son los atributos de calidad más importantes de un sistema. Sin embargo, también es significativo el rendimiento del software. Los usuarios rechazarán el software que sea demasiado lento.

No es posible que algún sistema se optimice para todos esos atributos; por ejemplo, mejorar la robustez puede conducir a pérdida de rendimiento. En consecuencia, el plan de calidad debe definir los atributos de calidad más importantes para el software que se desarrollará. Tal vez la eficiencia sea crítica y tengan que sacrificarse otros factores para que se logre esto. Si lo anterior se estableció en el plan de calidad, los ingenieros que trabajan en el desarrollo pueden cooperar para lograrlo. El plan debe incluir también una definición del proceso de valoración de la calidad. Ésta debe ser una forma acordada de valorar si cierto grado de calidad, como la mantenibilidad o robustez, está presente en el producto.

Una suposición que subyace en la gestión de la calidad del software es que la calidad del software se relaciona directamente con la calidad del proceso de desarrollo del

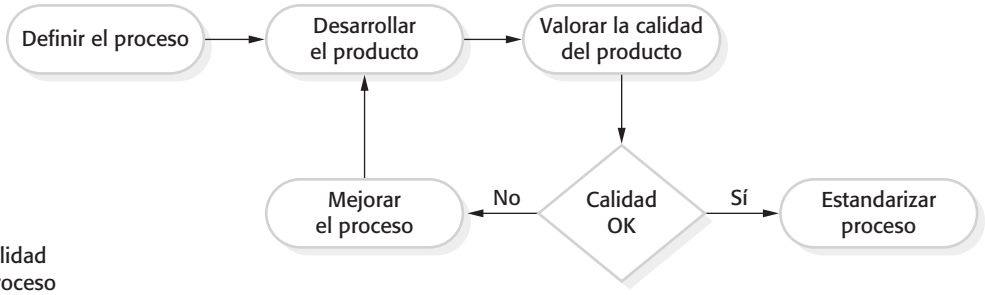


Figura 24.3 Calidad basada en el proceso

software. Esto proviene de nuevo de los sistemas fabriles, donde la calidad del producto está estrechamente relacionada con el proceso de producción. Un proceso de fabricación incluye configurar, establecer y operar las máquinas implicadas en el proceso. Una vez que las máquinas operan correctamente, se sigue de manera natural la calidad del producto. Entonces se mide la calidad del producto y el proceso se modifica hasta que se logra el nivel de calidad necesario. La figura 24.3 ilustra este enfoque basado en el proceso para obtener la calidad del producto.

En la manufactura existe un claro vínculo entre el proceso y la calidad del producto, ya que el proceso es relativamente sencillo de estandarizar y monitorizar. Una vez calibrados los sistemas de fabricación, pueden operar una y otra vez para generar productos de alta calidad; sin embargo, el software no se manufactura, se diseña. Por lo tanto, en el desarrollo del software es más compleja la relación entre calidad de proceso y calidad del producto. El diseño del software es un proceso creativo más que mecánico, pues es significativa la influencia de las habilidades y la experiencia individuales. Factores externos, como la novedad de una aplicación o la premura por el lanzamiento comercial de un producto, también afectan la calidad de éste sin importar el proceso usado.

No hay duda de que el proceso de desarrollo utilizado tiene una influencia importante sobre la calidad del software, y que los buenos procesos tienen más probabilidad de conducir a software de buena calidad. La gestión de la calidad y el mejoramiento del proceso pueden conducir a menores defectos en el software a desarrollar. Sin embargo, es difícil valorar los atributos de calidad del software, como la mantenibilidad, sin usar el software durante un largo periodo. En consecuencia, es difícil decir cómo las características del proceso influyen en dichos atributos. Más aún, debido al papel del diseño y la creatividad en el proceso de software, la estandarización del proceso en ocasiones puede exterminar la creatividad, lo cual, lejos de elevar la calidad, conducirá a un software de calidad inferior.

24.2 Estándares de software

Los estándares de software tienen una función muy importante en la gestión de calidad del software. Como se indicó, un aspecto importante del aseguramiento de calidad es la definición o selección de estándares que deben aplicarse al proceso de desarrollo de software o al producto de software. Como parte de este proceso QA, también pueden elegirse herramientas y métodos para apoyar el uso de dichos estándares. Una vez seleccionados éstos

Estándares de documentación

Los documentos del proyecto son una forma tangible de describir las diferentes representaciones de un sistema de software (requerimientos, UML, código, etcétera) y su proceso de producción. Los estándares de documentación definen la organización de diferentes tipos de documentos, así como el formato del documento. Son importantes porque facilitan la comprobación de que no se haya omitido material importante de los documentos y garantiza que los documentos del proyecto tengan una apariencia común. Los estándares pueden desarrollarse para el proceso de escribir documentos, los documentos en sí y el intercambio de documentos.

http://www.SoftwareEngineering-9.com/Web/QualityMan/docstandards.html

para su uso, deben definirse procesos específicos de proyecto para monitorizar el uso de los estándares y comprobar que éstos se siguieron.

Los estándares de software son importantes por tres razones:

1. Los estándares reflejan la sabiduría que es de valor para la organización. Se basan en conocimiento sobre la mejor o más adecuada práctica para la compañía. Con frecuencia, este conocimiento se adquiere sólo después de gran cantidad de ensayo y error. Configurarla dentro de un estándar, ayuda a la compañía a reutilizar esta experiencia y a evitar errores del pasado.
2. Los estándares proporcionan un marco para definir, en un escenario particular, lo que significa el término "calidad". Como se dijo, la calidad del software es subjetiva, y al usar estándares se establece una base para decidir si se logró un nivel de calidad requerido. Desde luego, esto depende del establecimiento de estándares que reflejen las expectativas del usuario para la confiabilidad, la usabilidad y el rendimiento del software.
3. Los estándares auxilian la continuidad cuando una persona retoma el trabajo iniciado por alguien más. Los estándares aseguran que todos los ingenieros dentro de una organización adopten las mismas prácticas. En consecuencia, se reduce el esfuerzo de aprendizaje requerido al iniciarse un nuevo trabajo.

Existen dos tipos de estándares de ingeniería de software relacionados que pueden definirse y usarse en la gestión de calidad del software:

1. *Estándares del producto* Se aplican al producto de software a desarrollar. Incluyen estándares de documentos (como la estructura de los documentos de requerimientos), estándares de documentación (como el encabezado de un comentario estándar para una definición de clase de objeto) y estándares de codificación, los cuales definen cómo debe usarse un lenguaje de programación.
2. *Estándares de proceso* Establecen los procesos que deben seguirse durante el desarrollo del software. Deben especificar cómo es una buena práctica de desarrollo. Los estándares de proceso pueden incluir definiciones de especificación, procesos de diseño y validación, herramientas de soporte de proceso y una descripción de los documentos que deben escribirse durante dichos procesos.

Estándares de producto	Estándares de proceso
Formato de revisión de diseño	Realizar revisión de diseño
Estructura de documento de requerimientos	Enviar nuevo código para construcción de sistema
Formato de encabezado por método	Proceso de liberación de versión
Estilo de programación Java	Proceso de aprobación del plan del proyecto
Formato de plan de proyecto	Proceso de control de cambio
Formato de solicitud de cambio	Proceso de registro de prueba

Figura 24.4 Estándares de producto y proceso

Los estándares deben entregar valor, en la forma de calidad aumentada del producto. No hay razón para definir estándares que sean costosos en términos de tiempo y esfuerzo, pues aplicarlos sólo conduce a mejoras secundarias en la calidad. Los estándares de producto deben diseñarse de forma que puedan aplicarse y comprobarse de manera efectiva en cuanto a costos, y los estándares de proceso deben incluir la definición de procesos que comprueben que se siguieron dichos estándares.

El desarrollo de estándares internacionales de ingeniería de software, por lo general, es un proceso prolongado en el que se reúnen los interesados en el estándar, elaboran borradores para comentar y, finalmente, acuerdan el estándar. Organismos nacionales e internacionales, como U.S. DoD, ANSI, BSI, OTAN y el IEEE, apoyan la determinación de estándares. Se trata de estándares generales que pueden aplicarse a través de varios proyectos. Entidades tales como la OTAN y otras organizaciones de defensa pueden requerir que sus propios estándares se usen en el desarrollo de contratos que suscriben con compañías de software.

Se han desarrollado estándares nacionales e internacionales que incluyen la terminología de ingeniería de software, lenguajes de programación como Java y C++, anotaciones como los símbolos de diagramación, procedimientos para derivar y escribir requerimientos de software, procedimientos de aseguramiento de calidad, y procesos de verificación y validación de software (IEEE, 2003). Estándares más especializados, como IEC 61508 (IEC, 1998), se desarrollaron para sistemas críticos de protección y seguridad.

Los equipos de gestión de calidad que elaboran estándares para alguna compañía, por lo general deben basar los estándares de dicha compañía en estándares nacionales e internacionales. Al usar estándares internacionales como punto de partida, el equipo de aseguramiento de calidad debe redactar un manual de estándares, el cual debe definir los estándares que necesita su organización. En la figura 24.4 se muestran ejemplos de estándares que podrían incluirse en dicho manual.

En ocasiones, los ingenieros de software consideran los estándares como demasiado prescriptivos y realmente poco relevantes para la actividad técnica del desarrollo de software. Esto es probable sobre todo cuando los estándares de proyecto requieren documentación y registro del trabajo tediosos. Aunque en general están de acuerdo con la necesidad de los estándares, los ingenieros encuentran a menudo razones para señalar que los estándares no necesariamente son adecuados para su proyecto particular. Para

minimizar el descontento y alentar la participación en los estándares, los administradores de calidad que establezcan los estándares deben dar los siguientes pasos:

1. *Involucrar a los ingenieros de software en la selección de estándares de producto* Si los desarrolladores comprenden por qué se seleccionaron los estándares, tienen más probabilidad de comprometerse con éstos. De preferencia, los documentos de estándares no deben establecer sólo el estándar a seguir, sino también deben incluir comentarios que expliquen por qué se tomaron las decisiones de estandarización.
2. *Revisar y modificar regularmente los estándares para reflejar las tecnologías cambiantes* Los estándares son costosos de desarrollar y tienden a guardarse como reliquias en un manual de estándares de una compañía. Debido a los costos y la discusión requeridos, muchas veces hay reticencia para cambiarlos. Aunque un manual de estándares es esencial, debe evolucionar para reflejar las circunstancias y la tecnología cambiantes.
3. *Ofrecer herramientas de software para dar soporte a los estándares* Los desarrolladores encuentran con frecuencia que los estándares son una pesadilla cuando la adhesión a ellos incluye un tedioso trabajo manual que podría hacerse mediante una herramienta de software. Si está disponible el soporte para herramientas, se requiere muy poco esfuerzo para seguir los estándares de desarrollo de software. Por ejemplo, los estándares de documento pueden implementarse mediante estilos de procesador de texto.

Diferentes tipos de software necesitan distintos procesos de desarrollo, puesto que los estándares deben ser adaptables. No hay razón para prescribir una forma particular de trabajar si es inadecuada para un proyecto o equipo de proyecto. Cada administrador de proyecto debe tener la autoridad de modificar los estándares de proceso de acuerdo con las circunstancias individuales. Sin embargo, cuando se hacen cambios, es importante garantizar que dichos cambios no conduzcan a una pérdida de calidad del producto. Esto afectará la relación de una empresa con sus clientes y conducirá probablemente a un aumento en los costos del proyecto.

El administrador del proyecto y el administrador de calidad pueden evitar problemas en los estándares al planear cuidadosamente la calidad oportuna en el proyecto. Deben decidir cuál de los estándares de la organización debe usarse sin cambio, cuáles deben modificarse y cuáles ignorarse. Es posible que deban crearse nuevos estándares en respuesta a requerimientos del cliente o del proyecto. Por ejemplo, tal vez se requieran estándares para especificaciones formales si no se han usado en proyectos anteriores.

24.2.1 El marco de estándares ISO 9001

Existe un conjunto internacional de estándares que pueden utilizarse en el desarrollo de los sistemas de administración de calidad en todas las industrias, llamado ISO 9000. Los estándares ISO 9000 pueden aplicarse a varias organizaciones, desde las industrias manufactureras hasta las de servicios. ISO 9001, el más general de dichos estándares, se aplica a organizaciones que diseñan, desarrollan y mantienen productos, incluido software. El estándar ISO 9001 se desarrolló originalmente en 1987, y su revisión más reciente fue en 2008.

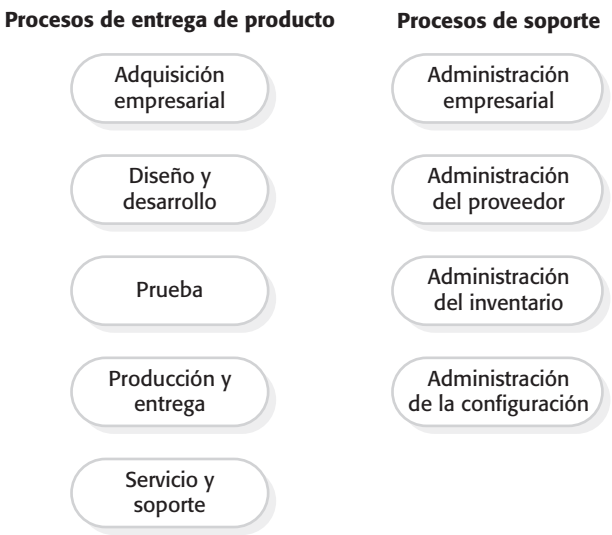


Figura 24.5 Procesos centrales ISO 9001

El estándar ISO 9001 no es en sí mismo un estándar para el desarrollo de software, sino un marco para elaborar estándares de software. Establece principios de calidad total, describe en general el proceso de calidad, y explica los estándares y procedimientos organizacionales que deben determinarse. Éstos tienen que documentarse en un manual de calidad de la organización.

La revisión principal del estándar ISO 9001 reorientó en 2000 el estándar hacia nueve procesos centrales (figura 24.5). Si una organización quiere estar conforme con el estándar ISO 9001, debe documentar cómo se relacionan sus procesos con dichos procesos centrales. También deberá definir y mantener registros que demuestren que se siguieron los procesos organizacionales establecidos. El manual de calidad de la compañía tiene que describir los procesos relevantes y los datos de proceso que deben recopilarse y conservarse.

El estándar ISO 9001 no define ni prescribe los procesos de calidad específicos que deben usarse en una compañía. Para estar de conformidad con ISO 9001, una compañía debe especificar los tipos de proceso que se muestran en la figura 24.5 y tener procedimientos que demuestren que se siguen sus procesos de calidad. Esto permite flexibilidad a través de sectores industriales y diversos tamaños de compañías. Pueden definirse estándares de calidad que sean adecuados para el tipo de software a desarrollar. Las compañías pequeñas pueden tener procesos no burocráticos y estar en conformidad con ISO 9001. Sin embargo, esta flexibilidad significa que no es posible hacer suposiciones sobre las similitudes o diferencias entre los procesos en distintas compañías que acatan ISO 9001. Algunas compañías tienen procesos de calidad muy rígidos con registros detallados, mientras que otras son mucho menos formales, con poca documentación adicional.

En la figura 24.6 se muestran las relaciones entre ISO 9001, manuales de calidad organizacional y planes de calidad de proyecto individuales. Este diagrama se derivó de un modelo presentado por Ince (1994), quien explica cómo puede usarse el estándar general ISO 9001 como base para procesos de gestión de calidad de software. Bamford y Dielbler (2003) explican cómo puede aplicarse el más reciente estándar ISO 9001:2000 en las compañías de software.

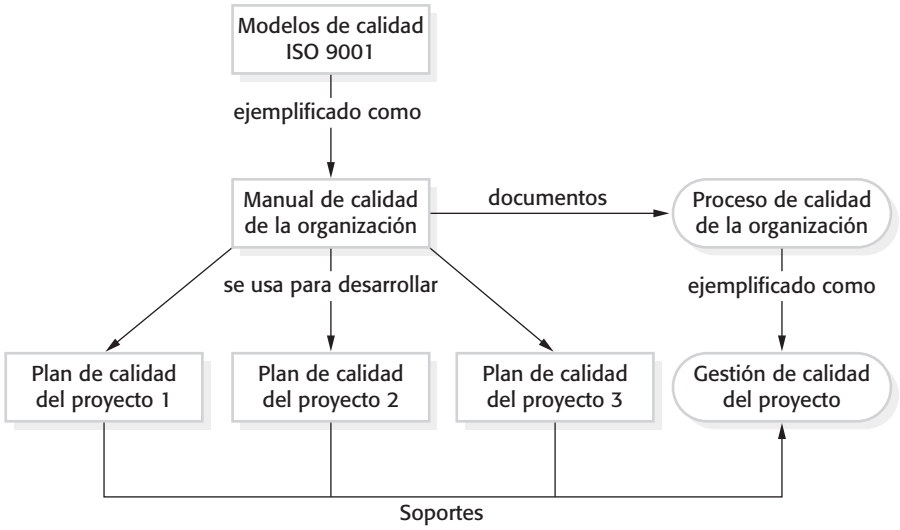


Figura 24.6
ISO 9001 y gestión de la calidad

Algunos clientes de software demandan que sus proveedores tengan la certificación ISO 9001. Así, los clientes podrán estar seguros de que la compañía que desarrolla el software tiene un sistema de gestión de calidad aprobado. Autoridades de acreditación independiente examinan los procesos de gestión de calidad y la documentación de proceso, y deciden si dichos procesos abarcan todas las áreas especificadas en ISO 9001. Si es así, certifican que los procesos de calidad de una compañía, definidos en el manual de calidad, concuerdan con el estándar ISO 9001.

Algunas personas consideran que la certificación ISO 9001 significa que la calidad del software producido por compañías certificadas será mejor que el derivado de compañías no certificadas. Esto no precisamente es cierto. El estándar ISO 9001 se enfoca en garantizar que la organización tenga procedimientos de gestión de calidad y que siga dichos procedimientos. No hay seguridad de que las compañías con certificación ISO 9001 empleen las mejores prácticas de desarrollo de software o que sus procesos conduzcan a software de alta calidad.

Por ejemplo, una compañía podría definir estándares de cobertura de pruebas que especifiquen que todos los métodos en los objetos deben llamarse al menos una vez. Lamentablemente, este estándar puede cumplirse mediante pruebas de software incompletas, que no incluyen pruebas con diferentes parámetros de métodos. En tanto se sigan los procedimientos de prueba definidos y se conserven registros de las pruebas realizadas, la compañía podría tener la certificación ISO 9001. Esta certificación define la calidad como la conformidad con estándares, y toma en cuenta la calidad como la advierten los usuarios del software.

Los métodos ágiles, que evitan la documentación y se enfocan en el código a desarrollar, tienen poco en común con los procesos de calidad formal que se examinan en ISO 9001. Se ha hecho cierto trabajo para reconciliar estos enfoques (Stalhane y Hanssen, 2008), pero la comunidad de desarrollo ágil en general se opone a lo que considera una carga burocrática de la conformidad con los estándares. Por esta razón, las compañías

que usan métodos de desarrollo ágil se preocupan pocas veces por la certificación ISO 9001.

24.3 Revisiones e inspecciones

Las revisiones e inspecciones son actividades QA que comprueban la calidad de los entregables del proyecto. Esto incluye examinar el software, su documentación y los registros del proceso para descubrir errores y omisiones, así como observar que se siguieron los estándares de calidad. Como se estudió en los capítulos 8 y 15, revisiones e inspecciones se usan junto con las pruebas del programa como parte del proceso general de verificación y validación del software.

Durante una revisión, un grupo de personas examinan el software y su documentación asociada en busca de problemas potenciales y la falta de conformidad con los estándares. El equipo de revisión realiza juicios informados sobre el nivel de calidad de un entregable de sistema o de proyecto. Entonces los administradores de proyecto pueden usar dichas valoraciones para tomar decisiones de planeación y asignar recursos al proceso de desarrollo.

Las revisiones de calidad se basan en documentos que se elaboraron durante el proceso de desarrollo del software. Al igual que las especificaciones, el diseño o el código del software, también pueden revisarse los modelos de proceso, planes de prueba, procedimientos de gestión de configuración, estándares de proceso y manuales de usuario. La revisión debe comprobar la coherencia e integridad de los documentos o el código objeto de prueba, y asegurarse de que se han seguido las normas de calidad.

Sin embargo, la revisión no sólo es acerca de la comprobación de conformidad con las normas, sino también se utiliza para ayudar a descubrir problemas y omisiones en la documentación del software o proyecto. Las conclusiones de la revisión deben registrarse formalmente como parte del proceso de gestión de calidad. Si se descubren problemas, los comentarios de los revisores deben pasar al autor del software o a quien resulte responsable de corregir los errores u omisiones.

El propósito de las revisiones e inspecciones es mejorar la calidad del software, no de valorar el rendimiento de los miembros del equipo de desarrollo. La revisión es un proceso público de detección de errores, comparado con el proceso más privado de prueba de componentes. Es necesario que los errores cometidos por los individuos se revelen a todo el equipo de programación. Para garantizar que todos los desarrolladores participen constructivamente con el proceso de revisión, los administradores de proyecto tienen que ser sensibles a las preocupaciones individuales. Deben desarrollar una cultura de trabajo que brinde apoyo y no culpar cuando se descubran errores.

Aunque una revisión de calidad ofrece a la administración datos sobre el software a desarrollar, las revisiones de calidad no son lo mismo que las revisiones de avance administrativo. Como se expuso en el capítulo 23, las revisiones de avance comparan el avance real en un proyecto de software frente al avance planeado. Su principal preocupación es si el proyecto entregará o no el software útil a tiempo y dentro del presupuesto. Las revisiones de progreso toman en cuenta factores externos, y circunstancias cambiantes pueden significar que el software en fase de desarrollo ya no se requiera o que tenga

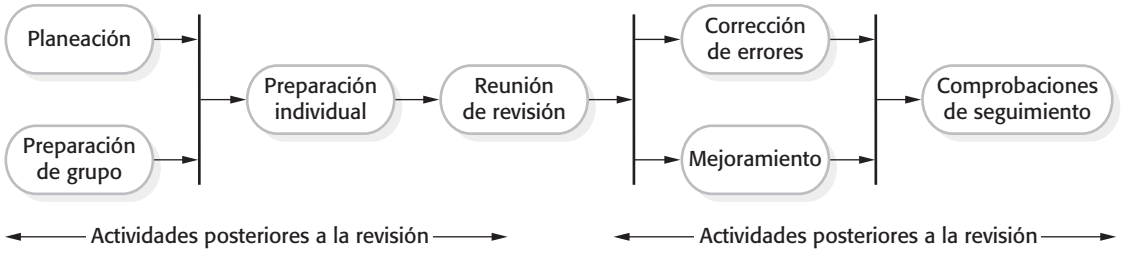


Figura 24.7
El proceso de revisión de software

que cambiarse radicalmente. Tal vez se cancelen los proyectos que desarrollaron software de alta calidad debido a cambios en la empresa o su entorno operacional.

24.3.1 El proceso de revisión

Aunque existen numerosas variaciones en los detalles de las revisiones, el proceso de revisión (figura 24.7) se estructura por lo general en tres fases:

1. *Actividades previas a la revisión* Se trata de actividades preparatorias esenciales para que sea efectiva la revisión. Por lo general, las actividades previas a la revisión se ocupan de la planeación y preparación de la revisión. La planeación de la revisión incluye establecer un equipo de revisión, organizar un tiempo, destinar un lugar para la revisión y distribuir los documentos a revisar. Durante la preparación de la revisión, el equipo puede reunirse para obtener un panorama del software a revisar. Miembros del equipo de revisión leen y entienden el software o los documentos y estándares relevantes. Trabajan de manera independiente para encontrar errores, omisiones y distanciamiento de los estándares. Si los revisores no pueden asistir a la reunión de revisión, pueden hacer sus comentarios por escrito acerca del software.

2. *La reunión de revisión* Durante la reunión de revisión un autor del documento o programa a revisar debe repasar el documento con el equipo de revisión. La revisión en sí debe ser relativamente corta, dos horas a lo sumo. Un miembro del equipo debe dirigir la revisión y otro registrar formalmente todas las decisiones y acciones de revisión a tomar. Durante la revisión, quien dirige es responsable de garantizar que se consideren todos los comentarios escritos. La dirección de la revisión debe firmar un registro de comentarios y acciones acordados durante la revisión.

3. *Actividades posteriores a la revisión* Después de terminada una reunión de revisión, deben tratarse los conflictos y problemas surgidos durante la revisión. Esto puede implicar corregir bugs de software, refactorizar el software de modo que esté conforme con los estándares de calidad, o reescribir los documentos. Algunas veces, los problemas descubiertos en una revisión de calidad son tales que es necesaria también una revisión administrativa con la finalidad de decidir si deben disponerse más recursos para corregirlos. Después de efectuar los cambios, la dirección de la revisión deberá comprobar que se hayan considerado todos los comentarios de la revisión. En ocasiones se requerirá una revisión ulterior para comprobar que los cambios realizados comprenden todos los comentarios de revisión anteriores.

Roles en el proceso de inspección

Cuando se estableció por primera vez una inspección del programa en IBM (Fagan, 1976; Fagan, 1986), se definieron algunos roles formales para los miembros del equipo de inspección. Éstos incluían: moderador, lector de código y secretario. Otros usuarios de la inspección modificaron dichos roles, pero generalmente se acepta que una inspección debe incluir al autor del código, un inspector, un secretario y un moderador que la dirija.

http://www.SoftwareEngineering-9.com/Web/QualityMan/roles.html

Con frecuencia, los equipos de revisión tienen un eje de tres o cuatro personas seleccionadas como revisores principales. Un miembro debe ser un diseñador ejecutivo, quien tendrá la responsabilidad de tomar decisiones técnicas significativas. Los revisores principales pueden invitar a otros miembros del proyecto, como los diseñadores de subsistemas relacionados, para contribuir a la revisión. Tal vez no participen en la revisión de todo el documento, pero deben concentrarse en aquellas secciones que afecten su trabajo. Como alternativa, el equipo de revisión puede hacer circular el documento y pedir comentarios por escrito de un amplio número de miembros del proyecto. El administrador del proyecto necesita participar en la revisión, a menos que se anticipen problemas que requieran cambios al plan del proyecto.

El proceso de revisión anterior se apoya en todos los miembros de un equipo de desarrollo asignado y disponible para una reunión de equipo. Sin embargo, ahora es común que los equipos de proyecto estén distribuidos, a veces a lo largo del país o en distintos continentes, así que muchas veces no es práctico que los miembros del equipo se reúnan en el mismo lugar. Ante tales situaciones, pueden usarse herramientas de edición de documentos para apoyar el proceso de revisión. Los miembros del equipo usan éstos para anotar comentarios en el documento o código fuente de software. Tales comentarios son visibles para otros miembros del equipo, quienes entonces podrán aprobarlos o rechazarlos. Sólo se requeriría una conversación telefónica cuando deban resolverse desacuerdos entre los revisores.

Por lo general, el proceso de revisión en el desarrollo de software ágil es informal. En Scrum, por ejemplo, hay una junta de revisión después de completar cada iteración del software (una revisión rápida), en la que pueden exponerse los conflictos y problemas de calidad. En la programación extrema, como se estudiará en la siguiente sección, la programación en grupos de dos personas garantiza que el código se examine y revise constantemente por otro miembro del equipo. Los conflictos de calidad general también se consideran en las reuniones diarias del equipo, pero XP se apoya en individuos que toman la iniciativa para mejorar y refactorizar el código. Por lo general, los enfoques ágiles no son dirigidos por estándares, de manera que no se consideran los asuntos de cumplimiento de estándares.

La falta de procedimientos de calidad formal en los métodos ágiles supone que puede haber problemas en el uso de enfoques ágiles en compañías que desarrollaron procedimientos de gestión de calidad detallados. Las revisiones de calidad en ocasiones aplazan el ritmo del desarrollo del software, así que éstas se emplean mejor dentro de un proceso de desarrollo dirigido por un plan. En este tipo de proceso, las revisiones pueden efectuarse mientras otros trabajos se realizan paralelamente. Esto no es práctico en enfoques ágiles que se centran de manera exclusiva en el desarrollo del código.

24.3.2 Inspecciones del programa

Las inspecciones del programa son "revisiones de pares" en las que los miembros del equipo colaboran para encontrar bugs en el programa en desarrollo. Como se explicó en el capítulo 8, las inspecciones pueden ser parte de los procesos de verificación y validación del software. Complementan las pruebas, puesto que no requieren la ejecución del programa. Esto quiere decir que es posible verificar versiones incompletas del sistema y comprobar representaciones como los modelos UML. Gilb y Graham (1993) sugieren que una de las formas más efectivas de usar las inspecciones es revisar los casos de prueba para un sistema. Las inspecciones permiten identificar problemas con las pruebas y, así, mejorar la efectividad de dichas pruebas en la detección de bugs de programa.

Las inspecciones del programa incluyen a miembros del equipo con diferentes antecedentes que realizan una cuidadosa revisión, línea por línea, del código fuente del programa. Buscan defectos y problemas, y los informan en una reunión de inspección. Los defectos pueden ser errores lógicos, anomalías en el código que indican una condición errónea o ciertas características que se hayan omitido del código. El equipo de revisión examina a detalle los modelos de diseño o el código del programa y destaca las anomalías y problemas a reparar.

Durante una inspección, con frecuencia se usa una lista de verificación de errores comunes de programación para enfocar la búsqueda de bugs. Esta lista de verificación se basa en ejemplos de libros, o bien, en el conocimiento de defectos normales en un dominio de aplicación común. Para diferentes lenguajes de programación se usan distintas listas de verificación, puesto que cada lenguaje tiene sus errores característicos. Humphrey (1989), en un amplio debate sobre inspecciones, ofrece algunos ejemplos de listas de verificación de inspección.

En la figura 24.8 se muestran las posibles comprobaciones que pueden hacerse durante el proceso de inspección. Gilb y Graham (1993) enfatizan que cada organización debe desarrollar su lista de verificación de inspección con base en estándares y prácticas locales. Dichas listas de verificación deben actualizarse regularmente, conforme se encuentren nuevos tipos de defectos. Los ítems en la lista de verificación varían según el lenguaje de programación, debido a diferentes niveles de comprobación posibles en el tiempo de compilación. Por ejemplo, un compilador Java comprueba que las funciones tengan el número correcto de parámetros; un compilador C no lo hace.

La mayoría de compañías que introdujeron las inspecciones descubrieron que éstas son muy efectivas para encontrar bugs. Fagan (1986) reportó que es posible detectar más del 60% de los errores en un programa mediante inspecciones informales de programa. Mills y sus colaboradores (1987) sugieren que un enfoque más formal a la inspección, con base en argumentos de exactitud, permite detectar más del 90% de los errores en un programa. McConnell (2004) compara las pruebas de unidad, en las que la tasa de detección de defectos es de alrededor del 25%, con las inspecciones, en las que la tasa de detección de defectos fue del 60%. También describe diversos estudios de caso, incluido un ejemplo en que la introducción de revisiones de pares condujo a un aumento en la productividad del 14% y una reducción en los defectos en el programa del 90 por ciento.

A pesar de su reconocida efectividad en términos de costos, muchas compañías de desarrollo de software se resisten a usar inspecciones o revisiones de pares. Los ingenieros de software con experiencia en pruebas de programa en ocasiones son reacios a aceptar que las inspecciones son más efectivas que las pruebas para la detección de defectos. Los administradores tal vez se muestren recelosos porque las inspecciones requieren

Clase de falla	Comprobación de inspección
Fallas de datos	• ¿Todas las variables del programa se inician antes de usar sus valores? • ¿Todas las constantes tienen nombre? • ¿La cota superior de los arreglos es igual al tamaño del arreglo o Valor – 1? • Si se usan cadenas de caracteres, ¿se asigna explícitamente un delimitador? • ¿Existe alguna posibilidad de desbordamiento de buffer?
Fallas de control	• Para cada enunciado condicional, ¿la condición es correcta? • ¿Hay certeza de que termine cada ciclo? • ¿Los enunciados compuestos están correctamente colocados entre paréntesis? • En caso de enunciados, ¿se justifican todos los casos posibles? • Si después de cada caso en los enunciados se requiere un paréntesis, ¿éste se incluyó?
Fallas de entrada/salida	• ¿Se usan todas las variables de entrada? • ¿A todas las variables de salida se les asigna un valor antes de que se produzcan? • ¿Entradas inesperadas pueden causar corrupción?
Fallas de interfaz	• ¿Todas las llamadas a función y método tienen el número correcto de parámetros? • ¿Los tipos de parámetro formal y real coinciden? • ¿Los parámetros están en el orden correcto? • Si los componentes acceden a memoria compartida, ¿tienen el mismo modelo de estructura de memoria compartida?
Fallas de gestión de almacenamiento	• Si se modifica una estructura vinculada, ¿todos los vínculos se reasignan correctamente? • Si se usa almacenamiento dinámico, ¿el espacio se asignó correctamente? • ¿El espacio se cancela explícitamente después de que ya no se requiere?
Fallas de gestión de excepción	• ¿Se tomaron en cuenta todas las posibles condiciones de error?

Figura 24.8 Lista de verificación de una inspección

costos adicionales durante el diseño y desarrollo. Quizá no quieran aceptar el riesgo de que no haya ahorros correspondientes en los costos de prueba del programa.

Los procesos ágiles pocas veces usan procesos de inspección formal o revisión de pares. En vez de ello, se apoyan en los miembros del equipo que cooperan para comprobar mutuamente el código y en lineamientos informales, tales como "comprobar antes de ingresar", lo que sugiere que los programadores deben comprobar su propio código. Los profesionales de la programación extrema argumentan que la programación en parejas es un sustituto efectivo de la inspección, ya que, en efecto, se trata de un proceso de inspección continuo. Dos personas observan cada línea de código y la comprueban antes de aceptarla.

La programación en grupos de dos conduce a un conocimiento profundo de un programa, pues ambos programadores deben entender su funcionamiento a detalle para continuar el desarrollo. En ocasiones es difícil lograr esta profundidad de conocimiento en otros procesos de inspección y, por lo tanto, la programación en grupos de dos permite

encontrar bugs que a veces no se descubrirían en inspecciones formales. Sin embargo, la programación en grupos de dos también puede conducir a malas interpretaciones de los requerimientos, en las que ambos miembros del par cometen el mismo error. Más aún, las parejas pueden tener reticencias para buscar errores, pues uno de los dos no quiere frenar el avance del proyecto. En ocasiones, las personas que participan no son tan objetivas como un equipo de inspección externo, y es probable que su habilidad para descubrir defectos esté comprometida por su cercana relación laboral.

24.4 Medición y métricas del software

La medición del software se ocupa de derivar un valor numérico o perfil para un atributo de un componente, sistema o proceso de software. Al comparar dichos valores unos con otros, y con los estándares que se aplican a través de una organización, es posible extraer conclusiones sobre la calidad del software, o valorar la efectividad de los procesos, las herramientas y los métodos de software.

Por ejemplo, suponga que una organización pretende introducir una nueva herramienta de prueba de software. Antes de introducir la herramienta, hay que registrar el número de defectos descubiertos de software en un tiempo determinado. Ésta es una línea de referencia para valorar la efectividad de la herramienta. Después de usar la herramienta durante algún tiempo, se repite este proceso. Si se descubren más defectos en el mismo lapso, después de introducida la herramienta, usted tal vez determine que ofrece apoyo útil para el proceso de validación del software.

La meta a largo plazo de la medición del software es usar la medición en lugar de revisiones para realizar juicios de la calidad del software. Al usar medición de software, un sistema podría valorarse preferentemente mediante un rango de métricas y, a partir de dichas mediciones, se podría inferir un valor de calidad del sistema. Si el software alcanzó un umbral de calidad requerido, entonces podría aprobarse sin revisión. Cuando es adecuado, las herramientas de medición pueden destacar también áreas del software susceptibles de mejora. Sin embargo, aún se está lejos de esta situación ideal y no hay señales de que la valoración automatizada de calidad será en el futuro una realidad previsible.

Una métrica de software es una característica de un sistema de software, documentación de sistema o proceso de desarrollo que puede medirse de manera objetiva. Los ejemplos de métricas incluyen el tamaño de un producto en líneas de código; el índice Fog (Gunning, 1962), que es una medida de la legibilidad de un pasaje de texto escrito; el número de fallas reportadas en un producto de software entregado, y el número de días-hombre requerido para desarrollar un componente de sistema.

Las métricas de software pueden ser métricas de control o de predicción. Como el nombre lo dice, las métricas de control apoyan la gestión del proceso, y las métricas de predicción ayudan a predecir las características del software. Las métricas de control se asocian por lo general con procesos de software. Ejemplos de las métricas de control o de proceso son el esfuerzo promedio y el tiempo requerido para reparar los defectos reportados. Las métricas de predicción se asocian con el software en sí y a veces se conocen como métricas de producto. Ejemplos de métricas de predicción son la complejidad ciclomática de un módulo (estudiado en el capítulo 8), la longitud promedio de los

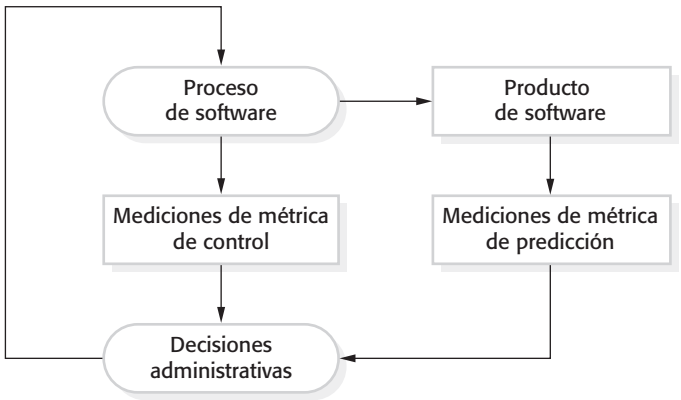


Figura 24.9 Mediciones de predicción y de control

identificadores de un programa, y el número de atributos y operaciones asociados con las clases de objetos en un diseño.

Tanto las métricas de control como las de predicción pueden influir en la toma de decisiones administrativas, como se muestra en la figura 24.9. Los administradores usan mediciones de proceso para decidir si deben hacerse cambios al proceso, y las métricas de predicción ayudan a estimar el esfuerzo requerido para hacer cambios al software. En este capítulo se estudian principalmente las métricas de predicción, cuyos valores se evalúan al analizar el código de un sistema de software. En el capítulo 26 se estudian las métricas de control y cómo se usan en el mejoramiento de procesos.

Existen dos formas en que pueden usarse las mediciones de un sistema de software:

1. *Para asignar un valor a los atributos de calidad del sistema* Al medir las características de los componentes del sistema, como su complejidad ciclomática, y luego agregar dichas mediciones, es posible valorar los atributos de calidad del sistema, tales como la mantenibilidad.

2. *Para identificar los componentes del sistema cuya calidad está por debajo de un estándar* Las mediciones pueden identificar componentes individuales con características que se desvían de la norma. Por ejemplo, es posible medir componentes para descubrir aquéllos con la complejidad más alta. Éstos tienen más probabilidad de tener bugs porque la complejidad los hace más difíciles de entender.

Lamentablemente, es difícil hacer mediciones directas de muchos de los atributos de calidad del software que se muestran en la figura 24.2. Los atributos de calidad, como mantenibilidad, comprensibilidad y usabilidad, son atributos externos que se refieren a cómo los desarrolladores y usuarios experimentan el software. Se ven afectados por factores subjetivos, como la experiencia y educación del usuario, y, por lo tanto, no pueden medirse de manera objetiva. Para hacer un juicio sobre estos atributos, hay que medir algunos atributos internos del software (como tamaño, complejidad, etcétera) y suponer que éstos se relacionan con las características de calidad por las que uno se interesa.

La figura 24.10 muestra algunos atributos externos de calidad del software y atributos internos que podrían, intuitivamente, relacionarse con ellos. Aunque el diagrama sugiere que pueden existir relaciones entre atributos externos e internos, no dice cómo se

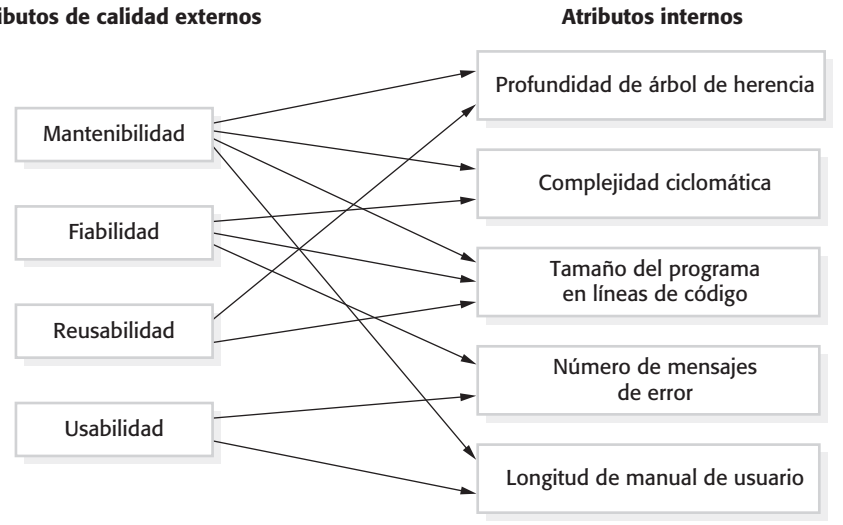


Figura 24.10
Relaciones entre software interno y externo

relacionan dichos atributos. Si la medida del atributo interno debe ser un factor de predicción útil de la característica externa del software, deben sostenerse tres condiciones (Kitchenham, 1990):

1. El atributo interno debe medirse con exactitud. Esto no siempre es un proceso directo y tal vez se requiera de herramientas de propósito especial para hacer las mediciones.
2. Debe existir una relación entre el atributo que pueda medirse y el atributo de calidad externo que es de interés. Esto es, el valor del atributo de calidad debe relacionarse, en alguna forma, con el valor del atributo que puede medirse.
3. Esta relación entre los atributos interno y externo debe comprenderse, validarse y expresarse en términos de una fórmula o un modelo. La formulación de un modelo implica identificar la manera funcional del modelo (lineal, exponencial, etcétera) mediante el análisis de datos recopilados, identificar los parámetros que se incluirán en el modelo, y calibrar dichos parámetros usando los datos existentes.

Los atributos de software internos, como la complejidad ciclomática de un componente, se miden usando herramientas de software que analizan el código fuente del software. Hay herramientas disponibles de fuente abierta que pueden utilizarse para hacer dichas mediciones. Aunque la intuición sugiere que podría existir una relación entre la complejidad de un componente de software y el número de fallas observadas en el uso, es difícil demostrar objetivamente que éste es el caso. Para probar esta hipótesis, se requieren datos de falla para un gran número de componentes y acceso al código fuente del componente para su análisis. Muy pocas compañías han establecido un compromiso a largo plazo para la recopilación de datos sobre su software, de manera que pocas veces están disponibles datos de fallas para el análisis.

En la década de 1990, numerosas y grandes compañías, como Hewlett-Packard (Grady, 1993), AT&T (Barnard y Price, 1994) y Nokia (Kilpi, 2001) introdujeron programas de métricas. Hicieron mediciones de sus productos y procesos y las usaron durante sus

procesos de gestión de calidad. La mayor parte de la atención se centró en la recolección de métricas sobre los defectos de programa y los procesos de verificación y validación. Offen y Jeffrey (1997) y Hall y Fenton (1997) tratan con más detalle la introducción en la industria de programas de métricas.

Existe escasa información disponible al público concerniente al uso actual en la industria de la medición sistemática del software. Muchas compañías reúnen información referente a su software, como el número de peticiones de cambio de requerimientos o el número de defectos descubiertos en las pruebas. Sin embargo, no es claro si usan entonces dichas mediciones de manera sistemática para comparar productos y procesos de software o para valorar el efecto de los cambios sobre los procesos y las herramientas de software. Existen algunas razones por las que esto se dificulta:

1. Es imposible cuantificar la rentabilidad de la inversión de introducir un programa de métricas organizacional. En años pasados existieron significativas mejoras en la calidad del software sin el uso de métricas, así que es difícil justificar los costos iniciales de introducir medición y valoración sistemáticas del software.
2. No hay estándares para las métricas de software o para los procesos estandarizados para medición y análisis. Muchas compañías son renuentes a introducir programas de medición hasta que se hallan disponibles tales estándares y herramientas de apoyo.
3. En gran parte de las compañías, los procesos de software no están estandarizados y se encuentran mal definidos y controlados. Por lo tanto, hay demasiada variabilidad de procesos dentro de la misma compañía para que las mediciones se usen en una forma significativa.
4. Buena parte de la investigación en la medición y métricas del software se enfoca en métricas basadas en códigos y procesos de desarrollo basados en un plan. Sin embargo, ahora cada vez más se desarrolla software mediante la configuración de sistemas ERP o COTS, o el uso de métodos ágiles. Por consiguiente, no se sabe si la investigación previa es aplicable a dichas técnicas de desarrollo de software.
5. La introducción de medición representa una carga adicional a los procesos. Esto contradice las metas de los métodos ágiles, los cuales recomiendan la eliminación de actividades de proceso que no están directamente relacionadas con el desarrollo de programas. En consecuencia, es improbable que las compañías que adoptaron los métodos ágiles aprueben un programa de métricas.

La medición y las métricas de software son la base de la ingeniería de software empírica (Endres y Rombach, 2003). Ésta es un área de investigación en la que se han usado experimentos respecto a los sistemas de software, y la recolección de datos referente a proyectos reales para formar y validar hipótesis sobre métodos y técnicas de ingeniería de software. Los investigadores que trabajan en esta área argumentan que sólo es posible confiar en el valor de los métodos y las técnicas de la ingeniería de software si se encuentra evidencia concreta de que en realidad ofrecen los beneficios que sugieren sus inventores.

Resulta lamentable que aun cuando es posible hacer mediciones objetivas y extraer conclusiones a partir de ellas, esto no necesariamente convence a quienes toman las decisiones. En vez de ello, la toma de decisiones está influida con frecuencia por factores

subjetivos, como la novedad, o la medida en que las técnicas son de interés para los profesionales. Por lo tanto, se considera que transcurrirán muchos años antes de que los resultados de la ingeniería de software empírica presenten un efecto significativo sobre la práctica de la ingeniería de software.

24.4.1 Métricas del producto

Las métricas del producto son métricas de predicción usadas para medir los atributos internos de un sistema de software. Los ejemplos de las métricas de productos incluyen el tamaño del sistema, la medida en líneas de código o el número de métodos asociados con cada clase de objeto. Por desgracia, como se explicó anteriormente en esta sección, las características del software que pueden medirse fácilmente, como el tamaño y la complejidad ciclomática, no tienen una relación clara y consistente con los atributos de calidad como comprensibilidad y mantenibilidad. Las relaciones varían dependiendo de los procesos de desarrollo, la tecnología empleada y el tipo de sistema a diseñar.

Las métricas del producto se dividen en dos clases:

1. Métricas dinámicas, que se recopilan mediante mediciones hechas de un programa en ejecución. Dichas métricas pueden recopilarse durante las pruebas del sistema o después de que el sistema está en uso. Un ejemplo es el número de reportes de bugs o el tiempo necesario para completar un cálculo.
2. Métricas estáticas, las cuales se recopilan mediante mediciones hechas de representaciones del sistema, como el diseño, el programa o la documentación. Ejemplos de mediciones estáticas son el tamaño del código y la longitud promedio de los identificadores que se usaron.

Estos tipos de métrica se relacionan con diferentes atributos de calidad. Las métricas dinámicas ayudan a valorar la eficiencia y fiabilidad de un programa. Las métricas estáticas ayudan a valorar la complejidad, comprensibilidad y mantenibilidad de un sistema de software o de los componentes del sistema.

Por lo general, existe una relación clara entre métricas dinámicas y características de calidad del software. Es muy sencillo medir el tiempo de ejecución requerido para funciones particulares y valorar el tiempo requerido con la finalidad de iniciar un sistema. Éstos se relacionan directamente con la eficiencia del sistema. De igual modo, el número de fallas del sistema y el tipo de fallas pueden registrarse y relacionarse directamente con la fiabilidad del software, que se estudió en el capítulo 15.

Como se comentó, las métricas estáticas, como las que se muestran en la figura 24.11, tienen una relación indirecta con los atributos de calidad. Se ha propuesto una gran cantidad de diferentes métricas y se han intentado muchos experimentos para derivar y validar las relaciones entre dichas métricas y atributos como complejidad y mantenibilidad. Ninguno de tales experimentos ha sido concluyente, pero el tamaño del programa y la complejidad del control parecen ser los factores de predicción más fiables de la comprensibilidad, la complejidad del sistema y la mantenibilidad.

Las métricas de la figura 24.11 son aplicables a cualquier programa, pero también se han propuesto métricas más específicas orientadas a objetos (OO). La figura 24.12 resume la suite de Chidamber y Kemerer (en ocasiones llamada suite CK) de seis métricas

Métrica de software	Descripción
Fan-in/Fan-out	Fan-in (abanico de entrada) es una medida del número de funciones o métodos que llaman a otra función o método (por ejemplo, X). Fan-out (abanico de salida) es el número de funciones a las que llama la función X. Un valor alto para fan-in significa que X está estrechamente acoplado con el resto del diseño y que los cambios a X tendrán extensos efectos dominó. Un valor alto de fan-out sugiere que la complejidad global de X puede ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.
Longitud de código	Ésta es una medida del tamaño de un programa. Por lo general, cuanto más grande sea el tamaño del código de un componente, más probable será que el componente sea complejo y proclive a errores. Se ha demostrado que la longitud del código es una de las métricas más fiables para predecir la proclividad al error en los componentes.
Complejidad ciclomática	Ésta es una medida de la complejidad del control de un programa. Tal complejidad del control puede relacionarse con la comprensibilidad del programa. En el capítulo 8 se estudia la complejidad ciclomática.
Longitud de identificadores	Ésta es una medida de la longitud promedio de los identificadores (nombres para variables, clases, métodos, etcétera) en un programa. Cuanto más largos sean los identificadores, es más probable que sean significativos y, por ende, más comprensible será el programa.
Profundidad de anidado condicional	Ésta es una medida de la profundidad de anidado de los enunciados if en un programa. Los enunciados if profundamente anidados son difíciles de entender y proclives potencialmente a errores.
Índice Fog	Ésta es una medida de la longitud promedio de las palabras y oraciones en los documentos. Cuanto más alto sea el valor del índice Fog de un documento, más difícil será entender el documento.

Figura 24.11 Métricas estáticas de productos de software

orientadas a objetos (1994). Aunque se propusieron originalmente a principio de la década de 1990, aún son las métricas OO de más amplio uso. Algunas herramientas de diseño UML recopilan automáticamente valores para dichas métricas conforme se crean los diagramas UML.

El-Amam (2001) hace una excelente revisión de las métricas orientadas a objetos, analiza las métricas CK y otras métricas OO, y concluye que todavía no se tiene suficiente evidencia para comprender cómo estas y otras métricas orientadas a objetos se relacionan con cualidades externas de software. Esta situación no ha cambiado realmente desde su análisis en 2001. Todavía no se sabe cómo usar las mediciones de los programas orientados a objetos para extraer conclusiones fiables acerca de su calidad.

24.4.2 Análisis de componentes de software

En la figura 24.13 se ilustra un proceso de medición que puede ser parte de un proceso de valoración de calidad del software. Cada componente del sistema puede analizarse por separado mediante un rango de métricas. Los valores de dichas métricas pueden compararse entonces para diferentes componentes y, tal vez, con datos de medición históricos

Métrica orientada a objetos	Descripción
Métodos ponderados por clase (*weighted methods per class*, WMC)	Éste es el número de métodos en cada clase, ponderado por la complejidad de cada método. Por lo tanto, un método simple puede tener una complejidad de 1, y un método grande y complejo tendrá un valor mucho mayor. Cuanto más grande sea el valor para esta métrica, más compleja será la clase de objeto. Es más probable que los objetos complejos sean más difíciles de entender. Tal vez no sean lógicamente cohesivos, por lo que no pueden reutilizarse de manera efectiva como superclases en un árbol de herencia.
Profundidad de árbol de herencia (*depth of inheritance tree*, DIT)	Esto representa el número de niveles discretos en el árbol de herencia en que las subclases heredan atributos y operaciones (métodos) de las superclases. Cuanto más profundo sea el árbol de herencia, más complejo será el diseño. Es posible que tengan que comprenderse muchas clases de objetos para entender las clases de objetos en las hojas del árbol.
Número de hijos (*number of children*, NOC)	Ésta es una medida del número de subclases inmediatas en una clase. Mide la amplitud de una jerarquía de clase, mientras que DIT mide su profundidad. Un valor alto de NOC puede indicar mayor reutilización. Podría significar que debe realizarse más esfuerzo para validar las clases base, debido al número de subclases que dependen de ellas.
Acoplamiento entre clases de objetos (*coupling between object classes*, CBO)	Las clases están acopladas cuando los métodos en una clase usan los métodos o variables de instancia definidos en una clase diferente. CBO es una medida de cuánto acoplamiento existe. Un valor alto para CBO significa que las clases son estrechamente dependientes y, por lo tanto, es más probable que el hecho de cambiar una clase afecte a otras clases en el programa.
Respuesta por clase (*response for a class*, RFC)	RFC es una medida del número de métodos que potencialmente podrían ejecutarse en respuesta a un mensaje recibido por un objeto de dicha clase. Nuevamente, RFC se relaciona con la complejidad. Cuanto más alto sea el valor para RFC, más compleja será una clase y, por ende, es más probable que incluya errores.
Falta de cohesión en métodos (*lack of cohesion in methods*, LCOM)	LCOM se calcula al considerar pares de métodos en una clase. LCOM es la diferencia entre el número de pares de método sin compartir atributos y el número de pares de método con atributos compartidos. El valor de esta métrica se debate ampliamente y existe en muchas variaciones. No es claro si realmente agrega alguna información útil además de la proporcionada por otras métricas.

Figura 24.12 Suite de métricas CK orientadas a objetos

recopilados en proyectos anteriores. Las mediciones anómalas, que se desvían significativamente de la norma, pueden implicar que existen problemas con la calidad de dichos componentes.

Las etapas clave en este proceso de medición de componentes son:

1. *Elegir las mediciones a realizar* Deben formularse las preguntas que la medición busca responder, y definir las mediciones requeridas para responder a tales preguntas. Deben recopilarse las mediciones que no son directamente relevantes para dichas preguntas. El paradigma GQM (por las siglas de *Goal-Question-Metric*, es decir, Meta-Pregunta-Métrica) de Basili (Basili y Rombach, 1988), que se estudia

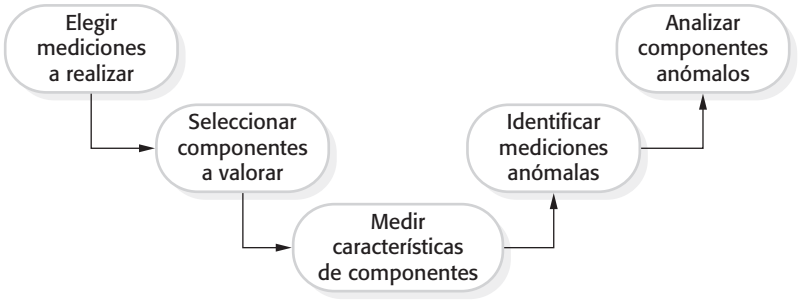


Figura 24.13 El proceso de medición de producto

en el capítulo 26, es un enfoque adecuado cuando se decide cuáles datos hay que recopilar.

2. *Seleccionar componentes a valorar* Probablemente usted no necesite estimar valores métricos para todos los componentes en un sistema de software, ya que en ocasiones podrá seleccionar una muestra representativa de componentes para medición, que le permitirá realizar una valoración global de la calidad del sistema. En otras circunstancias, tal vez desee enfocarse en los componentes centrales del sistema que están casi en uso constante. La calidad de dichos componentes es más importante que la de aquellos componentes que sólo se usan muy pocas veces.

3. *Medir las características de los componentes* Se miden los componentes seleccionados y se calculan los valores de métrica asociados. Por lo general, esto implica procesar la representación de los componentes (diseño, código, etcétera) mediante una herramienta de recolección automatizada de datos. Esta herramienta puede escribirse especialmente o ser una característica de las herramientas de diseño que ya están en uso.

4. *Identificar mediciones anómalas* Después de hacer las mediciones de componentes, se comparan entonces unas con otras y con mediciones anteriores que se hayan registrado en una base de datos de mediciones. Hay que observar los valores inusualmente altos o bajos para cada métrica, pues éstos sugieren que podría haber problemas con el componente que muestra dichos valores.

5. *Analizar componentes anómalos* Cuando identifique los componentes con valores anómalos para sus métricas seleccionadas, debe examinarlos para decidir si dichos valores de métrica anómalos significan que la calidad del componente se encuentra o no comprometida. Un valor de métrica anómalo para la complejidad (al parecer) no necesariamente significa un componente de mala calidad. Podría haber alguna otra razón para el valor alto, por lo que no necesariamente significa que haya problemas con la calidad del componente.

Siempre es conveniente mantener datos recopilados como un recurso organizacional, así como registros históricos de todos los proyectos aun cuando no se hayan usado durante un proyecto particular. Una vez establecida una base suficientemente grande de datos de medición, será posible hacer comparaciones de calidad de software a través de proyectos, además de validar las relaciones entre atributos de componentes internos y características de calidad.

24.4.3 Ambigüedad de mediciones

Cuando reúna datos cuantitativos relativos al software y los procesos de software, deberá analizar dichos datos para entender su significado. Es fácil malinterpretar los datos y hacer inferencias incorrectas. No basta con observar los datos por sí mismos, sino que también hay que considerar el contexto donde se recaban los datos.

Para ilustrar cómo pueden interpretarse los datos recopilados en diferentes formas, considere el siguiente escenario, que se ocupa del número de peticiones de cambio hechas por los usuarios de un sistema:

> *Una administradora decide monitorizar el número de peticiones de cambio enviadas por los clientes, con base en una suposición de que existe una relación entre dichas peticiones de cambio y la usabilidad y conveniencia del producto. Ella supone que cuanto más alto sea el número de peticiones de cambio, menos cumple el software las necesidades del cliente.*
>
> *Es costoso manejar las peticiones de cambio y modificar el software. Por lo tanto, la organización decide cambiar su proceso con la intención de mejorar la satisfacción del cliente y, al mismo tiempo, reducir los costos de hacer cambios. La intención es que el cambio de proceso dará como resultado mejores productos y menos peticiones de cambio.*
>
> *Los cambios de proceso se inician para aumentar la inclusión del cliente en el proceso de diseño del software. Se introducen pruebas beta de todos los productos; además, se incorporan en el producto entregado las modificaciones solicitadas por el cliente. Se entregan las nuevas versiones de los productos, que se desarrollan mediante este proceso modificado. En algunos casos se reduce el número de peticiones de cambio, aunque en otros aumenta. La administradora está confundida y descubre que es imposible valorar los efectos de los cambios de proceso sobre la calidad del producto.*

Para comprender por qué puede ocurrir este tipo de ambigüedad, hay que conocer las razones por las que los usuarios pueden hacer peticiones de cambio:

1. El software no es lo bastante bueno y no hace lo que quieren los clientes. Por lo tanto, solicitan cambios para obtener la funcionalidad que ellos requieren.
2. Como alternativa, el software puede ser muy bueno y, por consiguiente, se usa amplia e intensamente. Las peticiones de cambio pueden generarse porque existen muchos usuarios de software que piensan creativamente en nuevas ideas que podrían hacer con el software.

Por ende, aumentar la participación del cliente en el proceso puede reducir el número de peticiones de cambio para los productos con los que los clientes están descontentos. Los cambios de proceso han sido efectivos y han hecho al software más útil y adecuado. Sin embargo, alternativamente, los cambios al proceso pueden no haber funcionado, y los clientes tal vez decidieron buscar un sistema opcional. El número de peticiones de cambio disminuye porque el producto perdió participación en el mercado frente a un producto rival y, en consecuencia, hay menos usuarios del producto.

Por otra parte, los cambios al proceso pueden conducir a muchos nuevos clientes satisfechos que deseen participar en el proceso de desarrollo del producto. Por lo tanto, generan más peticiones de cambio. Los cambios al proceso de manejar las peticiones de cambio contribuyen a este aumento. Si la compañía tiene mayor capacidad de respuesta con los clientes, ellos generarán más peticiones de cambio porque saben que éstas se tomarán con seriedad. Creen que sus sugerencias se incorporarán quizás en versiones posteriores del software. O bien, el número de peticiones de cambio puede aumentar porque los sitios de prueba beta no eran los típicos del mayor uso del programa.

Para analizar los datos de petición de cambio, no basta con conocer el número de peticiones de cambio, sino que se necesita conocer quién hizo la petición, cómo usa el software y por qué hizo la petición. También se requiere información sobre los factores externos, como modificaciones al procedimiento de petición de cambio o cambios al mercado que puedan tener un efecto. Con esta información, es posible averiguar si los cambios al proceso fueron efectivos para aumentar la calidad del producto.

Esto ilustra las dificultades de entender los efectos de los cambios, y el enfoque "científico" a este problema es reducir el número de factores que tiendan a afectar las mediciones hechas. Sin embargo, los procesos y productos que se miden no están aislados de su entorno. El ambiente empresarial cambia constantemente y es imposible evitar los cambios a la práctica laboral sólo porque pueden hacerse comparaciones de datos inválidos. Como tales, los datos cuantitativos sobre las actividades humanas no siempre deben tomarse en serio. Las razones por las que cambia un valor medido con frecuencia son ambiguas. Dichas razones deben investigarse a profundidad antes de extraer conclusiones de cualquier medición que se haya realizado.

PUNTOS CLAVE

- La gestión de calidad del software se ocupa de garantizar que el software tenga un número menor de defectos y que alcance los estándares requeridos de mantenibilidad, fiabilidad, portabilidad, etcétera. Incluye definir estándares para procesos y productos, y establecer procesos para comprobar que se siguieron dichos estándares.
- Los estándares de software son importantes para el aseguramiento de la calidad, pues representan una identificación de las "mejores prácticas". Al desarrollar el software, los estándares proporcionan un cimiento sólido para diseñar software de buena calidad.
- Es necesario documentar un conjunto de procedimientos de aseguramiento de la calidad en un manual de calidad organizacional. Esto puede basarse en el modelo genérico para un manual de calidad sugerido en el estándar ISO 9001.
- Las revisiones de los entregables del proceso de software incluyen a un equipo de personas que verifican que se siguieron los estándares de calidad. Las revisiones son la técnica usada más ampliamente para valorar la calidad.
- En una inspección de programa o revisión de pares, un reducido equipo comprueba sistemáticamente el código. Ellos leen el código a detalle y buscan posibles errores y omisiones. Entonces los problemas detectados se discuten en una reunión de revisión del código.

- La medición del software puede usarse para recopilar datos cuantitativos tanto del software como del proceso de software. Se usan los valores de las métricas de software recopilados para hacer inferencias referentes a la calidad del producto y el proceso.
- Las métricas de calidad del producto son particularmente útiles para resaltar los componentes anómalos que pudieran tener problemas de calidad. Dichos componentes deben entonces analizarse con más detalle.

LECTURAS SUGERIDAS

Metrics and Models for Software Quality Engineering, 2nd edition. Éste es un análisis muy completo de las métricas del software que incluyen métricas de proceso, de producto y orientadas a objetos. También contiene cierto conocimiento matemático requerido para desarrollar y comprender modelos basados en medición de software. (S. H. Kan, Addison-Wesley, 2003.)

Software Quality Assurance: From Theory to Implementation. Un excelente vistazo actualizado a los principios y la práctica del aseguramiento de la calidad del software. Incluye un análisis de los estándares, como el ISO 9001. (D. Galin, Addison-Wesley, 2004.)

"A Practical Approach for Quality-Driven Inspections". En la actualidad muchos artículos concernientes a las inspecciones son más bien anticuados, ya que no consideran la práctica moderna del desarrollo de software. Este texto relativamente reciente describe un método de inspección que se ocupa de algunos de los problemas al utilizar la inspección y sugiere cómo pueden usarse las inspecciones en un entorno moderno de desarrollo. (C. Denger, F. Shull, *IEEE Software*, **24** (2), marzo-abril de 2007.) http://dx.doi.org/10.1109/MS.2007.31

"Misleading Metrics and Unsound Analyses". Un excelente artículo de los principales investigadores de métricas, quienes analizan las dificultades de comprensión de lo que significan realmente las métricas. (B. Kitchenham, R. Jeffrey y C. Connaughton, *IEEE Software*, **24** (2), marzo-abril de 2007.) http://dx.doi.org/10.1109/MS.2007.49.

"The Case for Quantitative Project Management". Ésta es una introducción a una sección especial de la revista que incluye otros dos artículos sobre administración cuantitativa de proyectos. Plantea razones para una mayor investigación en métricas y medición con la finalidad de mejorar la administración de proyectos de software. (B. Curtis et al., *IEEE Software*, **25** (3), mayo-junio de 2008.) http://dx.doi.org/10.1109/MS.2008.80.

EJERCICIOS

24.1. Explique por qué un proceso de software de alta calidad debería conducir a productos de alta calidad de software. Discuta los posibles problemas con este sistema de gestión de calidad.

24.2. Exponga cómo pueden usarse los estándares para obtener conocimiento de la organización sobre los métodos efectivos de desarrollo de software. Sugiera cuatro tipos de conocimiento que puedan reflejarse en los estándares de la organización.

24.3. Discuta la valoración de calidad del software según los atributos de calidad mostrados en la figura 24.2. Debe considerar a la vez cada atributo y explicar cómo puede valorarse.

24.4. Diseñe un formato electrónico que pueda usar para registrar comentarios de revisión y para enviar comentarios por correo electrónico a los revisores.

24.5. Describa brevemente posibles estándares que podría utilizar para:

- El uso de sentencias de control en C, C# o Java;
- enviar reportes para un proyecto final en una universidad;
- el proceso de hacer y aprobar cambios al programa (véase el capítulo 26);
- el proceso de comprar e instalar una nueva computadora.

24.6. Suponga que trabaja en una organización que desarrolla productos de bases de datos para individuos y empresas pequeñas. Esta organización está interesada en cuantificar su desarrollo de software. Escriba un reporte que sugiera métricas adecuadas y mencione cómo pueden recopilarse.

24.7. Exprese por qué las inspecciones de programa son una técnica efectiva para descubrir errores en un programa. ¿Qué tipos de errores tienen escasa probabilidad de descubrirse mediante inspecciones?

24.8. Diga por qué las métricas de diseño son, por sí mismas, un método inadecuado para predecir la calidad del diseño.

24.9. Exponga por qué es difícil validar las relaciones entre atributos de producto internos (como la complejidad ciclomática) y los atributos externos (como la mantenibilidad).

24.10. Un colega que es muy buen programador elabora software con un bajo número de defectos, pero siempre pasa por alto los estándares de calidad de la organización. ¿Cómo deberían reaccionar sus administradores ante este comportamiento?

REFERENCIAS

Bamford, R. y Deibler, W. J. (eds.) (2003). "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach". Boca Raton, Fla.: CRC Press.

Barnard, J. y Price, A. (1994). "Managing Code Inspection Information". *IEEE Software*, **11** (2), 59–69.

Basili, V. R. y Rombach, H. D. (1988). "The TAME project: Towards Improvement-Oriented Software Environments". *IEEE Trans. on Software Eng.*, **14** (6), 758–773.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. y Merrit, M. (1978). *Characteristics of Software Quality*. Amsterdam: North-Holland.

Chidamber, S. y Kemerer, C. (1994). "A Metrics Suite for Object-Oriented Design". *IEEE Trans. on Software Eng.*, **20** (6), 476–93.

Crosby, P. (1979). *Quality is Free*. Nueva York: McGraw-Hill.

El-Amam, K. 2001. "Object-oriented Metrics: A Review of Theory and Practice". National Research Council of Canada. http://seg.iit.nrc.ca/English/abstracts/NRC44190.html .

Endres, A. y Rombach, D. (2003). *Empirical Software Engineering: A Handbook of Observations, Laws and Theories*. Harlow, UK: Addison-Wesley.

Fagan, M. E. (1976). "Design and code inspections to reduce errors in program development". *IBM Systems J.*, **15** (3), 182–211.

Fagan, M. E. (1986). "Advances in Software Inspections". *IEEE Trans. on Software Eng.*, **SE-12** (7), 744–51.

Gilb, T. y Graham, D. (1993). *Software Inspection*. Wokingham: Addison-Wesley.

Grady, R. B. (1993). "Practical Results from Measuring Software Quality". *Comm. ACM*, **36** (11), 62–8.

Gunning, R. (1962). *Techniques of Clear Writing*. Nueva York: McGraw-Hill.

Hall, T. y Fenton, N. (1997). "Implementing Effective Software Metrics Programs". *IEEE Software*, **14** (2), 55–64.

Humphrey, W. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.

IEC. 1998. "Standard IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems". International Electrotechnical Commission: Ginebra.

IEEE. (2003). *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, Ca.: IEEE Computer Society Press.

Ince, D. (1994). *ISO 9001 and Software Quality Assurance*. Londres: McGraw-Hill.

Kilpi, T. (2001). "Implementing a Software Metrics Program at Nokia". *IEEE Software*, **18** (6), 72–7.

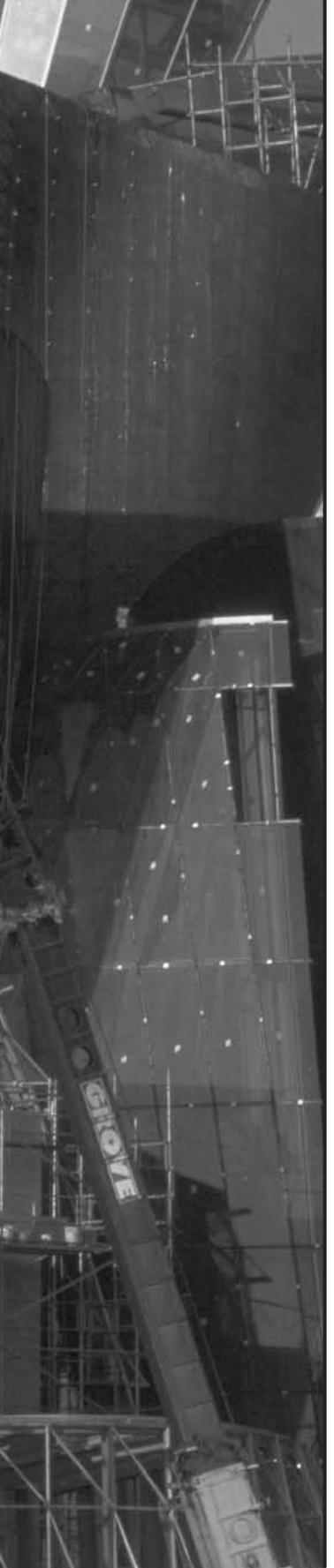
Kitchenham, B. (1990). "Measuring Software Development". En *Software Reliability Handbook*. Rook, P. (ed.). Amsterdam: Elsevier, 303–31.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction, 2nd edition*. Seattle: Microsoft Press.

Mills, H. D., Dyer, M. y Linger, R. (1987). "Cleanroom Software Engineering". *IEEE Software*, **4** (5), 19–25.

Offen, R. J. y Jeffrey, R. (1997). "Establishing Software Measurement Programs". *IEEE Software*, **14** (2), 45–54.

Stalhane, T. y Hanssen, G. K. (2008). "The application of ISO 9001 to agile software development". *9th International Conference on Product Focused Software Process Improvement, PROFES 2008*, Monte Porzio Catone, Italy: Springer.



25

Administración de la configuración

Objetivos

El objetivo de este capítulo es introducirlo a los procesos y las herramientas de administración de la configuración. Al estudiar este capítulo:

- comprenderá los procesos y procedimientos implicados en la gestión de cambio de software;
- conocerá la funcionalidad esencial que debe proporcionar una versión del sistema de gestión y las relaciones entre la gestión de versiones y la construcción de un sistema;
- entenderá las diferencias entre una versión del sistema y una entrega (*release*) de sistema, e identificará las etapas en el proceso de gestión de entregas del software.

Contenido

25.1 Administración del cambio
25.2 Gestión de versiones
25.3 Construcción del sistema
25.4 Gestión de entregas de software (*release*)

Los sistemas de software siempre cambian durante su desarrollo y uso. Se descubren bugs y éstos deben corregirse. Los requerimientos del sistema cambian, y es necesario implementar dichos cambios en una nueva versión del sistema. Se dispone de nuevas versiones de hardware y plataformas de sistema, por lo que hay que adaptar los sistemas para que funcionen con ellos. Los competidores introducen nuevas características en sus sistemas que se deben igualar. Conforme se hacen cambios al software, se crea una nueva versión del sistema. En consecuencia, la mayoría de los sistemas pueden considerarse como un conjunto de versiones, cada una de las cuales debe mantenerse y gestionarse.

La administración de la configuración (CM, por las siglas de *configuration management*) se ocupa de las políticas, los procesos y las herramientas para administrar los sistemas cambiantes de software. Es necesario gestionar los sistemas en evolución porque es fácil perder la pista de cuáles cambios y versiones del componente se incorporaron en cada versión del sistema. Las versiones implementan propuestas para cambios, correcciones de fallas y adaptaciones para diferentes tipos de hardware y sistemas operativos. Pueden existir al mismo tiempo numerosas versiones en uso y bajo desarrollo. Si no se cuenta con procedimientos efectivos de administración de la configuración, se puede malgastar esfuerzo al modificar la versión equivocada de un sistema, entregar a los clientes la versión incorrecta de un sistema u olvidar dónde se almacena el código fuente del software para una versión particular del sistema o componente.

La administración de la configuración es útil para proyectos individuales, ya que es fácil para una persona olvidar qué cambios se realizaron. Es esencial para los proyectos de equipo en los que muchos desarrolladores trabajan al mismo tiempo en un sistema de software. En ocasiones dichos desarrolladores laboran todos en el mismo lugar, pero es cada vez más frecuente que los miembros de los equipos de desarrollo estén distribuidos en diferentes sitios del planeta. El uso de un sistema de administración de la configuración garantiza que los equipos tengan acceso a información sobre un sistema que está bajo desarrollo sin que se interfiera con el trabajo de los demás.

La administración de la configuración de un producto de sistema de software comprende cuatro actividades estrechamente relacionadas (figura 25.1):

1. *Administración del cambio* Esto implica hacer un seguimiento de las peticiones de cambios al software por parte de clientes y desarrolladores, estimar los costos y el efecto de realizar dichos cambios, y decidir si deben implementarse los cambios y cuándo.
2. *Gestión de versiones* Esto incluye hacer un seguimiento de las numerosas versiones de los componentes del sistema y garantizar que los cambios hechos por diferentes desarrolladores a los componentes no interfieran entre sí.
3. *Construcción del sistema* Éste es el proceso de ensamblar los componentes del programa, datos y librerías, y luego compilarlos y vincularlos para crear un sistema ejecutable.
4. *Gestión de entregas* (*release*) Esto implica preparar el software para la entrega externa y hacer un seguimiento de las versiones del sistema que se entregaron para uso del cliente.

La administración de la configuración implica enfrentar un gran volumen de información, por lo que se han desarrollado numerosas herramientas de administración de la

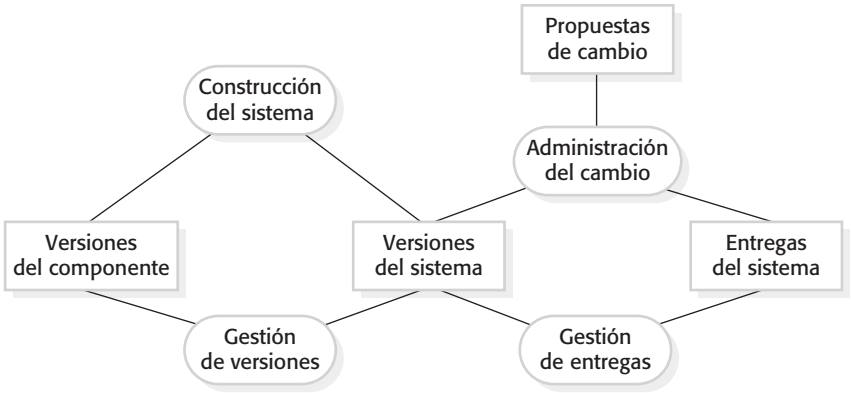


Figura 25.1 Actividades de administración de la configuración

configuración para dar soporte a los procesos de CM. Éstos abarcan desde las simples herramientas que apoyan una sola tarea de administración de la configuración, como el rastreo de bugs, hasta complejos y costosos conjuntos de herramientas integradas que apoyan todas las actividades de administración de la configuración.

Las políticas y los procesos de administración de la configuración definen cómo registrar y procesar los cambios propuestos al sistema, cómo decidir qué componentes del sistema modificar, cómo gestionar las diferentes versiones del sistema y sus componentes, y cómo distribuir estos cambios a los clientes. Las herramientas de administración de la configuración se usan para rastrear las propuestas de cambio, almacenar versiones de componentes del sistema, construir sistemas a partir de dichos componentes, y rastrear liberaciones de las versiones del sistema para los clientes.

En ocasiones la administración de la configuración se considera parte de la gestión de calidad del software (que se trató en el capítulo 24), donde el mismo administrador tiene responsabilidades tanto de gestión de calidad como de administración de la configuración. Cuando se implementa una nueva versión del software, se transfiere del equipo de desarrollo al equipo de aseguramiento de la calidad (QA, por las siglas de *quality assurance*). El equipo QA comprueba que la calidad del sistema sea aceptable. Si lo es, entonces se convierte en un sistema controlado, lo que significa que todos los cambios al sistema tienen que ajustarse y registrarse antes de que éste se implemente.

La definición y el uso de los estándares de administración de la configuración son esenciales para la certificación de la calidad tanto en los estándares ISO 9000 como en CMM y CMMI (Ahern *et al*., 2001; Bamford y Deibler, 2003; Paulk *et al*., 1995; Peach, 1996). Dichos estándares CM pueden basarse en estándares CM genéricos desarrollados por organismos como el IEEE. Por ejemplo, el estándar IEEE 828-1998 es un estándar para planes de administración de la configuración. Estos estándares se enfocan en los procesos CM y en documentos elaborados durante dichos procesos. Al usar los estándares externos como punto de partida, las compañías desarrollan estándares más detallados, definidos de la compañía, que se ajustan a sus necesidades particulares.

Uno de los problemas con la administración de la configuración es que diversas compañías hablan de los mismos conceptos utilizando diferentes términos. Existen razones históricas para esto. Probablemente los sistemas de software militares sean los primeros

Término	Explicación
Ítem de configuración o ítem de configuración de software (SCI, por las siglas de *Software Configuration Item*)	Cualquier aspecto asociado con un proyecto de software (diseño, código, datos de prueba, documento, etcétera) se coloca bajo control de configuración. Por lo general, existen diferentes versiones de un ítem de configuración. Los ítems de configuración tienen un nombre único.
Control de configuración	El proceso de asegurar que las versiones de sistemas y componentes se registren y mantengan de modo tal que los cambios se gestionen, y se identifiquen y almacenen todas las versiones de componentes durante la vida del sistema.
Versión	Una instancia de un ítem de configuración que difiere, en alguna forma, de otras instancias del mismo ítem. Las versiones siempre tienen un identificador único, que se compone generalmente del nombre del ítem de configuración más un número de versión.
Línea base (*baseline*)	Una línea base es una colección de versiones de componente que construyen un sistema. Las líneas base están controladas, lo que significa que las versiones de los componentes que conforman el sistema no pueden ser cambiadas. Por lo tanto, siempre debería ser posible recrear una línea base a partir de los componentes que lo constituyen.
Línea de código (*codeline*)	Una línea de código es un conjunto de versiones de un componente de software y otros ítems de configuración de los cuales depende dicho componente.
Línea principal (*mainline*)	Una secuencia de líneas base que representa diferentes versiones de un sistema.
Entrega, liberación (*release*)	Una entrega de un sistema que se libera para su uso a los clientes (u otros usuarios en una organización).
Espacio de trabajo (*workspace*)	Área de trabajo privada donde puede modificarse el software sin afectar a otros desarrolladores que estén usando o modificando dicho software.
Ramificación (*branching*)	La creación de una nueva línea de código a partir de una versión en una línea de código existente. La nueva línea de código y la existente pueden desarrollarse de manera independiente.
Combinación (*merging*)	La creación de una nueva versión de un componente de software al combinar versiones separadas en diferentes líneas de código. Dichas líneas de código pueden crearse mediante una rama anterior de una de las líneas de código implicadas.
Construcción de sistema	Creación de una versión ejecutable del sistema al compilar y vincular las versiones adecuadas de los componentes y las librerías que constituyen el sistema.

Figura 25.2
Terminología CM

sistemas en los que se usó la administración de la configuración, y la terminología para dichos sistemas reflejaba los procesos y procedimientos ya establecidos para la administración de la configuración del hardware. Los desarrolladores de sistemas comerciales no estaban familiarizados ni con los procedimientos ni con la terminología militar, por lo que inventaban con frecuencia sus propios términos. Los métodos ágiles crearon también nueva terminología, introducida en ocasiones de manera deliberada para distinguir el enfoque ágil de los métodos CM tradicionales. La figura 25.2 define la terminología de administración de la configuración usada en este capítulo.

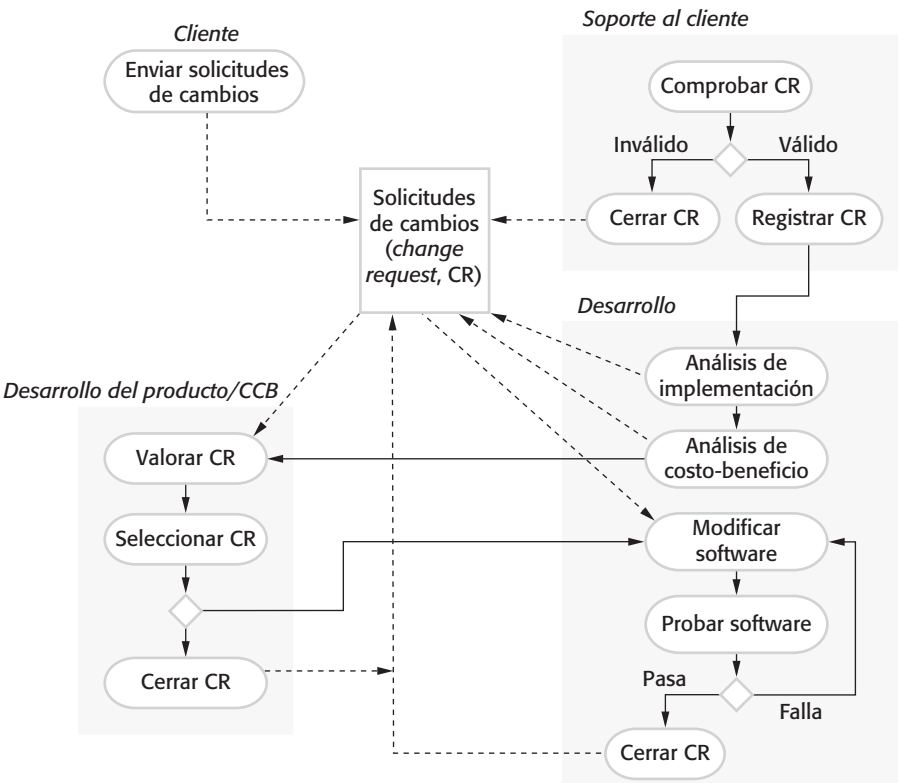


Figura 25.3 Proceso de administración del cambio

25.1 Administración del cambio

El cambio es un hecho en la vida de los grandes sistemas de software. Las necesidades y los requerimientos organizacionales cambian durante la vida de un sistema, los bugs deben repararse y los sistemas adaptarse a cambios en su entorno. Para garantizar que los cambios se apliquen al sistema de forma controlada, se necesita un conjunto de procesos de gestión de cambio soportado por herramientas. La administración del cambio tiene la intención de asegurar que la evolución del sistema sea un proceso gestionado en el que se da prioridad a los cambios más urgentes y rentables.

El proceso de administración del cambio se ocupa de analizar los costos y beneficios de los cambios propuestos, aprobar aquellos que lo ameritan e indagar cuál o cuáles de los componentes del sistema se modificaron. La figura 25.3 es un modelo de un proceso de administración que muestra las principales actividades de administración del cambio. Existen muchas variantes de este proceso en uso pero, para ser efectivos, los procesos de administración del cambio deben tener siempre un medio que compruebe, costee y apruebe los cambios. Este proceso debe entrar en efecto cuando el software se transfiere para la liberación a los clientes o la utilización dentro de una organización.

El proceso de administración del cambio se inicia cuando un "cliente" completa y envía una petición de cambio en que se describe el cambio requerido al sistema. Éste podría ser

Formato de petición de cambio

Proyecto: SICSA/AppProcessing **Número:** 23/02
Solicitante del cambio: I. Sommerville **Fecha:** 20/01/09
Cambio solicitado: El estatus de los solicitantes (rechazado, aceptado, etcétera) debe ser visible en la lista de despliegue de solicitantes.

Analizador del cambio: R. Looek **Fecha de análisis:** 25/01/09
Componentes afectados: ApplicantListDisplay, StatusUpdater

Componentes asociados: StudentDatabase

Valoración del cambio: Relativamente simple de implementar al cambiar el color de despliegue de acuerdo con el estatus. Debe agregarse una tabla para relacionar el estatus con los colores. No se requieren cambios a los componentes asociados.

Prioridad del cambio: Media
Implementación del cambio:
Esfuerzo estimado: 2 horas
Fecha para equipo SGA app.: 28/01/09 **Fecha de decisión CCB:** 30/01/09
Decisión: Aceptar cambio. Implementarse el cambio en la versión 1.2
Implementador del cambio: **Fecha de cambio:**
Fecha de envío a QA: **Decisión de QA:**
Fecha de envío a CM:
Comentarios:

Figura 25.4 Formato de petición de cambio parcialmente completado

el reporte de un bug, en el que se describan sus síntomas, o una petición para agregar alguna funcionalidad al sistema. Algunas compañías tratan por separado los reportes de bug y los nuevos requerimientos, pero, en principio, ambos son simplemente peticiones de cambio. Estas últimas pueden enviarse mediante un formato de petición de cambio (CRF, por las siglas de *change request form*). Aquí se usa el término *cliente* para incluir a cualquier participante que no sea parte del equipo de desarrollo, de modo que los cambios puede sugerirlos, por ejemplo, el departamento de marketing de una compañía.

Los formatos electrónicos de petición de cambios registran información que se comparte entre todos los grupos implicados en la administración del cambio. Conforme se procesa la petición del cambio, se agrega información al CRF para registrar las decisiones tomadas en cada etapa del proceso. Por lo tanto, en cualquier momento representa una fotografía instantánea del estado de petición del cambio. Además de registrar el cambio requerido, el CRF registra las recomendaciones concernientes al cambio, los costos estimados del cambio, y las fechas cuando se solicitó, aprobó, implementó y validó el cambio. El CRF también puede incluir una sección donde un desarrollador enfatice cómo puede implementarse el cambio.

En la figura 25.4 se muestra un ejemplo de formato de petición de cambio parcialmente completado. Éste es un ejemplo de un tipo de CRF que puede usarse en un proyecto grande y complejo de ingeniería en sistemas. Para proyectos más pequeños, se recomienda que las peticiones de cambio se registren de manera formal y el CRF se enfoque en la descripción del cambio requerido, con menos énfasis en los conflictos de implementación. Como desarrollador del sistema, usted decide cómo implementar dicho cambio y estima el tiempo requerido para ello.

Después de enviar una petición de cambio, ésta se verifica para asegurarse de que sea válida. Esta verificación puede venir tanto del cliente como del equipo de soporte de la aplicación, o para peticiones internas de un miembro del equipo de desarrollo. La comprobación es necesaria porque no todas las peticiones de cambio requieren acción. Si la petición de cambio es un reporte de bug, tal vez éste ya haya sido reportado. En ocasiones, lo que la gente considera como problemas en realidad son malas interpretaciones de lo que se espera que haga el sistema. Algunas veces, las personas solicitan características que ya se implementaron, pero que desconocen. Si algo de esto sucede, la petición de cambio se cierra y el formato se actualiza indicando la razón para el cierre. Si es una petición de cambio válida, entonces se registra como una petición sobresaliente para un análisis posterior.

Para peticiones válidas de cambio, la siguiente etapa del proceso consiste en evaluar y costear el cambio. Por lo general, esto es responsabilidad del equipo de desarrollo o del de mantenimiento, pues ellos están en condiciones de determinar lo que se requiere para la implementación del cambio. Debe comprobarse el efecto del cambio sobre el resto del sistema. Para hacer esto, hay que identificar todos los componentes afectados por el cambio. Si realizar el cambio significa que se necesitarán más modificaciones en alguna otra parte del sistema, esto aumentará el costo de implementar el cambio. A continuación, se valoran los cambios requeridos a los módulos del sistema. Finalmente, se estima el costo de efectuar el cambio y se toman en cuenta los costos de modificar los componentes asociados.

Continuando con este análisis, un grupo separado debe determinar si realizar el cambio al software es rentable desde una perspectiva empresarial. Para sistemas militares y gubernamentales este grupo se conoce usualmente como consejo de control del cambio (CCB, por las siglas de *change control board*). En la industria puede llamarse "grupo de desarrollo del producto", el cual es el responsable de tomar las decisiones sobre cómo debe evolucionar el sistema de software. Este grupo debe revisar y aprobar todas las peticiones de cambio, a menos que los cambios impliquen simplemente corregir errores menores en pantallas de despliegue, páginas Web o documentos. Estas peticiones menores deben transmitirse al equipo de desarrollo sin un análisis detallado, pues esto podría ser más costoso que implementar el cambio.

El CCB o el grupo de desarrollo del producto consideran el efecto del cambio desde un punto de vista estratégico y organizacional más que técnico. Decide si el cambio en cuestión está justificado económicamente y prioriza los cambios aceptados para su implementación. Los cambios aceptados se transmiten de regreso al grupo de desarrollo; las peticiones de cambio rechazadas se cierran y ya no se emprenden más acciones. Los factores significativos que deben tomarse en cuenta para decidir si un cambio debe aprobarse o no son los siguientes:

1. *Las consecuencias de no realizar el cambio* Cuando se valora una petición de cambio se debe considerar lo que ocurrirá si éste no se implementa. Si el cambio se relaciona con una falla reportada del sistema, la gravedad de dicha falla tiene que tomarse en cuenta. Si la falla del sistema causa la caída de este último, resulta muy grave, y no hacer el cambio puede perturbar el uso operacional del sistema. Por otra parte, si la falla tiene un efecto menor (por ejemplo, se presentan los colores equivocados en la pantalla), entonces no es importante corregir rápidamente el problema, de modo que el cambio tendrá una prioridad menor.

Clientes y cambios

Los métodos ágiles enfatizan la importancia de que los clientes participen en el proceso de priorización del cambio. El representante del cliente ayuda al equipo a decidir sobre los cambios que deben implementarse en la siguiente iteración de desarrollo. Aunque esto es efectivo para sistemas que están en desarrollo para un solo cliente, puede constituir un problema en el desarrollo de productos donde no hay un cliente real trabajando con el equipo. En esos casos, el equipo tiene que tomar sus propias decisiones respecto a la priorización del cambio.

http://www.SoftwareEngineering-9.com/Web/CM/agilechanges.html

2. *Los beneficios del cambio* ¿El cambio es algo que beneficiará a muchos usuarios del sistema o simplemente es una propuesta que beneficiará sobre todo a quien propone el cambio?
3. *El número de usuarios afectados por el cambio* Si sólo algunos usuarios resultan afectados, entonces al cambio se le puede asignar una baja prioridad. De hecho, hacer el cambio no resulta aconsejable si pudiera tener efectos adversos sobre la mayoría de los usuarios del sistema.
4. *Los costos de hacer el cambio* Si hacer el cambio afecta a muchos componentes del sistema (lo que, por lo tanto, aumenta las posibilidades de introducir nuevos bugs) y/o tarda mucho tiempo en implementarse, entonces se puede rechazar el cambio, debido a los elevados costos implicados.
5. *El ciclo de liberación del producto* Si una nueva versión del software se libera a los clientes, tal vez tenga sentido demorar la implementación del cambio hasta la siguiente liberación planeada (véase la sección 25.3).

La administración del cambio para productos de software (por ejemplo, un producto de sistema CAD), en vez de sistemas que se desarrollan específicamente para cierto cliente, tiene que manejarse en una forma relativamente diferente. En los productos de software, el cliente no participa de manera directa en las decisiones concernientes a la evolución del sistema, de manera que la relevancia del cambio no representa un problema para la compañía del cliente. Las peticiones de cambio para dichos productos provienen del equipo de soporte del cliente, el equipo de marketing de la compañía y los mismos desarrolladores. Dichas peticiones pueden reflejar sugerencias y retroalimentación de los clientes o análisis de lo que ofrecen los productos competitivos.

El equipo de soporte del cliente puede enviar peticiones de cambio asociadas con los bugs que los clientes descubrieron y reportaron después de que se entregó el sistema. Los clientes pueden usar una página Web o el correo electrónico para reportar los bugs. Entonces, un equipo de gestión de bugs comprueba que estos reportes sean válidos y los traduce a peticiones formales de cambio del sistema. El personal de marketing se reúne con los clientes e investiga productos competitivos. Pueden sugerir cambios que deban incluirse para facilitar la venta de una nueva versión de un sistema a clientes nuevos y existentes. Los propios desarrolladores del sistema pueden tener buenas ideas referentes a nuevas características que podrían agregarse al sistema.

El proceso de petición de cambio mostrado en la figura 25.3 se usa después de que un sistema se entregó a los clientes. Durante el desarrollo, cuando se crean nuevas versiones

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Objeto: currentRole
// Autor: R. Looek
// Fecha de creación: 13/11/2009
//
// © St Andrews University 2009
//
// Historial de modificación
// Versión  Modificador  Fecha       Cambio              Razón
// 1.0      J. Jones     11/11/2009  Agregar encabezado  Enviado a CM
// 1.1      R. Looek     13/11/2009  Nuevo campo         Pet. de cambio R07/02
```

Figura 25.5 Historia de derivación

del sistema mediante construcciones diarias (o más frecuentes), por lo general se usa un proceso de administración del cambio más sencillo. Los problemas y cambios aún deben registrarse, pero los cambios que sólo afectan a componentes y módulos individuales no necesitan valorarse de manera independiente; se transmiten directamente al desarrollador del sistema. Éste los acepta u ofrece razones por las que no son necesarios tales cambios. Sin embargo, una autoridad independiente, como el arquitecto del sistema, debe valorar y priorizar los cambios que afectan a aquellos módulos del sistema que produjeron diferentes equipos de desarrollo.

En algunos métodos ágiles, como en la programación extrema, los clientes participan directamente en la decisión de implementar un cambio. Cuando proponen un cambio a los requerimientos del sistema, trabajan con el equipo para valorar el efecto de dicho cambio y deciden entonces si éste tendría prioridad sobre las características planeadas para el siguiente incremento del sistema. No obstante, los cambios que implican mejoramiento del software se dejan a discreción de los programadores que trabajan en el sistema. La refactorización, en la que el software se mejora de manera continua, no se ve como una carga, sino como parte necesaria del proceso de desarrollo.

Conforme el equipo de desarrollo modifica los componentes de software, debe mantener un registro de los cambios hechos a cada componente. Algunas veces a esto se le conoce como historial de derivación de un componente. Una buena forma de conservar el historial de derivación es en un comentario estandarizado al principio del código fuente del componente (figura 25.5). Este comentario debe hacer referencia a la petición de cambio que provocó el cambio en el software. Entonces uno puede escribir rutinas sencillas que busquen todos los componentes y procesen los historiales de derivación para generar reportes de cambio de componentes. En el caso de documentos, los registros de los cambios incorporados en cada versión se anotan por lo general al frente del documento en una página aparte. Esto se discute en el capítulo Web sobre documentación.

La administración del cambio, por lo general, recibe soporte de herramientas especializadas de software. Éstas pueden ser herramientas relativamente sencillas basadas en la Web, como Bugzilla, que se usa para reportar problemas con muchos sistemas de código abierto. O bien, pueden usarse herramientas más complejas para automatizar todo el proceso de manejar las peticiones de cambio desde la propuesta inicial del cliente hasta la aprobación del cambio.

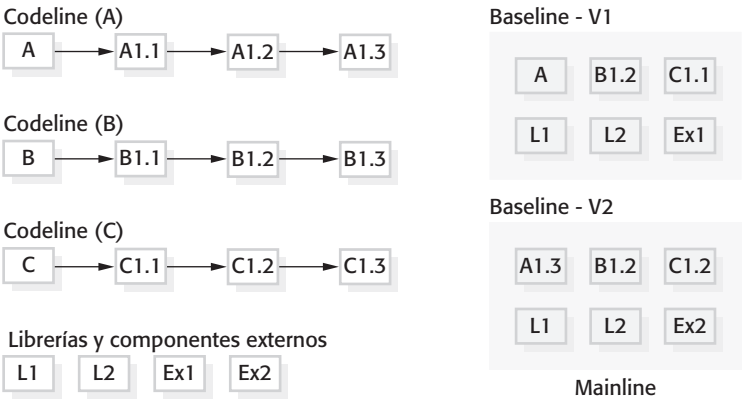


Figura 25.6 Líneas de código (*codelines*) y líneas base (*baselines*)

25.2 Gestión de versiones

La gestión de versiones (VM, por las siglas de *version management*) es el proceso de hacer un seguimiento de las diferentes versiones de los componentes de software o ítems de configuración, y los sistemas donde se usan dichos componentes. También incluye asegurar que los cambios hechos a dichas versiones por los diferentes desarrolladores no interfieran unos con otros. Por lo tanto, se puede considerar a la gestión de versiones como el proceso de administrar líneas de código y líneas base.

La figura 25.6 ilustra las diferencias entre línea de código y línea base. En esencia, una línea de código es una secuencia de versiones de código fuente con las versiones más recientes en la secuencia derivadas de las versiones anteriores. Las líneas de código se aplican regularmente a componentes de sistemas, de manera que existen diferentes versiones de cada componente. Una línea base es una definición de un sistema específico. Por consiguiente, la línea base especifica las versiones del componente que se incluyen en el sistema más una especificación de las librerías usadas, archivos de configuración, etcétera. En la figura 25.6 se observa que diferentes líneas base usan distintas versiones de los componentes de cada línea de código. En el diagrama se sombrearon los recuadros que representan componentes en la definición línea base para indicar que en realidad son referencias a componentes en una línea de código. La línea principal es una secuencia de versiones del sistema desarrolladas a partir de una línea base original.

Las líneas base pueden especificarse mediante un lenguaje de configuración, lo que permite definir cuáles componentes se incluyen en una versión de un sistema particular. Es posible especificar de manera explícita una versión de componente específica (X.1.2, por ejemplo) o simplemente especificar el identificador del componente (X). Si usa el identificador, esto significa que en la línea base debe usarse la versión más reciente del componente.

Las líneas base son importantes porque muchas veces es necesario volver a crear una versión específica de un sistema completo. Por ejemplo, una línea de producto puede ejemplificarse de modo que existan versiones de sistema individuales para diferentes clientes. Posiblemente se tenga que volver a crear la versión entregada a un cliente específico si, por ejemplo, dicho cliente reporta bugs en su sistema que deban repararse.

Para soportar la gestión de versiones, siempre se deben usar herramientas de gestión de versiones (llamadas en ocasiones sistemas de control de versiones o sistemas de control

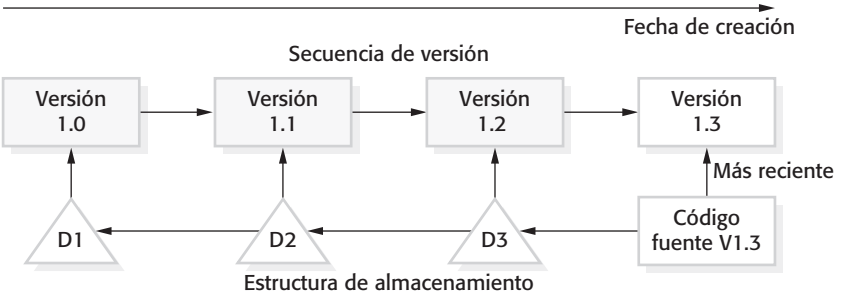


Figura 25.7 Gestión de almacenamiento con deltas

de código fuente). Estas herramientas identifican, almacenan y controlan el acceso a las diferentes versiones de los componentes. Se hallan disponibles muchos sistemas diferentes de gestión de versiones, incluidos los sistemas de código abierto ampliamente usados como CVS y Subversion (Pilato *et al*., 2004; Vesperman, 2003).

Los sistemas de gestión de versiones ofrecen a menudo varias características:

1. *Identificación de versión y entrega (release)* A las versiones gestionadas se les asignan identificadores cuando se envían al sistema. Dichos identificadores se basan, por lo general, en el nombre del ítem de configuración (por ejemplo, ButtonManager), seguido por uno o más números. De esta manera, ButtonManager 1.3 significa la tercera versión en codeline 1 del componente ButtonManager. Algunos sistemas CM también permiten la asociación de atributos con versiones (por ejemplo, móvil, pantalla pequeña), que también pueden usarse para identificación de la versión. Es importante que el sistema de identificación sea consistente, ya que esto simplifica el problema de definir configuraciones. Hace más sencillo el uso de referencias abreviadas (por ejemplo, *.V2, que significa la versión 2 de todos los componentes).

2. *Gestión de almacenamiento* Para reducir el espacio de almacenamiento requerido por múltiples versiones de los componentes que difieren sólo ligeramente, los sistemas de gestión de versiones ofrecen, por lo general, facilidades de gestión de almacenamiento. En vez de conservar una copia completa de cada versión, el sistema almacena una lista de diferencias (deltas) entre una versión y otra. Al aplicar esto a una versión fuente (por lo regular, la versión más reciente), puede recrearse una versión objetivo. Esto se ilustra en la figura 25.7.

3. *Registro del historial de cambios* Todos los cambios realizados al código de un sistema o componente se registran y enumeran. En algunos sistemas, dichos cambios pueden usarse para seleccionar la versión de un sistema en particular. Esto implica etiquetar componentes con palabras clave que describan los cambios realizados. Entonces se pueden usar dichas etiquetas (*tags*) para seleccionar los componentes a incluir en una línea base.

4. *Desarrollo independiente* Es posible que diferentes desarrolladores trabajen en el mismo componente al mismo tiempo. El sistema de gestión de versiones hace un seguimiento de los componentes que se marcaron para la edición y se asegura de que no interfieran los cambios hechos a un componente por diferentes desarrolladores.

5. *Soporte de proyecto* Un sistema de gestión de versiones puede soportar el desarrollo de varios proyectos que comparten componentes. En los sistemas de soporte de

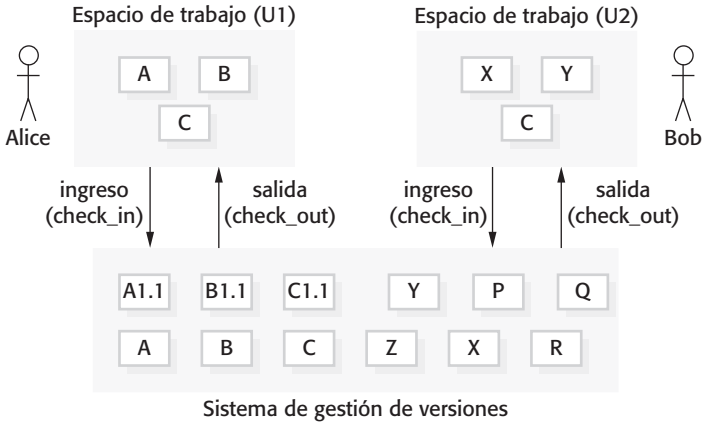


Figura 25.8 Ingreso y salida de un repositorio de versión

proyecto, como CVS (Vesperman, 2003), es posible ingresar (*chek in*) y sacar (*check out*) todos los archivos asociados con un proyecto en lugar de tener que trabajar a la vez con un archivo o directorio.

Cuando se desarrollaron por primera vez los sistemas de gestión de versiones, la gestión del almacenamiento fue una de sus funciones más importantes. Las características de gestión de almacenamiento en un sistema de control de versiones reduce el espacio de disco requerido para mantener todas las versiones del sistema. Cuando se crea una nueva versión, el sistema simplemente almacena una delta (una lista de diferencias) entre la nueva versión y la anterior que se usó para crear esa nueva versión (lo que se ilustra en la parte inferior de la figura 25.7). En esta misma figura, los recuadros sombreados representan versiones anteriores de un componente que se recrean automáticamente a partir de la versión de componente más reciente. Por lo general, las deltas se almacenan como listas de líneas que cambiaron y, al aplicarlas automáticamente, puede crearse una versión de un componente a partir de otro. Como es más probable que se use la versión más reciente de un componente, la mayoría de los sistemas almacenan completa dicha versión. Entonces, las deltas definen cómo recrear versiones anteriores del sistema.

La mayor parte del desarrollo de software es una actividad grupal, de modo que con frecuencia surgen situaciones en las que diferentes miembros del equipo trabajan paralelamente en el mismo componente. Por ejemplo, suponga que Alicia hace algunos cambios a un sistema, lo que implica cambiar los componentes A, B y C. Al mismo tiempo, Roberto trabaja en cambios que requieren modificar los componentes X, Y y C. Entonces, tanto Alicia como Roberto cambian C. Es importante evitar que estos cambios interfieran entre sí, es decir, que los cambios de Roberto a C sobrescriban en los de Alicia o viceversa.

Para apoyar el desarrollo independiente sin interferencia, los sistemas de gestión de versiones usan el concepto de repositorio público y un espacio de trabajo privado. Los desarrolladores sacan componentes del repositorio público hacia su espacio de trabajo privado y pueden cambiarlos como deseen en su mismo espacio. Cuando sus cambios están completos, ingresan los componentes al repositorio. Esto se ilustra en la figura 25.8. Si dos o más personas trabajan en un componente al mismo tiempo, cada uno debe sacar el componente del repositorio. Si se extrae un componente, el sistema de gestión de versiones por lo general advierte a otros usuarios que quieren sacar dicho componente que alguien más lo está usando. El sistema también garantizará que, al ingresar los

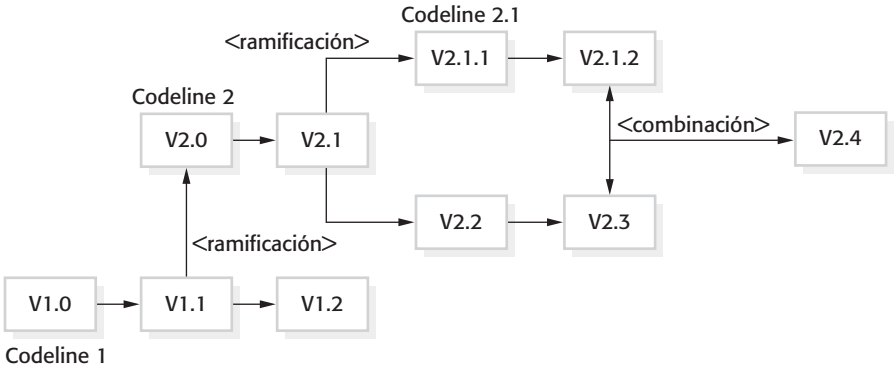


Figura 25.9
Ramificación (*branching*) y combinación (*merging*)

componentes modificados a las distintas versiones, se les asignen diferentes identificadores de versión y se almacenen por separado.

Una consecuencia del desarrollo independiente del mismo componente es que las líneas de código pueden ramificarse (*branch*). En vez de una secuencia lineal de versiones que refleje los cambios al componente con el paso del tiempo, puede haber varias secuencias independientes, como se muestra en la figura 25.9. Esto es normal en el desarrollo de sistemas, en el que los diferentes desarrolladores trabajan de manera independiente en distintas versiones del código fuente y lo cambian en diversas formas.

En alguna etapa, tal vez sea necesario combinar ramificaciones de líneas de código para crear una nueva versión de un componente que incluya todos los cambios realizados. Esto también se muestra en la figura 25.9, donde las versiones 2.1.2 y 2.3 del componente se combinan para crear la versión 2.4. Si los cambios realizados involucran partes completamente diferentes del código, las versiones del componente pueden combinarse automáticamente mediante el sistema de gestión de versiones, al combinar las deltas que se aplican al código. Con más frecuencia, existen traslapes entre los cambios realizados que además interfieren entre sí. Un desarrollador debe verificar los conflictos y modificar los cambios de manera que sean compatibles.

25.3 Construcción del sistema

La construcción del sistema es el proceso de crear un sistema ejecutable y completo al compilar y vincular los componentes del sistema, librerías externas, archivos de configuración, etcétera. Las herramientas de construcción del sistema y las de gestión de versiones deben comunicarse, pues el proceso de construcción implica extraer versiones del componente del repositorio administrado por el sistema de gestión de versiones. La descripción de configuración que se usa para identificar una línea base utiliza también la herramienta de construcción del sistema.

Construir es un proceso complejo, que potencialmente es proclive al error, pues tres diferentes plataformas de sistema pueden estar implicadas (figura 25.10):

1. El sistema de desarrollo, que incluye herramientas de desarrollo, como los compiladores, editores de código fuente, etcétera. Los desarrolladores sacan código del sistema de gestión de versiones hacia un espacio de trabajo privado antes de hacer

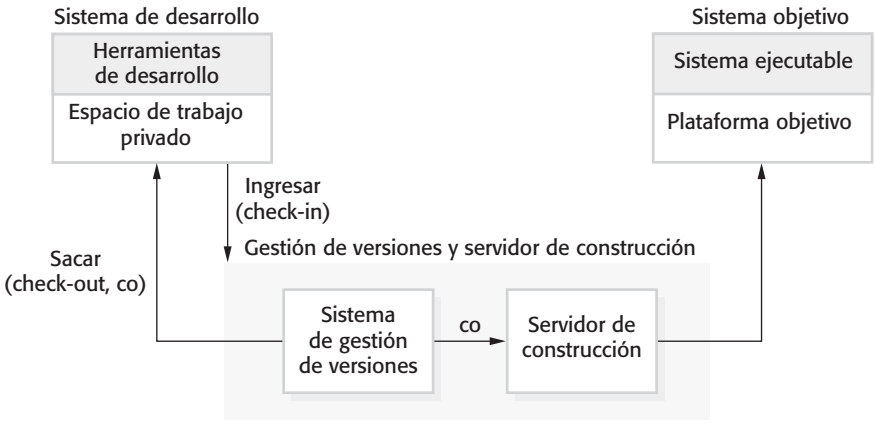


Figura 25.10 Plataforma de desarrollo, plataforma de construcción y plataforma de objetivo

cambios al sistema. Ellos tal vez quieran construir una versión del sistema para probarla en su entorno de desarrollo antes de aplicar los cambios que hicieron al sistema de gestión de versiones. Esto supone usar herramientas de construcción locales que usan versiones de componentes sacadas en el espacio de trabajo privado.

2. El servidor de construcción, que se usa para construir versiones ejecutables definitivas del sistema. Éste interactúa estrechamente con el sistema de gestión de versiones. Los desarrolladores ingresan código a este sistema antes de que se construya. La elaboración del sistema puede depender de librerías externas que no se incluyen en el sistema de gestión de versiones.

3. El entorno objetivo es la plataforma donde se ejecuta el sistema. Éste puede ser el mismo tipo de computadora que se usó para los sistemas de desarrollo y construcción. Sin embargo, para sistemas embebidos y de tiempo real, el entorno objetivo con frecuencia es más pequeño y sencillo que el entorno de desarrollo (por ejemplo, un teléfono celular). Para sistemas grandes, el entorno objetivo puede incluir bases de datos y otros sistemas COTS que no pueden instalarse en máquinas de desarrollo. En ambos casos, no es posible construir y probar el sistema en la computadora de desarrollo o en el servidor de construcción.

El sistema de desarrollo y el servidor de construcción pueden interactuar con el sistema de gestión de versiones. El sistema VM puede alojarse en el servidor de construcción o en un servidor dedicado. Para sistemas embebidos puede instalarse un entorno de simulación en el entorno de desarrollo para pruebas, en vez de usar la plataforma de sistema embebido real. Dichos simuladores pueden ofrecer mejor soporte de depuración que el disponible en un sistema embebido. Sin embargo, es muy difícil simular el comportamiento de un sistema embebido en todos los aspectos. Por lo tanto, las pruebas del sistema se deben realizar en la plataforma real donde se ejecutará el sistema, así como en el simulador del sistema.

La construcción del sistema implica ensamblar una gran cantidad de información acerca del software y su entorno operacional. Por lo tanto, para cualquier sistema aparte de los pequeños, siempre tiene sentido usar una herramienta de construcción automatizada para crear una construcción del sistema (figura 25.11). Observe que no sólo necesita los archivos del código fuente implicados en la construcción, sino tal vez se deban

Figura 25.11 Construcción del sistema

vincular dichos archivos con librerías, archivos de datos (como un archivo de mensajes de error) y archivos de configuración proporcionados externamente que definan la instalación objetivo. Es probable que se deban especificar las versiones del compilador y otras herramientas de software usadas en la construcción. De manera ideal, usted debería ser capaz de construir un sistema completo con un solo comando o clic del ratón.

Existe una gran cantidad de herramientas de construcción disponibles, y un sistema de construcción puede ofrecer algunas de las siguientes características o todas ellas:

1. *Generación de rutinas (scripts) de construcción* Si es necesario, el sistema de construcción debe analizar el programa en construcción, identificar los componentes dependientes y generar automáticamente una rutina de construcción (llamado algunas veces archivo de configuración). El sistema debe soportar también la creación manual y la edición de rutinas de construcción.
2. *Integración del sistema de gestión de versiones* El sistema de construcción debe sacar las versiones de componentes requeridas del sistema de gestión de versiones.
3. *Recompilación mínima* El sistema de construcción debe establecer qué código fuente necesita volver a compilarse y establecer las compilaciones si así se requiere.
4. *Creación de sistema ejecutable* El sistema de construcción debe vincular los archivos de código de objeto compilado entre sí y con otros archivos requeridos, como las librerías y los archivos de configuración, para crear un sistema ejecutable.
5. *Automatización de pruebas* Algunos sistemas de construcción pueden efectuar pruebas automatizadas utilizando herramientas de automatización de pruebas como JUnit. Éstas comprueban que la construcción no se haya "roto" por los cambios.
6. *Informes* El sistema de construcción debe ofrecer informes sobre el éxito o fracaso de la construcción y las pruebas que se efectuaron.
7. *Generación de documentación* El sistema de construcción puede generar notas referentes a las páginas de ayuda de la construcción y del sistema.

La rutina de construcción es una definición del sistema a construir. Incluye información respecto a los componentes y sus dependencias, así como sobre las versiones de las herramientas utilizadas para compilar y vincular el sistema. La rutina de construcción incluye la especificación de la configuración, de manera que el lenguaje de escritura de rutinas utilizado generalmente es el mismo que el lenguaje de descripción de la configu-

ración. El lenguaje de configuración incluye sentencias para describir los componentes del sistema a incluir en la construcción y sus dependencias.

Puesto que la compilación es un proceso de cómputo intensivo, las herramientas para soportar la construcción de sistemas se diseñan por lo general para minimizar la cantidad de compilación que se requiere. Esto se hace comprobando si está disponible una versión compilada de un componente. De ser así, no hay necesidad de volver a compilar dicho componente. Por lo tanto, debe haber una forma de vincular sin ambigüedades el código fuente de un componente con su código objeto equivalente.

La forma en que se hace esto es asociar una firma única con cada archivo donde se almacene un componente del código fuente. El código objeto correspondiente, que se compiló a partir del código fuente, tiene una firma relacionada que identifica cada versión del código fuente y cambia cuando éste se edita. Al comparar las firmas en los archivos de código fuente y objeto, es posible decidir si el componente del código fuente se usó para generar el componente de código objeto.

Hay dos tipos de firmas que pueden usarse:

1. *Modificación de marca de tiempo (timestamp)* La firma en el archivo del código fuente es la fecha y hora de cuándo éste se modificó. Si el archivo del código fuente de un componente se modifica después del archivo del código objeto relacionado, entonces el sistema supone que se requiere "recompilación" para crear un nuevo archivo del código objeto.

   Por ejemplo, suponga que los componentes Comp.java y Comp.class tienen firmas de modificación de 17:03:05:02:14:2009 y 16:34:25:02:12:2009, respectivamente. Esto significa que el código Java se modificó a las 17 horas con 3 minutos y 5 segundos del 14 de febrero de 2009, y la versión compilada se modificó a las 16 horas con 34 minutos y 25 segundos del 12 de febrero de 2009. En este caso, el sistema recompilaría automáticamente Comp.java porque la versión compilada no incluye los cambios hechos al código fuente desde el 12 de febrero.

2. *Sumas de verificación (checksums) de código fuente* La firma en el archivo del código fuente es una suma de verificación calculada a partir de datos en el archivo. Una función checksum calcula un número único usando el texto fuente como entrada. Si se modifica el código fuente (incluso por un carácter), esto generará una suma diferente. Por lo tanto, usted puede estar seguro de que los archivos de código fuente con diferentes sumas de verificación en realidad son diferentes. La suma de verificación se asigna al código fuente justo antes de la compilación e identifica de manera exclusiva el archivo fuente. Entonces el sistema de construcción etiqueta el archivo de código objeto generado con la firma checksum. Si no hay archivo de código objeto con la misma firma que el archivo de código fuente a incluir en un sistema, entonces es necesario recompilar el código fuente.

Como a menudo los archivos del código objeto no están en versiones, el primer enfoque significa que sólo el archivo del código objeto compilado más recientemente se mantiene en el sistema. Esto, por lo general, se relaciona con el archivo del código fuente por nombre (es decir, tiene el mismo nombre que el archivo de código fuente, pero con un sufijo diferente). Por lo tanto, el archivo fuente Comp.java puede generar el archivo objeto Comp.class. Puesto que los archivos fuente y objeto están vinculados por nombre y no por una firma de archivo fuente explícita, por lo general no es posible construir diferentes versiones de un componente de código fuente en el mismo directorio al mismo tiempo, pues ello generaría archivos objeto con el mismo nombre.

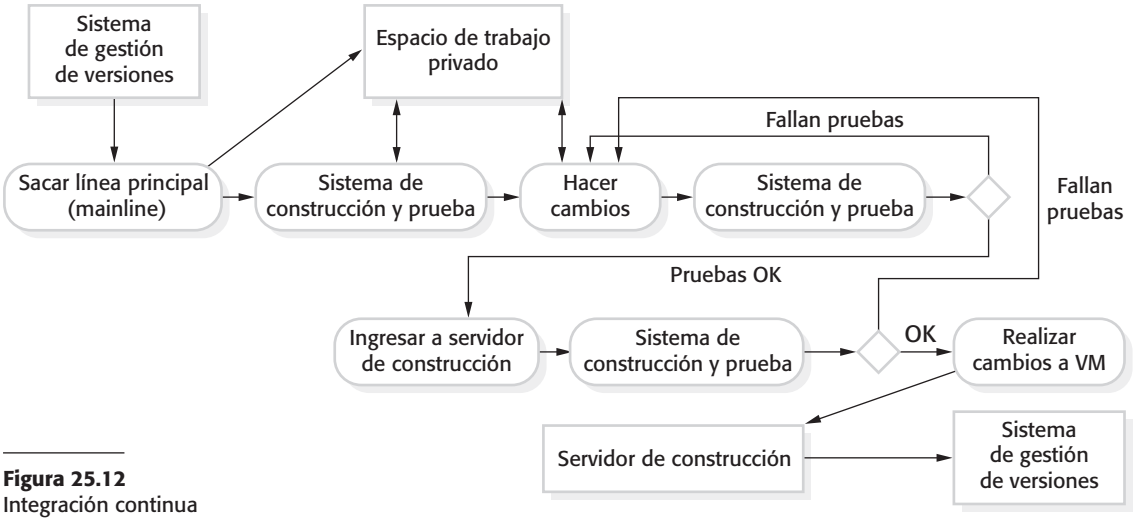


Figura 25.12
Integración continua

El enfoque checksum tiene la ventaja de permitir muchas versiones diferentes del código objeto de un componente para mantenerse al mismo tiempo. La firma, más que el nombre de archivo, es el vínculo entre el código fuente y objeto. Los archivos de código fuente y de código objeto tienen la misma firma. Por lo tanto, cuando se recompila un componente, no sobrescribe el código objeto, como sería normalmente el caso cuando se usa timestamp. En vez de ello, genera un nuevo archivo de código objeto y lo etiqueta con la firma del código fuente. Es posible la compilación paralela y compilar diferentes versiones de un componente al mismo tiempo.

Los métodos ágiles recomiendan que los componentes de sistema muy frecuentes deben realizarse con pruebas automatizadas (llamadas en ocasiones pruebas de humo) para descubrir problemas del software. Los componentes frecuentes pueden ser parte de un proceso de integración continua, como se muestra en la figura 25.12. Para ser congruentes con la noción de los métodos ágiles de elaborar muchos cambios pequeños, la integración continua implica reconstruir frecuentemente la línea principal (*mainline*), después de realizar pequeños cambios al código fuente. Los pasos de la integración continua son:

1. Saque la línea principal del sistema de gestión de versiones hacia el espacio de trabajo privado del desarrollador.
2. Construya el sistema y efectúe pruebas automatizadas para garantizar que el sistema construido pasa todas las pruebas. Si no lo hace, la construcción se descompone y hay que informar a quienquiera que ingrese al último sistema línea base (*baseline*). Ellos son responsables de reparar el problema.
3. Realice los cambios a los componentes del sistema.
4. Construya el sistema en el espacio de trabajo privado y vuelva a efectuar las pruebas del sistema. Si las pruebas fallan, continúe la edición.
5. Una vez que el sistema pasa sus pruebas, ingréselo en el sistema de construcción, pero no confirme como una línea base nueva del sistema.

6. Construya el sistema en el servidor de construcción y efectúe las pruebas. Necesita hacer esto en caso de que otros hayan modificado componentes luego de que usted los sacó del sistema. Si éste es el caso, saque el componente que falló y edítelo de modo que las pruebas pasen en su espacio de trabajo privado.
7. Si el sistema pasa sus pruebas en el sistema de construcción, confirme entonces los cambios que hizo como una nueva línea base en la línea principal del sistema.

El argumento para la integración continua es que permite que los problemas causados por las interacciones entre diferentes desarrolladores se descubran y reparen tan pronto como sea posible. El sistema más reciente en la línea principal es el sistema funcional definitivo. Sin embargo, aunque la integración continua es una buena idea, no siempre es posible implementar este enfoque a la construcción del sistema. Las razones para esto son:

1. Si el sistema es muy grande, puede tardar mucho tiempo construir y probar. Por lo tanto, no es práctico construir muchas veces al día dicho sistema.
2. Si la plataforma de desarrollo es diferente de la plataforma objetivo, tal vez no sea posible efectuar pruebas del sistema en el espacio de trabajo privado. Puede haber diferencias en el hardware, el sistema operativo o el software instalado. Por consiguiente, se requiere más tiempo para probar el sistema.

Para sistemas grandes o sistemas donde la plataforma de ejecución no es la misma que la plataforma de desarrollo, la integración continua quizá no sea práctica. Ante tales circunstancias se puede usar un sistema que se construya diariamente. Las características de esto son las siguientes:

1. La organización de desarrollo establece un tiempo de entrega (por ejemplo, 2 P.M.) para los componentes del sistema. Si los desarrolladores tienen nuevas versiones de los componentes que escriben, deben entregarlas en ese plazo. Los componentes pueden estar incompletos, pero deben ofrecer alguna funcionalidad básica que pueda ponerse a prueba.
2. A partir de dichos componentes se crea una nueva versión del sistema al compilarlos y vincularlos para formar un sistema completo.
3. Entonces este sistema se entrega al equipo de pruebas, que realiza un conjunto de pruebas de sistema predefinidas. Al mismo tiempo, los desarrolladores todavía trabajan en sus componentes, añaden funcionalidad y reparan las fallas descubiertas en pruebas anteriores.
4. Las fallas que se descubren durante las pruebas del sistema se documentan y regresan a los desarrolladores del sistema, quienes reparan dichas fallas en una versión posterior del componente.

Las ventajas de usar componentes frecuentes de software son que aumentan las posibilidades de descubrir de manera oportuna durante el proceso los problemas que surgen a partir de las interacciones de los componentes. La construcción frecuente alienta las pruebas de unidad profundas en los componentes. Psicológicamente, los desarrolladores se ponen bajo presión para "no romper la construcción"; esto es, tratan de evitar el ingreso de versiones de

componentes que causen que todo el sistema falle. Por lo tanto, tienen reticencia a entregar nuevas versiones de componentes que no se hayan probado de manera adecuada. En consecuencia, emplean menos tiempo durante las pruebas del sistema descubriendo y lidiando con las fallas de software que pudiera encontrar el desarrollador.

25.4 Gestión de entregas de software (*release*)

Una entrega (*release*) de sistema es una versión de un sistema de software que se distribuye a los clientes. Para software de mercado masivo es posible identificar por lo general dos tipos de entregas: release mayor, que proporciona funcionalidad significativamente nueva, y release menor, que repara bugs y corrige problemas reportados por el cliente. Por ejemplo, este libro se escribió en una computadora Apple Mac donde el sistema operativo es OS 10.5.8. Esto significa la release menor 8 de la release mayor 5 de OS 10. Las entregas mayores son muy importantes económicamente para el proveedor de software, pues los clientes tienen que pagar por ellas. Las entregas menores generalmente se distribuyen de manera gratuita.

Para software a la medida o líneas de producto de software, la gestión de las entregas del sistema es un proceso complejo. Es posible que deban producirse entregas especiales del sistema para cada cliente, y clientes individuales pueden ejecutar muchas entregas diferentes del sistema al mismo tiempo. Lo anterior significa que una compañía de software que vende un producto de software especializado tal vez deba gestionar decenas o incluso cientos de diferentes entregas de dicho producto. Sus sistemas y procesos de administración de la configuración deben diseñarse para dar información sobre qué clientes tienen cuáles releases del sistema y sobre la relación entre entregas y versiones del sistema. En caso de problemas, quizá sea necesario reproducir con exactitud el software que se entregó a un cliente particular.

Por lo tanto, cuando se produce una entrega de sistema, esto debe documentarse para garantizar que pueda recrearse con exactitud en el futuro. Esto es particularmente importante para sistemas embebidos personalizados de larga duración, como los que controlan máquinas complejas. Los clientes pueden usar una sola entrega de dichos sistemas durante muchos años y requerir cambios específicos a un sistema de software particular mucho tiempo después de su fecha de liberación original.

Para documentar una entrega, es necesario registrar las versiones específicas de los componentes de código fuente que se usaron en la creación del código ejecutable. Hay que conservar copias de los archivos de código fuente, los ejecutables correspondientes y todos los datos y archivos de configuración. También hay que registrar las versiones del sistema operativo, librerías, compiladores y otras herramientas utilizadas para construir el software. Esto puede requerir construir exactamente el mismo sistema en alguna fecha posterior. También puede significar que debe almacenar copias del software de plataforma y las herramientas usadas para crear el sistema en el sistema de gestión de versiones junto con el código fuente del sistema objetivo.

Preparar y distribuir una entrega de sistema es un proceso costoso, en particular para los productos de software de mercado masivo. Además del trabajo técnico que implica la creación de una distribución de entrega (release), hay que elaborar publicidad y material de difusión, así como establecer estrategias de marketing para convencer a los clientes de

Factor	Descripción
Calidad técnica del sistema	Si se reportan graves fallas del sistema que afectan la forma en que muchos clientes lo usan, puede ser necesario emitir una entrega de reparación de falla. Las fallas menores de sistema pueden remediarse con el uso de parches (distribuidos por lo general en Internet) que se aplican a la entrega actual del sistema.
Cambios de plataforma	Tal vez se deba crear una nueva entrega de una aplicación de software cuando se libera una nueva versión de la plataforma de sistema operativo.
Quinta ley de Lehman (véase el capítulo 9)	Esta "ley" sugiere que si se agrega mucha funcionalidad nueva a un sistema, también se introducirán bugs que limitarán la cantidad de funcionalidad que se puede incluir en la siguiente entrega. Por lo tanto, es posible que una entrega de sistema con funcionalidad significativamente nueva vaya seguida por una entrega que se enfoque en reparar problemas y mejorar el rendimiento.
Competencia	Para software de mercado masivo, quizá sea necesaria una entrega de sistema, debido a que productos competitivos introdujeron nuevas características y podría perderse la participación en el mercado si éstas no se ofrecen a los clientes existentes.
Requerimientos de marketing	El departamento de marketing de una organización tal vez hizo compromisos para que las entregas estén disponibles en una fecha particular.
Propuestas de cambio del cliente	Para sistemas a la medida, los clientes posiblemente hicieron y pagaron por un conjunto específico de propuestas de cambios del sistema, y esperan una entrega del sistema tan pronto como se hayan implementado.

Figura 25.13 Factores que influyen la planeación de release de sistema

comprar la nueva entrega del sistema. Debe considerarse con cuidado el tiempo de las entregas. Si las entregas son muy frecuentes o requieren actualizaciones de hardware, los clientes quizá no se cambien hacia la nueva entrega, en especial si tienen que pagar por ella. Si las entregas del sistema son muy poco frecuentes, tal vez se pierda la participación en el mercado, pues los clientes se cambiarán hacia sistemas alternativos.

En la figura 25.13 se muestran los diversos factores técnicos y organizacionales que hay que tomar en cuenta al decidir sobre cuándo liberar una nueva versión de un sistema.

Una entrega de sistema no sólo es el código ejecutable del sistema; ésta también puede incluir:

- Archivos de configuración que definan cómo debe configurarse la entrega (release) para instalaciones particulares;
- archivos de datos, como los archivos de mensajes de error, necesarios para la operación exitosa del sistema;
- un programa de instalación para ayudar a instalar el sistema en el hardware objetivo;
- documentación electrónica y escrita que describa al sistema;
- empaquetado y publicidad asociada diseñados para dicha entrega.

La creación de entrega es el proceso de instaurar la colección de archivos y documentación que incluyen todos los componentes de la entrega del sistema. El código

ejecutable del programa y todos los archivos de datos asociados deben identificarse en el sistema de gestión de versiones y etiquetarse con el identificador de la entrega (release). Quizás haya que escribir descripciones de configuración para hardware y sistemas operativos diferentes, e instrucciones preparadas para los clientes que necesiten configurar sus sistemas. Si se distribuyen en la máquina manuales legibles, deben almacenarse copias electrónicas con el software. Probablemente haya que escribir rutinas para el programa de instalación. Finalmente, cuando toda la información está disponible, debe prepararse una imagen maestra ejecutable del software y transferirse para distribución a los clientes o almacenes de ventas.

Cuando se planea la instalación de nuevas entregas del sistema, no se puede suponer que los clientes siempre instalarán nuevas entregas del sistema. Algunos usuarios del sistema pueden estar satisfechos con un sistema existente y considerarán que no vale la pena el costo de cambiar a una nueva entrega. Por lo tanto, las nuevas entregas del sistema no pueden depender de la instalación de entregas anteriores. Para ilustrar este problema, considere el siguiente escenario:

1. La entrega 1 de un sistema se distribuye y se pone en uso.
2. La entrega 2 requiere la instalación de nuevos archivos de datos, pero algunos clientes no necesitan las facilidades de la entrega 2, así que continúan con la entrega 1.
3. La entrega 3 requiere de los archivos de datos instalados en la entrega 2 y no tiene nuevos archivos de datos propios.

El distribuidor del software no puede suponer que los archivos requeridos por la entrega 3 ya se instalaron en todos los sitios. Algunos sitios pueden ir directamente de la entrega 1 a la entrega 3, y saltar la entrega 2. Otros sitios posiblemente modificaron los archivos de datos asociados con la entrega 2 para reflejar circunstancias locales. En consecuencia, los archivos de datos deben distribuirse e instalarse con la entrega 3 del sistema.

Los costos de marketing y empaquetado asociados con las nuevas entregas de productos de software son altos, de modo que los proveedores de productos, por lo general, sólo crean nuevas entregas para nuevas plataformas o para agregar funcionalidad significativamente novedosa. Entonces cobran a sus usuarios por este nuevo software. Cuando se descubren problemas en la entrega existente, los proveedores de software elaboran parches para reparar este software, disponibles en un sitio Web para que los clientes los descarguen.

El problema con usar parches descargables es que muchos clientes nunca pueden descubrir la existencia de dichas reparaciones de problemas ni entender por qué deben instalarlos. En vez de ello, siguen usando el sistema defectuoso existente, con los consecuentes riesgos para su empresa. En algunas situaciones, donde el parche está diseñado para reparar vacíos de seguridad, los riesgos de fallar en la instalación del parche pueden significar que la empresa es vulnerable a ataques externos. Para evitar estos problemas, los vendedores de software para mercado masivo, como Adobe, Apple y Microsoft, por lo general implementan actualizaciones automáticas, en las que los sistemas se actualizan siempre que está disponible una nueva versión mínima. Sin embargo, esto generalmente no funciona para sistemas a la medida, puesto que dichos sistemas no existen en una versión estándar para todos los clientes.

PUNTOS CLAVE

- La administración de la configuración es la gestión de un sistema de software en evolución. Cuando se mantiene un sistema se establece un equipo CM para garantizar que los cambios se incorporen en el sistema de una forma controlada y que se mantienen registros con los detalles de los cambios que se implementaron.
- Los principales procesos de administración de la configuración se ocupan de la administración del cambio, gestión de versiones, construcción del sistema y gestión de entregas de software (release). Para apoyar todos estos procesos se dispone de herramientas de software.
- La administración del cambio implica valorar las propuestas de cambios por los clientes del sistema y otros interesados, así como decidir si es conveniente en términos de costos implementarlos en una nueva versión de un sistema.
- La gestión de versiones incluye hacer un seguimiento de las diferentes versiones de los componentes de software que se crean conforme se hacen cambios en ellos.
- La construcción del sistema es el proceso de ensamblar componentes de sistema en un programa ejecutable para que operen en un sistema de cómputo objetivo.
- El software debe reconstruirse frecuentemente y probarse de inmediato después de construir una nueva versión. Esto facilita la detección de bugs y problemas que se introdujeron desde la última construcción.
- Las entregas de sistema contienen código ejecutable, archivos de datos, archivos de configuración y documentación. La gestión de entregas incluye tomar decisiones referentes a las fechas de entrega del sistema, preparar toda la información para su distribución y documentar cada entrega del sistema.

LECTURAS SUGERIDAS

Configuration Management Principles and Practice. Este libro, muy completo, estudia los enfoques estándar y tradicionales a la CM, así como los enfoques CM que son más adecuados para los procesos modernos, como el desarrollo de software ágil. (Anne Mette Jonassen Hass, Addison-Wesley, 2002.)

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Un libro de fácil lectura y relativamente breve, que ofrece buenos consejos prácticos respecto a la práctica de administración de la configuración, en especial para métodos ágiles de desarrollo. (S. P. Berczuk con B. Appleton, Addison-Wesley, 2003.)

"High-level Best Practices in Software Configuration Management". Este artículo Web, escrito por el personal de un proveedor de herramientas CM, es una excelente introducción a la buena práctica en la administración de la configuración del software. (L. Wingerd y C. Seiwald, 2006.) http://www.perforce.com/perforce/papers/bestpractices.html.

"Agile Configuration Management for Large Organizations". Este artículo Web describe las prácticas de administración de la configuración que pueden usarse en procesos de desarrollo ágil, con particular énfasis en cómo puede escalarse a proyectos y compañías grandes. (P. Schuh, 2007.) http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html.

EJERCICIOS

25.1. Sugiera cinco posibles problemas que pudieran surgir si una compañía no desarrolla políticas y procesos efectivos de administración de la configuración.

25.2. ¿Cuáles son los beneficios de usar un formato de petición de cambio como el documento central en el proceso de administración del cambio?

25.3. Describa seis características esenciales que deben incluirse en una herramienta para soportar procesos de administración del cambio.

25.4. Explique por qué es esencial que toda versión de un componente deba identificarse de manera exclusiva. Comente acerca de los problemas de usar un esquema de identificación de versión que se base simplemente en números de versión.

25.5. Imagine una situación donde dos desarrolladores modifican simultáneamente tres componentes de software diferentes. ¿Qué dificultades pueden surgir cuando tratan de fusionar los cambios que realizaron?

25.6. El software se desarrolla cada vez más en el seno de equipos, cuyos miembros trabajan en diferentes lugares. Sugiera algunas características en un sistema de gestión de versiones que puedan requerirse para apoyar este desarrollo de software distribuido.

25.7. Describa las dificultades que podrían surgir cuando se construye un sistema a partir de sus componentes. ¿Qué problemas particulares pueden ocurrir cuando un sistema se construye en una computadora anfitriona para alguna máquina objetivo?

25.8. Con referencia a la construcción de sistemas, explique por qué en ocasiones se tienen que conservar computadoras obsoletas en las que se desarrollaron grandes sistemas de software.

25.9. Un problema común con la construcción de sistemas ocurre cuando los nombres de archivo físico se incorporan en código de sistema y la estructura de archivo implicada en dichos nombres no corresponde con la de la máquina objetivo. Escriba un conjunto de lineamientos que ayuden al programador a evitar esto y cualquier otro problema de construcción del sistema que pueda considerar.

25.10. Detalle cinco factores que deben tomar en cuenta los ingenieros durante el proceso de construcción de una entrega (release) de un amplio sistema de software.

REFERENCIAS

Ahern, D. M., Clouse, A. y Turner, R. (2001). *CMMI Distilled*. Reading, Mass.: Addison-Wesley.

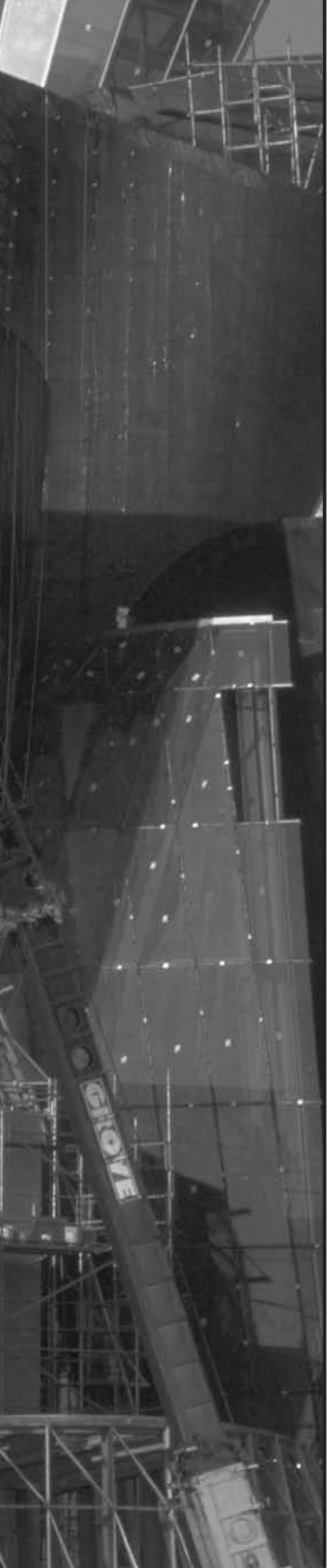
Bamford, R. y Deibler, W. J. (2003). "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach". Boca Raton, FL: CRC Press.

Paulk, M. C., Weber, C. V., Curtis, B. y Chrissis, M. B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley.

Peach, R. W. (1996). *The ISO 9000 Handbook, 3rd edition*. Nueva York: Irwin Professional Pub.

Pilato, C. M., Collins-Sussman, B. y Fitzpatrick, B. W. (2004). *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc.

Vesperman, J. (2003). *Essential CVS*. Sebastopol, Calif.: O'Reilly and Associates.



26

Mejora de procesos

Objetivos

El objetivo de este capítulo es introducirlo a la mejora del proceso de software como una forma de aumentar la calidad del software y reducir los costos de desarrollo. Al estudiar este capítulo:

- comprenderá las razones para la mejora del proceso de software como un medio que pretende mejorar tanto la calidad del producto como la eficiencia y efectividad de los procesos de software;
- entenderá los principios de mejora del proceso de software y el proceso de mejora del proceso cíclico;
- conocerá cómo puede usarse el enfoque Meta-Pregunta-Métrica para guiar la medición del proceso;
- estará al tanto de las ideas de capacidad y madurez de proceso, así como de la forma general del modelo CMMI del SEI para la mejora de procesos.

Contenido

26.1 El proceso de mejora de procesos
26.2 Medición del proceso
26.3 Análisis del proceso
26.4 Cambios en los procesos
26.5 El marco de trabajo para la mejora de procesos CMMI

En la actualidad existe una constante demanda de la industria por un mejor y más barato software, que debe entregarse en plazos cada vez más cortos. Por consiguiente, numerosas compañías de software han dirigido la atención hacia la mejora de procesos de software como una forma de aumentar la calidad de su software, reducir sus costos o acelerar los procesos de desarrollo. La mejora de procesos significa comprender los procesos existentes y cambiarlos para incrementar la calidad del producto o reducir los costos y el tiempo de desarrollo.

Se usan dos enfoques muy diferentes para la mejora y el cambio de procesos:

1. El enfoque de madurez de procesos, que se ha orientado en mejorar el proceso y la gestión del proyecto e introducir en una organización buenas prácticas de ingeniería de software. El nivel de madurez del proceso refleja la medida en que se adoptan buenas prácticas técnicas y administrativas en los procesos de desarrollo de software organizacional. Las metas principales de este enfoque consisten en mejorar la calidad del producto y la previsibilidad del proceso.
2. El enfoque ágil, orientado al desarrollo iterativo y la reducción de las sobrecargas en el proceso de software. Las características primarias de los métodos ágiles son la entrega rápida de funcionalidad y la capacidad de respuesta ante los cambiantes requerimientos del cliente.

Los partidarios de cada uno de estos enfoques generalmente son escépticos al reconocer los beneficios del otro. El enfoque de madurez del proceso se basa en el desarrollo orientado por un plan y, por regla general, requiere aumento de "sobrecarga", en el sentido de que se introducen actividades que no son directamente relevantes para la programación. Los enfoques ágiles se centran en el código a desarrollar y minimizan de manera deliberada la formalidad y la documentación.

En el capítulo 3 y otras partes del libro se estudiaron los métodos ágiles, así que este capítulo se enfoca en la administración y la mejora del proceso basado en la madurez. Esto no significa que se prefiera este enfoque a los métodos ágiles. De hecho, el autor considera que, para proyectos pequeños y medianos, adoptar las prácticas ágiles es probablemente la estrategia de mejora de proceso más efectiva en costo. Sin embargo, para sistemas grandes, sistemas críticos y sistemas que involucran a desarrolladores en diferentes compañías, los conflictos de administración a menudo son las razones por las que los proyectos pueden tener problemas. Para compañías cuyo negocio es la ingeniería de sistemas grandes y complejos, debe considerarse un enfoque a la mejora de procesos centrado en la madurez.

Como se explicó en el capítulo 24, el proceso de desarrollo usado para crear un sistema de software influye en la calidad de dicho sistema. Por eso, muchas personas creen que mejorar el proceso de desarrollo de software conducirá a un software de mejor calidad. Esta noción de mejora de procesos es una idea original del ingeniero estadounidense W. E. Deming, quien trabajó con la industria japonesa después de la Segunda Guerra Mundial para ayudar a mejorar la calidad. Durante muchos años, la industria japonesa se ha comprometido con la mejora continua de los procesos, lo que ha conducido al reconocimiento de la alta calidad de los bienes manufacturados en Japón.

Deming, entre otros, introdujo la idea del control estadístico de calidad. Ésta se basa en medir el número de defectos de productos y relacionarlos con el proceso. La meta es disminuir el número de defectos del producto al analizar y modificar el proceso para que se reduzcan las posibilidades de introducir defectos y mejore la detección de los mismos.

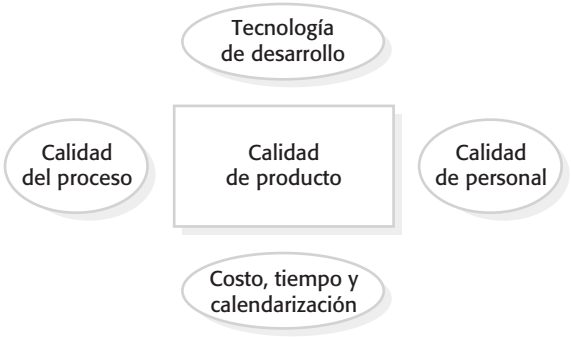


Figura 26.1 Factores que afectan el producto de software

Una vez lograda la reducción en el conteo de defectos, el proceso se estandariza y comienza un ciclo más de la mejora.

Humphrey (1988), en su libro de gran influencia sobre gestión de procesos, argumenta que pueden aplicarse las mismas técnicas a la ingeniería de software. Afirma:

> *W. E. Deming, en su trabajo con la industria japonesa después de la Segunda Guerra Mundial, aplicó a la industria los conceptos de control estadístico de procesos. Si bien existen importantes diferencias, dichos conceptos son tan aplicables al software como a los automóviles, las cámaras, los relojes de pulsera y el acero.*

Aunque hay claras similitudes, el autor no está de acuerdo con Humphrey en que los resultados de la ingeniería fabril puedan transferirse fácilmente a la ingeniería de software. Ahí donde la manufactura se incluye, es evidente la relación proceso/producto. Por lo general, la manufactura implica el ajuste de herramientas automatizadas y procesos de verificación de productos. Si alguien comete un error al calibrar una máquina, esto afectará a todos los productos elaborados por dicha máquina. Evitar los errores al configurar las máquinas e introducir procesos de verificación más efectivos claramente mejora la calidad del producto.

Esta relación calidad de proceso/calidad de producto es menos evidente cuando el producto es intangible y depende, en alguna medida, de procesos intelectuales que no pueden automatizarse. La calidad del software no está influida por sus procesos de fabricación, sino por su proceso de diseño, en el que las habilidades y la experiencia de la gente son significativas. En ciertos casos, el proceso utilizado puede ser el determinante más significativo de la calidad del producto. Sin embargo, para aplicaciones innovadoras en particular, el personal que interviene en el proceso tiene mayor influencia sobre la calidad que el proceso utilizado.

Para productos de software, o cualquier producto intelectual, como libros o películas, donde la calidad depende del diseño, hay cuatro factores elementales que afectan la calidad del producto, los cuales se muestran en la figura 26.1.

La influencia de cada uno de estos factores depende del tamaño y tipo del proyecto. Para sistemas muy grandes que incluyen subsistemas separados, desarrollados por equipos que pueden trabajar en diferentes lugares, el factor principal que afecta la calidad del producto es el proceso de software. Los mayores problemas con los proyectos grandes son la integración, la gestión del proyecto y las comunicaciones. Por lo general, existe una mezcla de habilidades y experiencia entre los miembros del equipo y, como

el proceso de desarrollo a menudo tiene lugar a lo largo de varios años, el equipo de desarrollo es volátil. Puede cambiar por completo durante la vida del proyecto.

Sin embargo, para proyectos pequeños, en los que sólo existen en el equipo algunos miembros, la calidad del equipo de desarrollo es más importante que el proceso de desarrollo utilizado. Por consiguiente, el manifiesto ágil proclama la importancia de la gente por encima de los procesos. Si el equipo tiene un alto nivel de habilidad y experiencia, es probable que la calidad del producto sea alta, independientemente del proceso utilizado. Si el equipo carece de experiencia y habilidad, un buen proceso puede limitar el daño, pero, en sí mismo, no llevará a un software de alta calidad.

Donde los equipos son reducidos, la buena tecnología de desarrollo es particularmente importante. El equipo pequeño no puede dedicar mucho tiempo a tediosos procedimientos administrativos. Los integrantes del equipo pasan la mayor parte de su tiempo diseñando y programando el sistema; por lo tanto, las buenas herramientas afectan significativamente su productividad. Para proyectos grandes es esencial un nivel básico de tecnología de desarrollo para la gestión de la información. No obstante, paradójicamente, las herramientas sofisticadas de software son menos importantes en los proyectos grandes. Los miembros del equipo dedican una menor proporción de su tiempo a las actividades de desarrollo y más tiempo para comunicarse y entender otras partes del sistema. Las herramientas de desarrollo no hacen diferencia en esto. Sin embargo, las herramientas Web 2.0 que soportan las comunicaciones, tales como los wikis y blogs, pueden mejorar de manera sustancial la comunicación entre los miembros de los equipos distribuidos.

Sin considerar los factores de personal, procesos o herramientas, si un proyecto tiene un presupuesto inadecuado o se planea con un calendario de entregas poco realista, la calidad del producto se verá afectada. Un buen proceso requiere de recursos para una implementación efectiva. Si dichos recursos son insuficientes, el proceso no puede ser realmente efectivo. Si los recursos son inadecuados, sólo personal de excelencia puede salvar el proyecto. Aun así, si el déficit es demasiado grande, la calidad del producto se degradará. Al no haber suficiente tiempo para el desarrollo, es probable que el software entregado tenga funcionalidad reducida o niveles más bajos de fiabilidad o rendimiento.

Con demasiada frecuencia, la causa real de los problemas en la calidad del software no es una gestión deficiente, procesos inadecuados o capacitación de escasa calidad. Más bien, es el hecho de que las organizaciones deben competir para sobrevivir. Para ganar un contrato, una compañía puede subestimar el esfuerzo requerido o prometer la entrega rápida de un sistema. Con la intención de cumplir estos compromisos, tal vez acuerde un calendario de desarrollo poco realista. En consecuencia, la calidad del software se ve afectada de manera negativa.

26.1 El proceso de mejora de procesos

En el capítulo 2 se introdujo la idea general de un proceso de software como una secuencia de actividades que, cuando se ejecuta, conduce a la producción de un sistema de software. Se describieron procesos genéricos, como el modelo en cascada y el desarrollo basado en reutilización; además, se estudiaron las actividades más importantes del proceso. Dichos procesos genéricos se ejemplifican dentro de una organización para crear el proceso particular que se utiliza en el desarrollo del software.

Los procesos de software pueden ser observados en todas las organizaciones, desde compañías integradas por una sola persona hasta grandes multinacionales. Dichos

Característica del proceso	Cuestiones clave
Comprensión	¿En qué medida el proceso está definido explícitamente y qué tan fácil es entender la definición del proceso?
Estandarización	¿Hasta qué punto el proceso se basa en el proceso genérico estándar? Esto puede ser importante para algunos clientes que requieran la conformidad con un conjunto de estándares definidos de proceso. ¿Hasta dónde se usa el mismo proceso en todas las partes de una compañía?
Visibilidad	¿Las actividades del proceso culminan en resultados claros, de modo que el avance del proceso se observa externamente?
Mensurabilidad	¿El proceso incluye recolección de datos u otras actividades que permitan medir las características del proceso o del producto?
Soportabilidad	¿En qué medida pueden usarse herramientas de software para apoyar las actividades del proceso?
Aceptabilidad	¿El proceso definido es aceptable y útil para los ingenieros responsables de elaborar el producto de software?
Fiabilidad	¿El proceso está diseñado de tal forma que se evitan o se detectan errores de proceso antes de que deriven en errores de producto?
Robustez	¿El proceso puede continuar a pesar de problemas inesperados?
Mantenibilidad	¿El proceso puede evolucionar para reflejar los requerimientos cambiantes de la organización o mejoras identificadas en el proceso?
Rapidez	¿Qué tan rápido puede completarse el proceso de entrega de un sistema a partir de una especificación dada?

Figura 26.2 Atributos de proceso

procesos son de diferentes tipos dependiendo del grado de formalidad del proceso, los tipos de productos desarrollados, el tamaño de la organización, etcétera. No existe algo como un proceso de software "ideal" o "estándar" que sea aplicable en todas las organizaciones o para todos los productos de software de un tipo particular. Cada compañía debe desarrollar su propio proceso en función de su tamaño, sus antecedentes y las habilidades de su personal, el tipo de software que ha desarrollado, los requerimientos del cliente y del mercado, y la cultura empresarial.

Por lo tanto, la mejora de procesos no significa simplemente adoptar métodos o herramientas particulares o usar un proceso genérico publicado. Aunque las organizaciones que desarrollan el mismo tipo de software sin duda tienen mucho en común, siempre existen factores organizacionales locales, procedimientos y estándares que influyen en el proceso. Ocasionalmente se tendrá éxito al introducir mejoras de proceso si sólo se intenta cambiar el proceso por uno usado en otra parte. Siempre hay que considerar el entorno y la cultura locales y cómo esto puede verse afectado por las propuestas de cambio del proceso.

También hay que considerar qué aspectos del proceso se desea mejorar. La meta podría ser mejorar la calidad del software y, en tal caso, habría que introducir nuevas actividades de proceso que cambien la forma en que se desarrolla y prueba el software. Quizás usted esté interesado en mejorar algunos atributos del proceso en sí y deba determinar cuáles atributos de proceso son los más importantes para su compañía. En la figura 26.2 se muestran algunos ejemplos de atributos de proceso que pueden ser objeto de mejora.

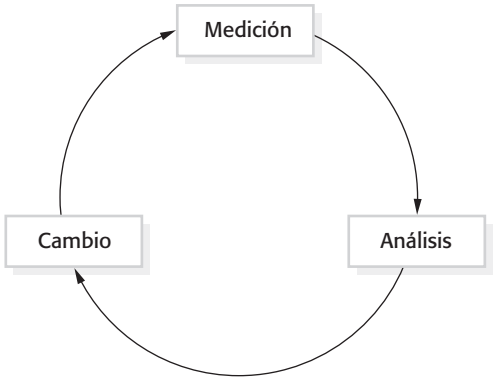


Figura 26.3 El ciclo de mejora de procesos

Está claro que, en ocasiones, dichos atributos se relacionan de forma positiva y otras veces de forma negativa. Por lo tanto, un proceso que califica alto en el atributo de visibilidad también será probablemente comprensible. Un observador del proceso puede inferir la existencia de actividades a partir de las salidas producidas. Por otra parte, la visibilidad del proceso puede estar inversamente relacionada con la rapidez. Hacer visible un proceso requiere que las personas implicadas tengan que generar información sobre el proceso en sí. Esto puede reducir la producción del software, debido al tiempo que toma elaborar esos documentos.

Es imposible hacer mejoras de proceso que optimicen simultáneamente todos los atributos de proceso. Por ejemplo, si su meta es tener un proceso de desarrollo rápido, entonces tal vez haya que reducir la visibilidad del proceso. Si quiere hacer un proceso más mantenible, hay que adoptar procedimientos y herramientas que reflejen la amplia práctica de la organización y que se usen en diferentes partes de la compañía. Esto podría reducir la aceptabilidad local del proceso. Probablemente los ingenieros tengan que introducir procedimientos locales y herramientas no estandarizadas para apoyar su forma de trabajar. En tanto que éstos sean efectivos, no querrán renunciar a ellos en favor de un proceso estandarizado.

El proceso de mejora de procesos es cíclico, como se observa en la figura 26.3. Incluye tres subprocesos:

1. *Medición del proceso* Se miden atributos del proyecto actual o el producto. La meta es mejorar las medidas de acuerdo con los objetivos de la organización implicada la mejora del proceso. Esto constituye una línea de referencia que ayuda a determinar si son efectivas las mejoras del proceso.
2. *Análisis del proceso* Se valora el proceso actual y se identifican las debilidades y los cuellos de botella del proceso. Durante esta etapa pueden desarrollarse modelos de proceso (llamados también mapas de proceso) que describen el proceso. El análisis puede enfocarse al considerar las características del proceso como rapidez y robustez.
3. *Cambio del proceso* Los cambios al proceso son propuestas para atacar algunas de las debilidades identificadas del proceso. Estos cambios se introducen y el ciclo vuelve a recopilar datos sobre la efectividad de los cambios.

Sin datos concretos respecto a un proceso o software desarrollado usando dicho proceso, es imposible estimar el valor de la mejora del proceso. Sin embargo, es improbable que las compañías que inician el proceso de mejora de procesos cuenten con datos de proceso disponibles como una línea de referencia para la mejora. Por eso, como parte del primer ciclo de cambios, quizás haya que introducir actividades de proceso para recopilar datos sobre el proceso de software y medir las características del producto de software.

La mejora del proceso es una actividad de largo plazo, así que cada una de las etapas en el proceso de mejora puede durar varios meses. También es una actividad continua pues, independientemente de los procesos introducidos, el ambiente de negocios cambiará y los nuevos procesos tendrán que evolucionar para tomar en cuenta dichos cambios.

26.2 Medición del proceso

Las mediciones del proceso son datos cuantitativos acerca del proceso de software, como el tiempo que tarda en realizarse cierta actividad del proceso. Por ejemplo, es posible medir el tiempo requerido para desarrollar casos de prueba del programa. Humphrey (1989), en su libro referente a la mejora de procesos, argumenta que la medición del proceso y los atributos del producto son esenciales para la mejora de los procesos. Además, refiere que la medición desempeña un importante papel en la mejora de procesos personales a pequeña escala (Humphrey, 1995), donde los individuos tratan de ser más productivos.

Las mediciones de procesos pueden usarse para valorar si la eficiencia de un proceso mejoró o no. Por ejemplo, se puede monitorizar el esfuerzo y el tiempo dedicados a las pruebas. Las mejoras efectivas al proceso de pruebas deben reducir el esfuerzo o tiempo de pruebas. No obstante, las mediciones del proceso, por sí mismas, no pueden usarse para determinar si mejoró la calidad del producto. También deben recopilarse datos de calidad del producto (véase el capítulo 24) y relacionarse con las actividades del proceso.

Pueden recopilarse tres tipos de métricas de proceso:

1. *El tiempo que tarda en completarse un proceso particular* Éste puede ser el tiempo total dedicado al proceso, tiempo calendario, tiempo empleado en el proceso por ciertos ingenieros en particular, etcétera.
2. *Los recursos requeridos para un proceso particular* Los recursos pueden incluir esfuerzo total en días-hombre, costos de viaje o recursos de cómputo.
3. *El número de ocurrencias de un evento particular* Los ejemplos de eventos que pueden monitorizarse incluyen el número de defectos descubiertos durante la inspección del código, el número de cambios solicitados a los requerimientos y el número promedio de líneas de código modificadas en respuesta a un cambio de requerimientos.

Los primeros dos tipos de medición pueden usarse para descubrir si los cambios al proceso mejoraron la eficiencia de un proceso. Suponga que en un proceso de desarrollo

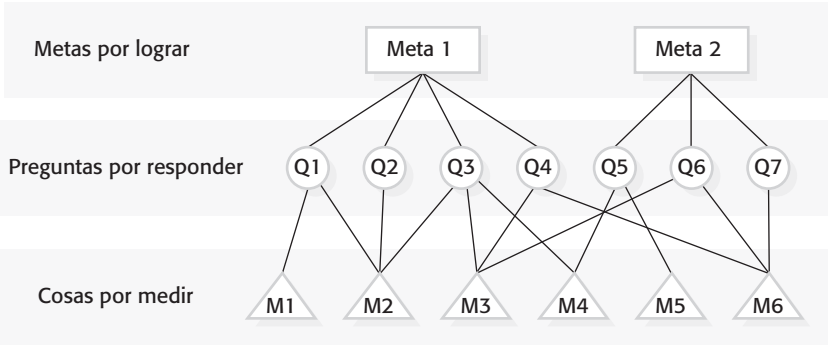


Figura 26.4 El paradigma GQM

de software hay puntos fijos, como la aceptación de requerimientos, la conclusión del diseño arquitectónico o la finalización de la generación de datos de prueba. Es posible medir el tiempo y esfuerzo requeridos para moverse de uno de estos puntos fijos a otro. Después de introducir los cambios, las mediciones de los atributos del sistema pueden indicar si los cambios al proceso tuvieron éxito para reducir el tiempo o esfuerzo requerido.

Las mediciones de la cantidad de eventos que se producen tienen una relación más directa con la calidad del software. Por ejemplo, aumentar el número de defectos descubiertos al cambiar el proceso de inspección del programa se reflejará probablemente en una mejora en la calidad del producto. Sin embargo, esto debe confirmarse mediante posteriores mediciones del producto.

Una dificultad fundamental en la medición del proceso es conocer qué información respecto al proceso debe recopilarse para apoyar la mejora de los procesos. Basili y Rombach (1988) proponen lo que llaman el paradigma GQM (Meta-Pregunta-Métrica, por las siglas de *Goal-Question-Metric*), que se usa ampliamente en la medición del software y los procesos. Basili y Green (1993) describen cómo se usa este enfoque en un programa de mejora de procesos a largo plazo basado en mediciones, en la agencia espacial estadounidense NASA.

El paradigma GQM (figura 26.4) se usa en la mejora de procesos para ayudar a responder tres preguntas fundamentales:

1. ¿Por qué se introduce la mejora de procesos?
2. ¿Qué información se necesita para ayudar a identificar y valorar las mejoras?
3. ¿Qué mediciones de proceso y producto se requieren para obtener esta información?

Estas preguntas se relacionan directamente con las abstracciones (metas, preguntas, métricas) en el paradigma GQM:

1. *Metas* Una meta es algún objetivo que la organización pretende lograr. No debe ocuparse directamente de los atributos del proceso, sino más bien de cómo el

proceso afecta los productos o a la organización en sí. Ejemplos de metas pueden ser un mejor nivel de madurez de procesos (véase la sección 26.5), un tiempo de desarrollo de producto más corto o un aumento en la fiabilidad del producto.

2. *Preguntas* Se trata de mejoras de las metas, en las que se identifican áreas específicas de incertidumbre relacionadas con las metas. Por lo general, una meta tendrá algunas preguntas asociadas que necesiten responderse. Ejemplos de preguntas relacionadas con la meta de acotar los tiempos de desarrollo de productos pueden ser: ¿dónde están los cuellos de botella en el proceso actual?, ¿cómo puede reducirse el tiempo requerido para finalizar los requerimientos de producto con los clientes?, ¿cuántas pruebas son efectivas para descubrir defectos de producto?

3. *Métricas* Se trata de mediciones que deben recopilarse para ayudar a responder las preguntas y confirmar si las mejoras del proceso lograron o no las metas deseadas. Para ayudar a responder las preguntas anteriores, se pueden recopilar datos acerca del tiempo que tarda en completarse cada actividad del proceso (normalizado por tamaño de sistema), el número de comunicaciones formales entre clientes y usuarios para cada cambio de requerimientos, y la cantidad de defectos descubiertos mediante la ejecución de pruebas.

La ventaja de usar el enfoque GQM en la mejora de procesos es que separa las preocupaciones de la organización (las metas) de preocupaciones específicas del proceso (las preguntas). Proporciona una base para determinar cuáles datos deben recopilarse y sugiere que los datos recopilados deben analizarse en diferentes formas, dependiendo de la pregunta que se pretende responder.

El enfoque GQM se desarrolló y combinó con el modelo de madurez de capacidad del SEI (Paulk *et al*., 1995) en el método AMI (por las siglas de *Analyze*, *Measure*, *Improve*, es decir, Analizar, Medir, Mejorar) de mejora de procesos del software. Los desarrolladores del método AMI proponen un enfoque por etapas para la mejora de procesos, en el que las mediciones comienzan después de que una organización introduce cierta estandarización en sus procesos, en lugar de comenzar las mediciones de inmediato. El manual AMI (Pulford *et al*., 1996) ofrece lineamientos y consejos prácticos sobre cómo implementar la mejora de los procesos con base en mediciones.

Como se estudió en el capítulo 24, en ocasiones es problemático interpretar lo que en realidad significan las mediciones. Por ejemplo, suponga que se mide el tiempo promedio tomado para reparar los bugs que se reportaron en el software entregado para pruebas externas. Éste es el tiempo entre la recepción de un reporte de error por parte del equipo y el momento en que este reporte se marca formalmente como "aclarado". Se introduce una nueva herramienta basada en la Web para reportar errores y, después de usar algún tiempo la herramienta, se observa que se reduce el tiempo para reparar los bugs reportados.

Entonces es posible afirmar que la introducción de las herramientas para reportar errores redujo realmente el tiempo para reparar los bugs. Cuando se observan cambios en una métrica, siempre es tentador atribuir dichos cambios a los cambios de proceso introducidos. Sin embargo, es peligroso hacer suposiciones simplistas en lo tocante a las mejoras. Los cambios en una métrica pueden ser causa de algo completamente diferente, como un cambio del personal en el equipo de proyecto, cambios a la calendarización del

Análisis de práctica de proceso

Un enfoque al análisis de procesos es usar cuestionarios para descubrir la medida en que se utilizan las buenas prácticas de ingeniería de software. Por lo tanto, para algunas etapas en el proceso, como la ingeniería de requerimientos, es posible identificar cuáles prácticas son más adecuadas para el tipo de sistema a desarrollar en una compañía y formular preguntas sobre qué tan ampliamente se utilizan. Éste es el enfoque que se sugiere en el presente libro en relación con la mejora de procesos en la ingeniería de requerimientos (Sommerville y Sawyer, 1997).

http://www.SoftwareEngineering-9.com/Web/ProcImp/goodpractice.html

proyecto o cambios administrativos. También es posible que se modifique la práctica del equipo, tan sólo porque se hace una medición. En el caso de las herramientas para reportar errores, algunas de las razones por las que se observa el cambio incluyen:

1. El nuevo sistema puede tener carga reducida y, así, más tiempo disponible para reparar bugs. Esto conduce a una reducción en los tiempos promedios para "reparar bugs". La mejora del proceso pudo hacer una diferencia real.
2. El nuevo sistema tal vez no haga la diferencia con el tiempo que en verdad se tarda en corregir los bugs, pero puede facilitar el registro de información. Por lo tanto, los tiempos de reparación de bugs se miden más exactamente con el nuevo sistema. No hay un cambio real en el tiempo promedio para corregir bugs.
3. Las mediciones antes de que el nuevo sistema se introdujera se hicieron, tal vez, a través de las pruebas de un sistema. Los bugs que eran más fáciles y rápidos de corregir ya se habían corregido y sólo permanecieron los "bugs duros" que tardaban más en repararse. Sin embargo, después de introducir el sistema de reporte de bugs, las mediciones se hicieron al comienzo de las pruebas del nuevo sistema y los bugs corregidos fueron los "bugs sencillos", que podían repararse rápidamente.
4. Un nuevo gerente del equipo de pruebas pudo instruir a los miembros del equipo para reportar las inconsistencias de la interfaz de usuario como bugs, mientras que antes se ignoraban. Esto significó el reporte de muchos más "bugs sencillos" que podían corregirse rápidamente.

La medición es una forma de generar evidencia respecto a un proceso y los cambios de proceso. No obstante, esta evidencia tiene que interpretarse junto con otra información sobre el proceso antes de poder estar seguros de que son efectivos los cambios al proceso. Siempre se debe usar la medición en conjunto con la valoración cualitativa de los cambios. Esto implica hablar con las personas implicadas en el proceso acerca de los cambios que se introdujeron, y obtener su impresión de la efectividad de los mismos. Esto no sólo revela otros factores que pudieron influir en el proceso, sino también la medida en la que el equipo adoptó los cambios propuestos y cómo éstos afectaron la práctica de desarrollo actual.

26.3 Análisis del proceso

El análisis del proceso es el estudio de los procesos para ayudar a entender sus características clave y cómo las personas implicadas realizan en la práctica dichos procesos. En la figura 26.3 se sugiere que el análisis del proceso sigue a la medición del proceso. Ésta es una simplificación porque, en realidad, tales actividades están entrelazadas. Es necesario realizar cierto análisis para saber qué medir y, al hacer las mediciones, se desarrolla inevitablemente una comprensión más a fondo del proceso a medir.

El análisis del proceso tiene un número de objetivos relacionados estrechamente:

1. Comprender las actividades implicadas en el proceso y las relaciones entre dichas actividades.
2. Entender las relaciones entre las actividades del proceso y las mediciones realizadas.
3. Relacionar el proceso o procesos específicos analizados con procesos comparables en otras partes de la organización, o idealizar procesos del mismo tipo.

Durante el análisis del proceso se trata de comprender lo que sucede en un proceso. Se busca información acerca de los problemas e ineficiencias del proceso. También se debe estar interesado en la medida en que el proceso se utiliza, las herramientas de software empleadas para apoyar el proceso, y en cómo el proceso está influido por restricciones de la organización. La figura 26.5 muestra algunos de los aspectos del proceso que se pueden investigar durante el análisis del proceso.

Las técnicas usadas más comúnmente para el análisis del proceso son:

1. *Cuestionarios y entrevistas* Los ingenieros y administradores que trabajan en un proyecto se preguntan sobre lo que sucede realmente. Las respuestas para un cuestionario formal se mejoran durante las entrevistas personales con los implicados en el proceso. Como se estudia más adelante, la discusión puede estructurarse en torno a modelos de procesos de software.
2. *Estudios etnográficos* En los estudios etnográficos (véase el capítulo 4) se observa a los participantes en el proceso mientras trabajan, y los estudios pueden usarse para entender la naturaleza del desarrollo del software como una actividad humana. Tales análisis revelan sutilezas y complejidades que pueden no revelarse en cuestionarios y entrevistas.

Cada uno de estos enfoques tiene ventajas y desventajas. El análisis basado en cuestionarios puede realizarse muy rápidamente una vez identificadas las preguntas correctas. Sin embargo, si las preguntas están mal planteadas o son inadecuadas, esto podría desencadenar en una comprensión incompleta o imprecisa del proceso. Además, el análisis basado en cuestionarios quizá les parezca una forma de valoración o evaluación. Por lo tanto, los ingenieros encuestados pueden dar respuestas que consideren que serán del agrado de los encuestadores, y no la verdad sobre el proceso utilizado.

Las entrevistas con el personal implicado en el proceso son más abiertas que los cuestionarios. Inicie con un guión de preguntas preparado, pero adáptelas de acuerdo con las

Aspecto del proceso	Preguntas
Adopción y estandarización	¿El proceso está documentado y estandarizado en toda la organización? Si no lo está, ¿significa que algunas mediciones son sólo específicas para una instancia del proceso? Si los procesos no están estandarizados, entonces los cambios a un proceso pueden no ser transferibles a procesos comparables en otras partes de la compañía.
Práctica de ingeniería de software	¿Existen buenas y conocidas prácticas de ingeniería de software que no se incluyan en el proceso? ¿Por qué no se incluyen? ¿La falta de estas prácticas afecta las características del producto, tales como el número de defectos en un sistema de software entregado?
Restricciones organizacionales	¿Cuáles son las restricciones de la organización que afectan el diseño del proceso y las formas en que se realizan los procesos? Por ejemplo, si el proceso implica lidiar con material clasificado, puede haber actividades en el proceso para comprobar que la información clasificada no se incluya en algún material que se entregará a organizaciones externas. Las restricciones de la organización pueden significar que no es posible hacer cambios a los procesos.
Comunicaciones	¿Cómo se gestionan las comunicaciones en el proceso? ¿Cómo se relacionan los conflictos en las comunicaciones con las mediciones realizadas del proceso? Los problemas en la comunicación son un conflicto grave en muchos procesos, y los cuellos de botella en las comunicaciones con frecuencia son las razones para las demoras del proyecto.
Introspección	¿El proceso es reflexivo (es decir, los actores que participan en el proceso piensan explícitamente y discuten el proceso y cómo puede mejorarse)? ¿Hay mecanismos mediante los cuales los actores del proceso puedan sugerir mejoras a los procesos?
Aprendizaje	¿Cómo aprenden los individuos que se unen a un equipo de desarrollo sobre los procesos de software utilizados? ¿La compañía tiene manuales de proceso y programas de capacitación de procesos?
Soporte de herramientas	¿Qué aspectos del proceso están soportados por las herramientas de software y cuáles no lo están? Para áreas sin soporte, ¿existen herramientas que puedan desplegarse de forma rentable para ofrecer soporte? Para áreas con soporte, ¿las herramientas son efectivas y eficientes? ¿Hay mejores herramientas disponibles?

Figura 26.5
Aspectos de análisis del proceso

respuestas que obtenga de diferentes personas. Si da a los participantes la oportunidad de discutir los temas más ampliamente, descubrirá que los participantes hablan acerca de los problemas en los procesos, las formas en que los procesos cambian en la práctica, etcétera.

Casi en todos los procesos, los individuos implicados hacen cambios locales para adaptar el proceso a las circunstancias locales. Los análisis etnográficos tienen más probabilidad que las entrevistas de descubrir el verdadero uso del proceso. Sin embargo, este tipo de análisis suele ser una actividad prolongada que tal vez dure varios meses, ya que depende de la observación externa del proceso conforme éste se realiza. Para hacer un análisis completo, uno tiene que involucrarse desde las etapas iniciales de un proyecto hasta la entrega y el mantenimiento del producto. Para proyectos grandes, esto podría durar varios años, así que

no es muy práctico hacer un análisis etnográfico completo de los procesos en un proyecto grande. Los análisis etnográficos en realidad son más útiles cuando se requiere una comprensión profunda de fragmentos del proceso. Una vez identificadas las áreas que necesitan más investigación a partir del material de entrevistas, entonces es posible enfocarse en el estudio etnográfico enfocado a descubrir detalles del proceso.

Cuando analice un proceso, con frecuencia será útil comenzar con un modelo que defina las actividades del proceso y las entradas y salidas de dichas actividades. El modelo puede incluir también información sobre los actores del proceso: Los individuos o los roles responsables de realizar las actividades y los entregables críticos que deben producirse. Es conveniente usar una notación informal para describir los modelos de proceso o notaciones tabulares más formales, diagramas de actividad UML, o una notación de modelado de procesos empresariales, como BPMN (que se estudió en el capítulo 19). En este libro existen muchos ejemplos de modelos de proceso usados para presentar y describir los procesos de software.

Los modelos de proceso son una buena forma de enfocar la atención en las actividades en un proceso y la transferencia de información entre tales actividades. Dichos modelos de proceso no tienen que ser formales o completos, ya que su objetivo es provocar discusión más que documentar a detalle el proceso. La discusión con los individuos implicados en el proceso y las observaciones de dicho proceso se estructuran a menudo en torno a un conjunto de preguntas respecto al modelo de proceso formal. Ejemplos de estas preguntas son:

1. ¿Qué actividades tienen lugar en la práctica, pero no se muestran en el modelo? Inevitablemente, los modelos están incompletos, pero si diferentes personas identifican distintas actividades faltantes, esto indica que el proceso no se realiza de manera consistente a través de la organización.
2. ¿Hay actividades del proceso, mostradas en el modelo, que (el actor del proceso) considera ineficientes? ¿En qué formas son ineficientes y cómo pueden mejorarse? ¿Cómo dichas actividades ineficientes afectan las mediciones del proceso que pudieran realizarse?
3. ¿Qué ocurre cuando las cosas salen mal? ¿El equipo sigue el proceso definido en el modelo, o abandonan el proceso y se toman acciones de emergencia? Si el proceso se abandona, esto indica que los ingenieros de software no creen que el proceso sea suficientemente bueno o que éste tenga suficiente flexibilidad para manipular excepciones.
4. ¿Quiénes son los actores que participan en las diferentes etapas del proceso y cómo se comunican? ¿Qué cuellos de botella se presentan comúnmente en el intercambio de información?
5. ¿Qué soporte de herramientas se usa para las actividades que se muestran en el modelo? ¿Éste es efectivo y de uso universal? ¿Cómo podría mejorarse el soporte de herramientas?

Cuando se completa un análisis del proceso de software, se tiene una comprensión más profunda de dicho proceso y del potencial para mejorar los procesos en el futuro. También se deben comprender las restricciones sobre la mejora de los procesos y cómo puede esto limitar el alcance de las mejoras que pueden introducirse.

Modelado de procesos de software

El modelado de procesos de software comenzó a principios de la década de 1980 (Osterweil, 1987) con el objetivo a largo plazo de usar modelos de proceso como una forma de organizar y coordinar el soporte de herramientas para los procesos. Un modelo de proceso debe incluir información concerniente a las actividades del proceso, entradas y salidas, así como acerca de los actores que participan en el proceso. Se desarrollaron notaciones de modelado de procesos de software de propósito especial, pero se han sustituido principalmente por notaciones para modelado de procesos empresariales, como BPMN (White, 2004) o modelos de actividad UML.

http://www.SoftwareEngineering-9.com/Web/ProcImp/spm.html

26.3.1 Excepciones de proceso

Los procesos de software son entidades complejas. En una organización puede haber un modelo de proceso definido, pero esto tal vez sólo represente la situación en la que el equipo de desarrollo no se enfrenta con algunos problemas imprevistos. En realidad, los problemas imprevistos son un hecho cotidiano para los líderes de proyecto. El modelo de proceso "ideal" debe modificarse de manera dinámica conforme se encuentren soluciones a dichos problemas. Los ejemplos de los tipos de excepción que debe enfrentar un líder de proyecto incluyen:

- Muchas personas clave se enferman al mismo tiempo, justo antes de una revisión crítica del proyecto;
- una violación grave en la seguridad de la computadora, lo que significa que todas las comunicaciones externas están fuera de acción durante varios días;
- una reorganización de la compañía, lo que significa que los administradores tienen que pasar buena parte de su tiempo trabajando en asuntos de la organización y no en la gestión del proyecto;
- una petición no prevista de escribir una propuesta para un nuevo proyecto, lo cual significa que el esfuerzo debe transferirse del proyecto actual a la elaboración de una propuesta.

Básicamente, una excepción afectará, y por lo general alterará en alguna forma, los recursos, presupuestos o calendarización de un proyecto. Es difícil predecir todas las excepciones por adelantado e incorporarlas en un modelo de proceso formal. Por lo tanto, con frecuencia habrá que arreglárselas para manejar las excepciones y luego cambiar dinámicamente el proceso "estándar" para hacer frente a dichas circunstancias inesperadas.

26.4 Cambios en los procesos

El cambio al proceso implica hacer modificaciones al proceso existente. Como se sugirió, esto incluye introducir nuevas prácticas, métodos o herramientas; cambiar el orden de las actividades del proceso; introducir o eliminar entregables del proceso; mejorar

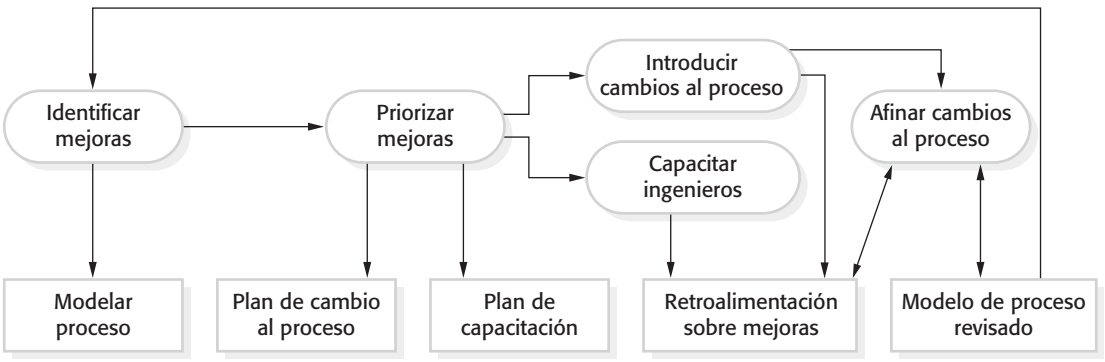


Figura 26.6 El proceso de cambios al proceso

las comunicaciones, o introducir nuevos roles y responsabilidades. Los cambios al proceso deben estar dirigidos por metas de mejora, tales como "reducir en un 25% el número de defectos descubiertos durante las pruebas de integración". Después de implementar los cambios, se usan las mediciones del proceso para valorar la efectividad de éstos.

Existen cinco etapas clave en el proceso de cambios al proceso (figura 26.6):

1. *Identificación de mejoras* Esta etapa se ocupa de usar los resultados del análisis del proceso para identificar las formas de enfrentar los problemas de calidad, los cuellos de botella en la programación o las ineficiencias en costo que se identificaron durante el análisis del proceso. Se pueden sugerir nuevos procesos, estructuras de proceso, métodos y herramientas para enfrentar los problemas de los procesos. Por ejemplo, tal vez una compañía considere que muchas de sus dificultades de software surgen de problemas con los requerimientos. Al usar una guía de mejores prácticas para la ingeniería de requerimientos (Sommerville y Sawyer, 1997), es posible identificar varias prácticas de ingeniería de requerimientos que pudieran introducirse o cambiarse.

2. *Priorización de las mejoras* Esta etapa se ocupa de valorar los posibles cambios a los procesos y priorizarlos para su implementación. Cuando se identifican muchos posibles cambios, por lo general es imposible introducirlos todos a la vez, y determinar cuáles son los más importantes. Estas decisiones se pueden tomar con base en la necesidad de mejorar áreas de proceso específicas, los costos de introducir un cambio, los efectos de un cambio en la organización u otros factores. Por ejemplo, una compañía quizá considere la introducción de procesos de gestión de requerimientos para administrar los requerimientos en evolución como el cambio de proceso de prioridad más alta.

3. *Introducción de cambios a los procesos* La introducción de cambios al proceso significa establecer nuevos procedimientos, métodos y herramientas, e integrarlos con otras actividades de proceso. Hay que permitir suficiente tiempo para introducir cambios y garantizar que éstos sean compatibles con otras actividades de proceso, así como con estándares y procedimientos organizacionales. Esto tal vez implique adquirir herramientas para gestión de requerimientos y diseñar procesos para usar dichas herramientas.

4. *Capacitación de proceso* Sin capacitación es imposible obtener todos los beneficios de los cambios al proceso. Los ingenieros implicados necesitan comprender los cambios que se propusieron y cómo realizar los procesos nuevos y modificados. Con demasiada frecuencia, los cambios a los procesos se imponen sin adecuada capacitación y el efecto de dichos cambios es degradar, antes que mejorar, la calidad del producto. En el caso de gestión de requerimientos, la capacitación puede comprender una discusión del valor de la gestión de los requerimientos, una explicación de las actividades del proceso y una introducción de las herramientas seleccionadas.
5. *Afinación del cambio* Los cambios de proceso propuestos nunca serán completamente efectivos al momento de introducirlos. Se requiere una fase de afinación en que puedan descubrirse problemas menores, para entonces proponer e introducir modificaciones al proceso. Esta fase de afinación debe durar varios meses hasta que los ingenieros de desarrollo estén satisfechos con el nuevo proceso.

Por lo general, no es recomendable introducir muchos cambios al mismo tiempo. Además de las dificultades de capacitación que esto provoca, la introducción de numerosos cambios hace imposible valorar en el proceso el efecto de cada cambio. Una vez introducido un cambio, el proceso de mejora puede iterarse con un subsecuente análisis usado para identificar problemas del proceso, proponer mejoras o alguna otra actividad.

Además de los problemas de valorar la efectividad de los procesos de cambio estudiados, existen dos dificultades importantes que pueden enfrentar quienes participan en los procesos de cambio:

1. *Resistencia al cambio* Los miembros del equipo o líderes de proyecto pueden oponerse a la introducción de cambios al proceso y exponer razones por las que los cambios no funcionarán, o retrasar la introducción de cambios. En algunos casos, podrían obstruir de manera deliberada los cambios a los procesos, o interpretar los datos para demostrar la ineficiencia de los cambios propuestos a los procesos.
2. *Persistencia del cambio* Aunque es posible introducir inicialmente cambios a los procesos, es común que se desechen las innovaciones a los procesos después de poco tiempo y que los procesos vuelvan a su estado anterior.

La resistencia al cambio puede provenir tanto de los líderes del proyecto como de los ingenieros que intervienen en el proceso a cambiar. Con frecuencia, los líderes de proyecto se resisten al cambio en los procesos porque cualquier innovación tiene riesgos desconocidos asociados con éste. Los cambios al proceso tal vez pretendan acelerar la producción del software o reducir los defectos del software. Sin embargo, siempre existe el peligro de que dichos cambios al proceso sean ineficaces o que el tiempo requerido para introducir los cambios sea mayor al tiempo ahorrado. Se juzga a los líderes de proyecto en función de si su proyecto produce o no software a tiempo y dentro del presupuesto. Por lo tanto, quizá prefieran un proceso ineficiente, pero predecible, que un proceso mejorado que tenga beneficios para la organización y que, al mismo tiempo, conlleve riesgos asociados a corto plazo.

Los ingenieros pueden resistirse por razones similares a la introducción de nuevos procesos, o porque consideran que dichos procesos son una amenaza para su profesionalismo. Esto es, tal vez sientan que el nuevo proceso predefinido les da menos discrecio-

nalidad o que no reconoce el valor de sus habilidades y experiencia. Quizá piensen que el nuevo proceso signifique que se requerirá menos personal y que ellos podrían perder sus empleos. Probablemente no quieran aprender nuevas habilidades, herramientas o formas de trabajar.

Como líder, usted tiene que ser sensible a los sentimientos de las personas afectadas cuando se introducen cambios a los procesos. Debe incluir todo el tiempo al equipo en el del proceso de cambio, comprender sus dudas e invitarlos a participar en la planeación del nuevo proceso. Al hacerlos partícipes en el cambio del proceso, es mucho más probable que deseen hacer el trabajo. La reingeniería de procesos empresariales (Hammer, 1990; Ould, 1995), una moda de la década de 1990 que implicó hacer cambios radicales a los procesos, no tuvo éxito debido principalmente a que fracasó para tomar en cuenta las preocupaciones legítimas de los individuos implicados.

Para hacer frente a las preocupaciones de los líderes de proyecto de que el proceso de cambio afectará de manera negativa la planeación y los costos del proyecto, hay que aumentar el presupuesto del proyecto para permitir costos y demoras adicionales que resulten del cambio. También hay que ser realistas respecto a los beneficios a corto plazo del cambio. Es improbable que los cambios conduzcan a mejoras inmediatas a gran escala. Los beneficios del cambio a los procesos son a largo plazo antes que de corto plazo, de manera que se debe dar apoyo a los cambios al proceso a través de varios proyectos.

Es común el problema de los cambios introducidos que posteriormente se modifican. Los cambios quizá sean propuestos por un "evangelista", quien cree firmemente que los cambios conducirán a mejoras. Tal vez trabaje arduamente para garantizar que los cambios sean efectivos y se acepte el nuevo proceso. Sin embargo, si el "evangelista" se marcha, entonces puede sustituirlo alguien que esté menos comprometido con el nuevo proceso. Por lo tanto, es probable que los individuos implicados simplemente regresen a las formas anteriores de hacer las cosas. Esto es más probable si los cambios introducidos no se adoptaron universalmente y aún no se materializan todos los beneficios de los cambios al proceso.

Debido a problemas de persistencia del cambio, el modelo CMMI, estudiado en la sección 26.5, argumenta firmemente a favor de la institucionalización de los cambios a los procesos. Esto significa que los cambios a los procesos no dependen de los individuos, sino que los cambios se vuelven parte de la práctica estándar de la compañía, con apoyo y capacitación a lo largo de toda ella.

26.5 El marco de trabajo para la mejora de procesos CMMI

El Software Engineering Institute (SEI) se estableció para mejorar las capacidades de la industria de software estadounidense. A mediados de la década de 1980, el SEI inició un estudio de formas para valorar las capacidades de los contratistas de software. El resultado de esta valoración de capacidad fue el Modelo de Madurez de Capacidades de Software (CMM, por las siglas de *Software Capability Maturity Model*) del SEI (Paulk *et al*., 1993; Paulk *et al*., 1995). Dicho modelo ha influido enormemente para convencer a la comunidad de la ingeniería de software de tomar con seriedad la mejora de procesos. Al CMM de software le siguió una variedad de modelos de madurez de capacidades,

incluido el Modelo de Madurez de Capacidades del Personal (P-CMM) (Curtis *et al*., 2001) y el Modelo de Capacidades de Ingeniería de Sistemas (Bate, 1995).

Otras organizaciones han desarrollado también modelos comparables de madurez de proceso. El enfoque SPICE a la valoración de capacidades y la mejora de procesos (Paulk y Konrad, 1994) es más flexible que el modelo SEI. Incluye niveles de madurez comparables con los niveles CMM, pero también identifica procesos, como los procesos cliente-proveedor, que atraviesan dichos niveles. Conforme aumenta el nivel de madurez, también debe mejorar el rendimiento de dichos procesos transversales.

El proyecto Bootstrap en la década de 1990 tenía la meta de extender y adaptar el modelo de madurez SEI para hacerlo aplicable a través de un amplio rango de compañías. Este modelo (Haase *et al*., 1994; Kuvaja *et al*., 1994) usa los niveles de madurez de SEI (que se explican en la sección 26.5.1). De igual forma, propone un modelo de proceso base (fundado en el modelo que se usa en la Agencia Espacial Europea) que puede emplearse como punto de partida para la definición local de procesos. Incluye lineamientos cuya finalidad es desarrollar un sistema de calidad en toda una compañía para apoyar la mejora de procesos.

Con la intención de integrar el cúmulo de modelos de capacidad con base en la noción de madurez de proceso (incluidos sus propios modelos), el SEI se embarcó en un nuevo programa para desarrollar un modelo de capacidad integrado (CMMI). El marco CMMI sustituye los CMM de ingeniería de software y sistemas, e integra otros modelos de madurez de capacidades. Tiene dos ejemplificaciones, en etapas y continuo, y enfrenta algunas de las debilidades reportadas en el CMM de software.

El modelo CMMI (Ahern *et al*., 2001; Chrissis *et al*., 2007) tiene la intención de ser un marco para la mejora de procesos con amplia aplicabilidad a través de varias compañías. Su versión en etapas es compatible con el CMM de software y permite que los procesos de desarrollo y gestión del sistema de una organización se valoren asignándoles un nivel de madurez de 1 a 5. Su versión continua permite una clasificación más fina de la madurez del proceso. Este modelo ofrece una forma de clasificar 22 áreas de proceso (figura 26.7) en una escala de 0 a 5.

El modelo CMMI es muy complejo, con más de 1,000 páginas de descripción. Aquí se simplificó principalmente para su análisis. Los principales componentes del modelo son:

1. Un conjunto de áreas de proceso que se relacionan con las actividades de proceso del software. El CMMI identifica 22 áreas de proceso que son relevantes para la capacidad y la mejora del proceso de software. Están organizadas en cuatro grupos en el modelo CMMI continuo. Dichos grupos y las áreas de proceso relacionadas se listan en la figura 26.7.

2. Algunas metas, las cuales son descripciones abstractas de un estado deseable que debe lograr una organización. El CMMI tiene metas específicas que se asocian con cada área de proceso y definen el estado deseable de dicha área. También define metas genéricas asociadas con la institucionalización de la buena práctica. La figura 26.8 muestra ejemplos de metas específicas y genéricas en el CMMI.

3. Un conjunto de buenas prácticas, las cuales son descripciones de formas para lograr una meta. Muchas prácticas específicas y genéricas pueden asociarse con cada meta dentro de un área de proceso. En la figura 26.9 se muestran algunos ejemplos de

Categoría	Área de proceso
Gestión de procesos	Definición de proceso organizacional (OPD) Enfoque de proceso organizacional (OPF) Capacitación organizacional (OT) Rendimiento de proceso organizacional (OPP) Innovación y despliegue organizacional (OID)
Gestión de proyectos	Planeación de proyecto (PP) Monitorización y control de proyecto (PMC) Gestión de acuerdo con proveedor (SAM) Gestión de proyectos integrados (IPM) Gestión de riesgos (RSKM) Gestión cuantitativa de proyectos (QPM)
Ingeniería	Gestión de requerimientos (REQM) Desarrollo de requerimientos (RD) Solución técnica (TS) Integración de producto (PI) Verificación (VER) Validación (VAL)
Soporte	Administración de la configuración (CM) Gestión de la calidad de proceso y producto (PPQA) Medición y análisis (MA) Análisis y resolución de decisiones (DAR) Análisis y resolución causal (CAR)

Figura 26.7 Áreas de proceso en el CMMI

prácticas recomendadas. Sin embargo, el CMMI reconoce que lo importante es la meta, más que la forma en que se alcanza dicha meta. Las organizaciones pueden usar cualquier práctica adecuada para lograr cualquiera de las metas CMMI: No tienen que adoptar las prácticas recomendadas por el CMMI.

Las metas y prácticas genéricas no son técnicas, sino que están asociadas con la institucionalización de buenas prácticas, lo que significa que esto depende de la madurez de la organización. En una etapa temprana del desarrollo de madurez, la institucionalización quizá pretenda garantizar que se establezcan los planes en la compañía y se definan los procesos para todo el desarrollo de software. No obstante, para una organización con procesos avanzados más maduros, la institucionalización puede suponer introducir control de procesos mediante técnicas estadísticas y otras técnicas cuantitativas a través de la organización.

Meta	Área de proceso
Se manejan acciones correctivas para cerrar cuando el rendimiento o los resultados del proyecto se desvían significativamente del plan.	Monitorización y control del proyecto (meta específica)
El rendimiento y avance reales del proyecto se monitorizan tomando como base el plan del proyecto.	Monitorización y control del proyecto (meta específica)
Los requerimientos se analizan y validan; además, se desarrolla una definición de la funcionalidad requerida.	Desarrollo de requerimientos (meta específica)
Se determinan sistemáticamente las causas de raíz de los defectos y otros problemas.	Análisis causal y resolución (meta específica)
El proceso se institucionaliza como un proceso definido.	Meta genérica

Figura 26.8 Áreas de proceso en el CMMI

Una valoración CMMI implica examinar los procesos en una organización y clasificar dichos procesos o áreas de proceso en una escala de seis puntos que se relacionan con el nivel de madurez en cada área de proceso. La idea es que, cuanto más maduro sea el proceso, mejor será. La siguiente es la escala de seis puntos que asigna un nivel de madurez a un área de proceso:

1. *Incompleto* Al menos no se satisface una de las metas específicas asociadas con el área de proceso. No hay metas genéricas en este nivel, pues no tiene sentido la institucionalización de un proceso incompleto.
2. *Realizado* Las metas asociadas con el área de proceso están satisfechas, y para todos los procesos el alcance del trabajo a realizar se estableció de manera explícita y se comunicó a los miembros del equipo.
3. *Gestionado* En este nivel se satisfacen las metas asociadas con el área de proceso y se establecen políticas organizacionales que determinan cuándo debe usarse cada proceso. Tiene que haber planes de proyecto documentados que establezcan las metas del proyecto. En la institución debe haber procedimientos para la gestión de recursos y la monitorización de procesos.
4. *Definido* Este nivel se enfoca en la estandarización organizacional y el despliegue de procesos. Cada proyecto tiene un proceso gestionado que se adapta a los requerimientos del proyecto desde un conjunto definido de procesos organizacionales. Deben recopilarse activos y mediciones de proceso, además de usarse para futuras mejoras de proceso.
5. *Gestionado cuantitativamente* En este nivel hay una responsabilidad organizacional cuya finalidad es usar métodos estadísticos y cuantitativos para controlar los subprocesos; esto es, deben utilizarse mediciones recopiladas de proceso y producto en la gestión del proceso.
6. *Optimizado* En este nivel superior, la organización debe usar las mediciones de proceso y producto para impulsar la mejora de los procesos. Hay que analizar las tendencias y adaptar los procesos a las necesidades cambiantes de la empresa.

Meta	Prácticas asociadas
Los requerimientos se analizan y validan; además, se desarrolla una definición de la funcionalidad requerida.	Analizar sistemáticamente los requerimientos derivados para asegurar que éstos son necesarios y suficientes.
	Validar los requerimientos para garantizar que el producto resultante se desempeñará como se pretende en el entorno del usuario, empleando en su caso múltiples técnicas según sea apropiado.
Se determinan sistemáticamente las causas de raíz de los defectos y otros problemas.	Seleccionar para análisis los defectos críticos y otros problemas.
	Realizar análisis causales de los defectos seleccionados y otros problemas, y proponer acciones para enfrentarlos.
El proceso se institucionaliza como un proceso definido.	Establecer y mantener una política organizacional para planear y realizar el proceso de desarrollo de requerimientos.
	Asignar responsabilidad y autoridad para realizar el proceso, desarrollar los productos de trabajo y ofrecer los servicios del proceso de desarrollo de requerimientos.

Figura 26.9 Metas y prácticas asociadas en el CMMI

Ésta es una descripción muy simplificada de los niveles de capacidad y, para ponerla en práctica, se necesita trabajar con descripciones más detalladas. Los niveles son progresivos, con las descripciones explícitas del proceso en los niveles inferiores, pasando a través de la estandarización de procesos hacia el cambio y la mejora de los procesos dirigidos por las mediciones del proceso y el software, que se ubican en el nivel más alto. Para mejorar sus procesos, una compañía debe tener como objetivo aumentar el nivel de madurez de los grupos de procesos que son relevantes para su negocio.

26.5.1 El modelo CMMI en etapas

El modelo CMMI en etapas es comparable con el Modelo de Madurez de Capacidades de Software en el sentido de que ofrece un medio para valorar la capacidad de proceso de una organización en uno de cinco niveles, y prescribe las metas que deben lograrse en cada uno de dichos niveles. La mejora de proceso se logra al implementar prácticas en cada nivel, y desplazarse en el modelo de los niveles inferiores a los superiores.

En la figura 26.10 se muestran los cinco niveles en el modelo CMMI por etapas. Corresponden a los niveles de capacidad 1 a 5 en el modelo continuo. La principal diferencia entre los modelos CMMI por etapas y continuo es que el primero se usa para valorar la capacidad de la organización como un todo, mientras que el segundo mide la madurez de áreas de proceso específicas dentro de la organización.

Cada nivel de madurez tiene un conjunto de áreas de proceso y metas genéricas asociadas. Éstas reflejan la buena práctica de ingeniería y gestión de software, además de la institucionalización de la mejora de los procesos. Los niveles de madurez más bajos

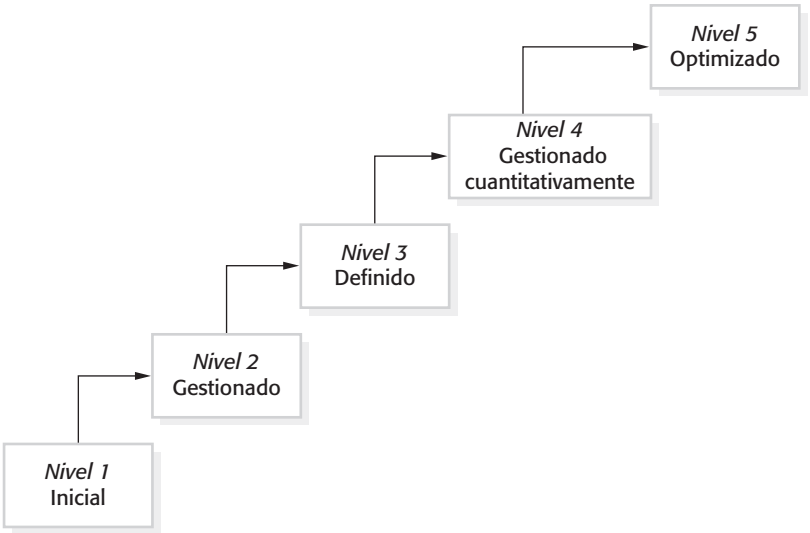


Figura 26.10 El modelo de madurez CMMI por etapas

pueden alcanzarse al introducir buenas prácticas; sin embargo, los niveles más altos requieren un compromiso con la medición y la mejora de los procesos.

Por ejemplo, las áreas de proceso definidas en el modelo asociado con el segundo nivel (el nivel gestionado) son:

1. *Gestión de requerimientos* Gestionar los requerimientos de los productos y componentes de producto del proyecto, así como identificar inconsistencias entre dichos requerimientos y los planes y productos de trabajo del proyecto.
2. *Planeación de proyecto* Establecer y mantener planes que definan las actividades del proyecto.
3. *Monitorización y control del proyecto* Facilitar la comprensión del avance del proyecto, de forma que puedan tomarse las acciones correctivas adecuadas cuando el rendimiento del proyecto se desvíe significativamente del plan.
4. *Gestión de acuerdos con el proveedor* Gestionar la adquisición de productos y servicios de proveedores externos al proyecto con los que existan acuerdos formales.
5. *Medición y análisis* Desarrollar y sostener una capacidad de medición que sirva para apoyar necesidades de información gerencial.
6. *Aseguramiento de calidad de proceso y producto* Ofrecer al personal y a la gerencia una perspectiva objetiva de los procesos y productos de trabajo asociados.
7. *Administración de la configuración* Establecer y mantener la integridad de los productos de trabajo mediante la identificación de la configuración, control de la configuración, registro del estatus de la configuración y auditorías a la configuración.

Además de estas prácticas específicas, las organizaciones que operan en el segundo nivel en el modelo CMMI deben lograr la meta genérica de institucionalizar cada uno de

los procesos como un proceso gestionado. Los siguientes son algunos ejemplos de prácticas institucionales asociadas con la planeación de proyectos que conducen al proceso de planeación de proyecto a convertirse en un proceso gestionado:

- Establecer y mantener una política organizacional para planear y realizar el proceso de planeación del proyecto.
- Brindar recursos adecuados para realizar el proceso de gestión de proyecto, desarrollar los productos de trabajo y ofrecer los servicios del proceso.
- Monitorizar y controlar el proceso de planeación del proyecto con base en el plan y tomar acciones correctivas adecuadas.
- Revisar las actividades, el estatus y los resultados del proceso de planeación de proyecto con gestión de alto nivel, y resolver cualquier conflicto.

La ventaja del CMMI por etapas consiste en que es compatible con el modelo de madurez de capacidades de software que se propuso a mediados de la década de 1980. Numerosas compañías comprendieron y están comprometidas con el uso de este modelo para la mejora de procesos. Por lo tanto, es sencillo para ellas hacer una transición de este al modelo CMMI en etapas. Más aún, el modelo en etapas define una clara ruta de mejora para las organizaciones. Éstas pueden planear su paso del segundo al tercer nivel, y así sucesivamente.

Sin embargo, la principal desventaja del modelo en etapas (y del CMM de software) es su naturaleza prescriptiva. Cada nivel de madurez tiene sus propias metas y prácticas. El modelo en etapas supone que todas las metas y prácticas en un nivel se implementaron antes de la transición al siguiente nivel. Sin embargo, las circunstancias organizacionales pueden ser tales que sea más adecuado implementar las metas y prácticas a niveles superiores antes que las prácticas a nivel más bajo. Cuando una organización hace esto, una valoración de la madurez dará una imagen engañosa de su capacidad.

26.5.2 El modelo CMMI continuo

Los modelos de madurez continuos no clasifican a una organización de acuerdo con niveles discretos. En vez de ello, son modelos de grano más fino que consideran prácticas individuales o en grupos y valoran el uso de la buena práctica dentro de cada grupo de procesos. Por lo tanto, la valoración de la madurez no es un solo valor, sino un conjunto de valores que muestran la madurez de la organización en cada proceso o grupo de procesos.

El CMMI continuo considera las áreas de proceso que se muestran en la figura 26.7 y asigna el nivel de valoración de capacidad de 0 a 5 (como se describió anteriormente) a cada área del proceso. Por lo general, las organizaciones operan en diferentes niveles de madurez para distintas áreas de proceso. En consecuencia, el resultado de una valoración CMMI continua es un perfil de capacidad que muestra cada área de proceso y su valoración de capacidad asociada. En la figura 26.11 se presenta un fragmento de un perfil de capacidad que muestra los procesos en diferentes niveles de capacidad. Esto indica que el nivel de madurez en la administración de la configuración, por ejemplo, es alto, pero que la madurez en la gestión del riesgo es baja. Una compañía puede desarrollar perfiles de capacidad reales y objetivo, donde el perfil objetivo refleja el nivel de capacidad que le gustaría alcanzar para dicha área de proceso.

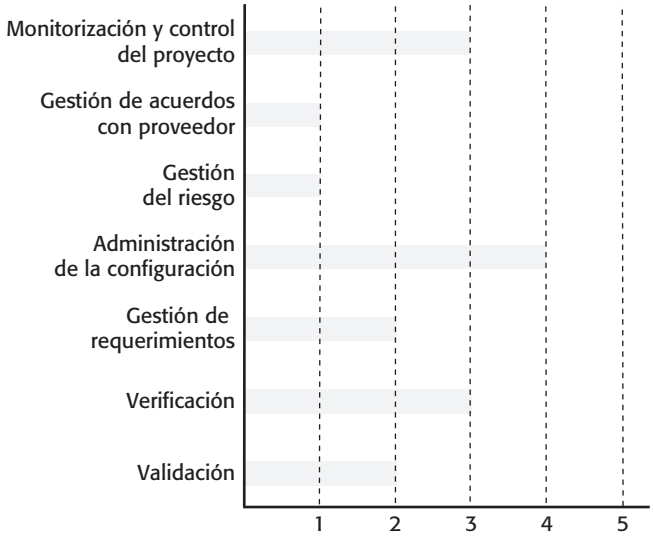


Figura 26.11 Un perfil de capacidad de proceso

La principal ventaja del modelo continuo es que las compañías pueden elegir procesos para mejorar de acuerdo con sus necesidades y requerimientos particulares. Diferentes tipos de organizaciones tienen distintos requerimientos para la mejora de los procesos. Por ejemplo, una compañía que desarrolle software para la industria aeroespacial puede enfocarse en mejorar la especificación del sistema, la administración de la configuración y la validación, mientras que una compañía de desarrollo Web tal vez esté más interesada en los procesos que enfrenta el cliente. El modelo en etapas requiere que las compañías se enfoquen a la vez en diferentes etapas. En contraste, el CMMI continuo permite discreción y flexibilidad, al tiempo que acepta que las compañías trabajen dentro del marco de mejora CMMI.

PUNTOS CLAVE

- Las metas de mejora de procesos son una mayor calidad de los productos, reducción en costos de proceso y entrega más rápida del software.
- Los principales enfoques para la mejora de procesos son los enfoques ágiles, orientados a reducir los costos de los procesos, y los enfoques fundados en la madurez sobre la base de una mejor gestión de procesos y el uso de buenas prácticas de ingeniería de software.

- El ciclo de mejora de procesos incluye la medición, el análisis, el modelado y el cambio de los procesos.
- Los modelos de proceso, que muestran las actividades en un proceso y sus relaciones con los productos de software, se usan para descripciones de procesos. Sin embargo, en la práctica, los ingenieros que participan en el desarrollo de software siempre adaptan modelos a las circunstancias locales.
- La medición debe usarse para responder preguntas específicas sobre el proceso de software utilizado. Dichas preguntas deben basarse en metas de mejora organizacional.
- Los tres tipos de métricas de proceso que se usan en el proceso de medición son: métricas de tiempo, métricas de utilización de recursos y métricas de eventos.
- El modelo de madurez de proceso CMMI es un modelo de mejora de proceso integrado que soporta tanto el proceso de la mejora continua como en etapas.
- La mejora de los procesos en el modelo CMMI se basa en alcanzar un conjunto de metas relacionadas con las buenas prácticas de ingeniería de software y la descripción, estandarización y control de las prácticas utilizadas para lograr dichas metas. El modelo CMMI incluye prácticas recomendadas que pueden usarse, pero éstas no son obligatorias.

LECTURAS SUGERIDAS

"Can you trust software capability evaluations?" Este artículo echa un vistazo escéptico al tema de la evaluación de la capacidad, en la que se valora la madurez de los procesos de una compañía, y analiza por qué dichas evaluaciones pueden no ofrecer una imagen verdadera de la madurez de una organización. (E. O'Connell y H. Saiedian, *IEEE Computer*, **33** (2), febrero de 2000) http://dx.doi.org/10.1109/2.820036.

Software Process Improvement: Results and Experience from the Field. Este libro es una colección de ensayos enfocados en estudios de caso de mejora de procesos en varias compañías noruegas, pequeñas y medianas. También incluye una excelente introducción a los asuntos generales de la mejora de procesos. (Conradi, R., Dybå, T., Sjøberg, D., Ulsund, T. (eds.), Springer, 2006.)

CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition. Una completa descripción del CMMI. El CMMI es amplio y complejo, además de que resulta prácticamente imposible hacerlo fácil de leer y entender. Este libro realiza una labor razonable al incluir cierto material anecdótico e histórico, pero a veces es un tanto denso. (M. B. Chrissis, M. Konrad, S. Shrum, Addison-Wesley, 2007.)

EJERCICIOS

26.1. ¿Cuáles son las diferencias importantes entre el enfoque ágil y el enfoque de madurez del proceso para la mejora de los procesos de software?

26.2. ¿Ante qué circunstancias es probable que la calidad de producto esté determinada por la calidad del equipo de desarrollo? Dé ejemplos de los tipos de productos de software que son particularmente dependientes del talento y la habilidad individuales.

26.3. Mencione tres herramientas de software especializadas que puedan desarrollarse para soportar un programa de mejora de procesos en una organización.

26.4. Suponga que la meta de mejora de procesos en una organización es aumentar el número de componentes de reutilización que se producen durante el desarrollo. Sugiera tres preguntas en el paradigma GQM a las que esto pueda conducir.

26.5. Describa tres tipos de métrica de proceso de software que puedan recopilarse como parte de un proceso de mejora de procesos. Dé un ejemplo de cada tipo de métrica.

26.6. Diseñe un proceso para valorar y priorizar propuestas de cambios a procesos. Documente esto como un modelo que muestre los roles implicados en este proceso. Debe usar diagramas de actividad UML o BPMN para describir el proceso.

26.7. Escriba dos ventajas y dos desventajas del enfoque a la valoración y la mejora de procesos que se exprese en los marcos de mejora de procesos como el CMMI.

26.8. ¿En qué circunstancias recomendaría el uso de la representación en etapas del CMMI?

26.9. ¿Cuáles son las ventajas y desventajas de usar un modelo de madurez de proceso que se enfoque en las metas a lograr, y no en la introducción de buenas prácticas?

26.10. ¿Considera que los programas de mejora de procesos, que implican medir el trabajo del personal en el proceso e introducir cambios en dicho proceso, pueden ser inherentemente insensibles? ¿Qué resistencia podría surgir ante un programa de mejora de procesos y por qué?

REFERENCIAS

Ahern, D. M., Clouse, A. y Turner, R. (2001). *CMMI Distilled*. Reading, Mass.: Addison-Wesley.

Basili, V. y Green, S. (1993). "Software Process Improvement at the SEL". *IEEE Software,* **11** (4), 58–66.

Basili, V. R. y Rombach, H. D. (1988). "The TAME Project: Towards Improvement-Oriented Software Environments". *IEEE Trans. on Software Eng.,* **14** (6), 758–773.

Bate, R. 1995. "A Systems Engineering Capability Maturity Model Version 1.1". Software Engineering Institute.

Chrissis, M. B., Konrad, M. y Shrum, S. (2007). *CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition.* Boston: Addison-Wesley.

Curtis, B., Hefley, W. E. y Miller, S. A. (2001). *The People Capability Model: Guidelines for Improving the Workforce*. Boston: Addison-Wesley.

Haase, V., Messnarz, R., Koch, G., Kugler, H. J. y Decrinis, P. (1994). "Bootstrap: Fine Tuning Process Assessment". *IEEE Software,* **11** (4), 25–35.

Hammer, M. (1990). "Reengineering Work: Don't Automate, Obliterate". *Harvard Business Review,* julio agosto de 1990, 104–112.

Humphrey, W. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.

Humphrey, W. S. (1988). "Characterizing the Software Process". *IEEE Software,* **5** (2), 73–79.

Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Reading, Mass.: Addison-Wesley.

Kuvaja, P., Similä, J., Krzanik, L., Bicego, A., Saukkonen, S. y Koch, G. (1994). *Software Process Assessment and Improvement: The BOOTSTRAP Approach*. Oxford: Blackwell Publishers.

Osterweil, L. (1987). "Software Processes are Software Too". *9th Int. Conf. on Software Engineering*, IEEE Press, 2–12.

Ould, M. A. (1995). *Business Processes: Modeling and Analysis for Re-engineering and Improvement*. Chichester: John Wiley & Sons.

Paulk, M. C., Curtis, B., Chrissis, M. B. y Weber, C. V. (1993). "Capability Maturity Model, Version 1.1". *IEEE Software,* **10** (4), 18–27.

Paulk, M. C. y Konrad, M. (1994). "An Overview of ISO's SPICE Project". *IEEE Computer,* **27** (4), 68–70.

Paulk, M. C., Weber, C. V., Curtis, B. y Chrissis, M. B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley.

Pulford, K., Kuntzmann-Combelles, A. y Shirlaw, S. (1996). *A Quantitative Approach to Software Management*. Wokingham: Addison-Wesley.

Sommerville, I. y Sawyer, P. (1997). *Requirements Engineering: A Good Practice Guide*. Chichester: John Wiley & Sons.

White, S. A. 2004. "An Introduction to BPMN". http://www.bpmn.org/Documents/Introduction%20to%20BPMN.

Glosario

Ada

Lenguaje de programación que desarrolló el Departamento de Defensa estadounidense en la década de 1980 como un lenguaje estándar para desarrollar software militar. Se basa en investigación acerca de lenguajes de programación de la década de 1970 e incluye sentencias como tipos de datos abstractos y soporte para concurrencia. Todavía se utiliza en grandes sistemas militares y aeroespaciales complejos.

administración de la configuración

Proceso de administrar los cambios a un producto de software en evolución. La administración de la configuración implica planeación de la configuración, gestión de versiones, construcción de sistema y administración del cambio.

administración del cambio

Proceso para registrar, comprobar, analizar, estimar e implementar los cambios propuestos a un sistema de software.

análisis estático

Análisis basado en herramientas del código fuente de un programa para descubrir errores y anomalías. Las anomalías, como las asignaciones sucesivas a una variable sin uso intermedio, pueden ser indicadores de errores de programación.

arquitectura cliente-servidor

Modelo arquitectónico para sistemas distribuidos donde la funcionalidad del sistema se ofrece como un conjunto de servicios proporcionados por un servidor. A ellos acceden computadoras cliente que usan los servicios. Variantes de este enfoque, como las arquitecturas cliente-servidor de tres capas, usan múltiples servidores.

arquitectura de referencia

Arquitectura genérica idealizada que incluye todas las características que pueden incorporar los sistemas. Es una manera de informar a los diseñadores acerca de la estructura general de dicha clase de sistema en lugar de una base para crear una arquitectura de sistema específica.

arquitectura de software

Modelo de la estructura y organización fundamentales de un sistema de software.

arquitectura dirigida por modelo (*model-driven architecture*, MDA)

Enfoque al desarrollo de software con base en la construcción de un conjunto de modelos de sistema, que pueden procesarse de forma automática o semiautomática para generar un sistema ejecutable.

aseguramiento de la calidad (*quality assurance*, QA)

Proceso global para definir cómo puede lograrse la calidad del software y cómo la organización que desarrolla el software sabe que éste satisface el nivel de calidad requerido.

ataque de negación de servicio

Un ataque en un sistema de software basado en Web que trata de sobrecargar el sistema de forma que no pueda proporcionar su servicio normal a los usuarios.

BEA

Un proveedor estadounidense de sistemas ERP.

bomba de insulina

Dispositivo médico controlado mediante software que puede suministrar dosis controladas de insulina a personas que sufren de diabetes. Se usó como estudio de caso en varios capítulos de este libro.

BPMN

Business Process Modeling Notation (Notación para el Modelado de Procesos de Negocio). Una notación para definir flujos de trabajo.

C

Lenguaje de programación que originalmente se desarrolló para implementar el sistema Unix. C es un lenguaje de implementación de sistema de nivel relativamente bajo, que permite el acceso al hardware del sistema y que puede compilarse a código eficiente. Se usa ampliamente para programación de sistemas de bajo nivel y desarrollo de sistemas embebidos.

C#

Lenguaje de programación orientado a objetos, desarrollado por Microsoft, que tiene mucho en común con C++, pero que incluye características que permiten más comprobación de escritura a tiempo de compilación.

C++

Lenguaje de programación orientado a objetos que es un superconjunto de C.

CASE (*Computer-Aided Software Engineering*), ingeniería de software auxiliada por computadora

El proceso de desarrollar software usando soporte automatizado.

caso de confiabilidad

Documento estructurado que se usa para respaldar las afirmaciones realizadas por un desarrollador de un sistema acerca de la confiabilidad de este último.

caso de seguridad

Argumento estructurado de que un sistema es seguro y/o está protegido. Muchos sistemas críticos deben tener casos de seguridad asociados que se valoran y aprueban mediante reguladores externos antes de que el sistema se certifique para su uso.

caso de uso

Especificación de un tipo de interacción con un sistema.

ciclo de vida del software

Con frecuencia se usa como otro nombre para el proceso de software; originalmente se acuñó para referirse al modelo en cascada del proceso de software.

clase de objeto

Una clase define los atributos y las operaciones de los objetos. Los objetos se crean en tiempo de ejecución al ejemplificar la definición de clase. El nombre de la clase se puede usar como un tipo de nombre en algunos lenguajes orientados a objetos.

CMM (Capability Maturity Model)

El modelo de madurez de capacidad del Software Engineering Institute, que se usa para valorar el nivel de madurez de desarrollo del software en una organización. Aunque lo desplazó el CMMI, todavía se usa ampliamente.

CMMI

Enfoque integrado al modelado de madurez de capacidad de proceso con base en la adopción de buenas prácticas de ingeniería de software y gestión de calidad integrada. Apoya el modelado de madurez discreto y continuo, e integra modelos de madurez de sistemas y de procesos de ingeniería de software.

cobertura de prueba

Efectividad de las pruebas del sistema para probar el código de todo un sistema. Algunas compañías tienen estándares para cobertura de prueba (por ejemplo, las pruebas del sistema deben garantizar que todos los enunciados del programa se ejecuten al menos una vez).

código de ética y práctica profesional

Conjunto de lineamientos que establecen el comportamiento ético y profesional esperado por parte de los ingenieros de software. Lo definieron las principales sociedades profesionales estadounidenses (la ACM y el IEEE) y define el comportamiento ético bajo ocho encabezados: público, cliente y empleador, producto, juicio, administración, colegas, profesión y uno mismo.

COM+

Modelo de componente y middleware de soporte diseñado para usar en plataformas de Microsoft; actualmente lo reemplaza .NET.

componente

Unidad de software independiente y portable que está completamente definido y al que se accede a través de un conjunto de interfaces.

computación en nube

La provisión de computación y/o servicios de aplicación a través de Internet con el uso de una "nube" de servidores de un proveedor externo. La "nube" se implementa usando un gran número de computadoras y tecnología de virtualización para usar de manera efectiva dichos sistemas.

confiabilidad

La confiabilidad de un sistema es una propiedad agregada que toma en cuenta la protección, fiabilidad, disponibilidad, seguridad y otros atributos del sistema. La confiabilidad de un sistema refleja el grado en el que los usuarios pueden confiar en él.

construcción de sistema

Proceso de compilar los componentes o las unidades que constituyen un sistema y los vinculan con otros componentes para crear un programa ejecutable. La construcción del sistema por lo general es automatizada, por lo que se minimiza la recompilación. Esta automatización puede instalarse en el sistema de procesamiento de lenguaje (como en Java) o puede implicar herramientas de software para soportar construcción del sistema.

CORBA (arquitectura común de intermediarios en petición de objetos, *Common Request Broker Architecture*)

Conjunto de estándares propuestos por el Object Management Group (OMG) que define modelos de objetos distribuidos y comunicaciones de objetos; es influyente en el desarrollo de sistemas distribuidos, pero actualmente se utiliza rara vez.

CVS

Herramienta de software de fuente abierta ampliamente usada para gestión de versiones.

desarrollo de software orientado a aspectos

Un enfoque del desarrollo de software que combina desarrollo generativo y basado en componentes. Se identifican las competencias, intereses, asuntos o propiedades (*concerns*) transversales en un programa y la implementación de esas competencias se definen como aspectos. Los aspectos incluyen una definición de dónde se incorporan en un programa. Luego, un tejedor (*weaver*) de aspectos teje los aspectos en los lugares adecuados del programa.

desarrollo dirigido por modelo (*model-driven development*, MDD)

Enfoque a la ingeniería de software centrado en modelos de sistema que se expresan en UML, en vez de utilizar código de lenguaje de programación. Esto extiende la MDA al considerar actividades distintas al desarrollo, como la ingeniería de requerimientos y las pruebas.

desarrollo dirigido por pruebas

Enfoque al desarrollo del software, donde se escriben pruebas ejecutables antes del código del programa. El conjunto de pruebas se corre automáticamente después de cada cambio al programa.

desarrollo incremental

Enfoque al desarrollo de software donde éste se entrega y despliega en incrementos.

desarrollo iterativo

Enfoque al desarrollo de software donde los procesos de especificación, diseño, programación y pruebas están entremezclados.

desarrollo orientado a objetos (OO)

Enfoque al desarrollo de software donde las abstracciones fundamentales en el sistema son objetos independientes. El mismo tipo de abstracción se usa durante la especificación, el diseño y el desarrollo.

desarrollo rápido de aplicación (*rapid application development*, RAD)

Enfoque al desarrollo de software dirigido a la entrega rápida del software. Con frecuencia implica el uso de programación de bases de datos y herramientas de soporte de desarrollo, como generadores de pantalla y reportes.

detección de fallas

Uso de procesos y comprobación en tiempo de operación para detectar y eliminar fallas en el desarrollo de un programa antes de que den por resultado una falla en la operación del sistema.

diagrama de clase

Tipos de diagrama UML que muestran las clases de objetos en un sistema y sus relaciones.

diagrama de estado

Tipo de diagrama UML que muestra los estados de un sistema y los eventos que disparan una transición de un estado a otro.

diagrama de secuencia

Diagrama que muestra la secuencia de las interacciones requeridas para completar cierta operación. En UML, los diagramas de secuencia pueden asociarse con casos de uso.

dinámica de evolución de programa

Estudio de las formas en las que cambia un sistema de software en evolución. Se afirma que las leyes de Lehman gobiernan la dinámica de la evolución del programa.

diseño de interfaz de usuario

Proceso de diseñar la forma en la que los usuarios del sistema pueden ingresar a la funcionalidad de éste, y la forma en que se despliega la información producida por el sistema.

disponibilidad

La facilidad con la que un sistema proporciona servicios cuando se le solicitan. Por lo general, la disponibilidad se expresa como un número decimal, de manera que una disponibilidad de 0.999 significa que el sistema puede entregar servicios para 999 de 1,000 unidades de tiempo.

dominio

Área problemática o empresarial específica donde se usan los sistemas de software. Los ejemplos de dominio incluyen control en tiempo real, procesamiento de datos empresariales y conmutación de telecomunicaciones.

DSDM

Método de desarrollo de sistema dinámico (Dynamic System Development Method); mencionado como uno de los primeros métodos de desarrollo ágiles.

EJB (Enterprise Java Beans)

Modelo de componentes basado en Java.

entrega (*release*)

Versión de un sistema de software que se pone a disposición de los clientes del sistema.

escenario

Descripción de una forma típica en la que se usa un sistema o en la que un usuario realiza cierta actividad.

etnografía

Técnica de observación que puede usarse en la adquisición y el análisis de requerimientos. El etnógrafo se sumerge en el entorno del usuario y observa sus hábitos laborales cotidianos. A partir de las observaciones es posible inferir requerimientos para soporte de software.

familia de aplicación

Conjunto de programas de aplicación de software que tienen una arquitectura común y una funcionalidad genérica. Éstas se pueden ajustar a las necesidades de clientes específicos al modificar componentes y parámetros del programa.

fiabilidad

Capacidad de un sistema para entregar servicios de acuerdo con las especificaciones. La fiabilidad puede especificarse de manera cuantitativa como una probabilidad de falla a pedido o como la tasa de ocurrencia de fallas.

flujo de trabajo

Definición detallada de un proceso empresarial que tiene la intención de lograr cierta tarea. Por lo general, el flujo de trabajo se expresa gráficamente y muestra las actividades de proceso individual y la información que produce y consume cada actividad.

framework de aplicación

Conjunto de clases concretas y abstractas reutilizables que implementan características comunes a muchas aplicaciones en un dominio (por ejemplo, interfaces de usuario). Las clases en el framework de aplicación se especializan e instancian para crear una aplicación.

fuente abierta

Enfoque al desarrollo de software donde el código fuente de un sistema se hace público y se alienta a usuarios externos a participar en el desarrollo del sistema.

generador de programa

Programa que genera otro programa a partir de una especificación abstracta de alto nivel. El generador incrusta conocimiento que se reutiliza en cada actividad de generación.

gestión de requerimientos

Proceso de administrar los cambios a los requerimientos para asegurarse de que los cambios realizados se analizan adecuadamente y se rastrean a lo largo del sistema.

gestión de versiones

Proceso de gestionar los cambios a un sistema de software y sus componentes, de modo que sea posible conocer cuáles cambios se implementaron en cada versión del componente/sistema, y también para recuperar y recrear versiones anteriores del componente/sistema.

gestión del riesgo

Proceso de identificación de riesgos, valoración de su severidad, planeación de medidas para implementar en caso de que surjan riesgos, y monitorización del software y el proceso de software para detectar riesgos.

gráfica de actividades (PERT)

Gráfica que usan los líderes de proyecto para mostrar las dependencias entre tareas que deben completarse. La gráfica muestra las tareas, el tiempo esperado para completarlas y sus dependencias mutuas. La ruta crítica es la ruta más larga (en términos del tiempo requerido para completar las tareas) a través de la gráfica de actividad. La ruta crítica define el tiempo mínimo requerido para completar el proyecto.

gráfica de barras

Gráfica que utilizan los líderes de proyecto para mostrar las tareas del proyecto, el calendario asociado con dichas tareas y las personas que trabajarán en ellas. Indica las fechas de inicio y fin de las tareas, y las asignaciones de personal, contra un cronograma.

gráfica de Gantt

Nombre alternativo para una gráfica de barras.

herramienta CASE

Una herramienta de software, como un editor de diseño o un depurador de programa, usada para apoyar una actividad en el proceso de desarrollo de software.

ingeniería de sistemas

Proceso que se ocupa de especificar un sistema, integrar sus componentes y probar que el sistema satisface sus requerimientos. La ingeniería de sistemas se ocupa de todo el sistema sociotécnico (software, hardware y procesos operacionales), no sólo del software del sistema.

ingeniería de software basada en componentes (CBSE, *Component-Based Software Engineering*)

Desarrollo de software mediante la composición de componentes de software independientes y portables que son congruentes con un modelo de componentes.

ingeniería de software de cuarto limpio (Cleanroom)

Enfoque al desarrollo de software donde la meta es evitar introducir fallas en el desarrollo de software (por analogía con un cuarto limpio usado en la fabricación de semiconductores). El proceso implica especificación de software formal, transformación estructurada de una especificación a un programa, el desarrollo de argumentos correctos y pruebas estadísticas del programa.

inspección de programa

Revisión donde un grupo de inspectores examina un programa, línea por línea, con la intención de detectar errores. Con frecuencia las inspecciones se realizan con base en una lista de verificación de errores de programación comunes.

interfaz

Especificación de los atributos y las operaciones asociados con un componente de software. La interfaz se usa como medio para acceder a la funcionalidad del componente.

Interfaz de Programa de Aplicación (API)

Una interfaz, por lo general especificada como un conjunto de operaciones que permiten el acceso a la funcionalidad de un programa de aplicación. Esto significa que es posible que esta funcionalidad sea llamada directamente por otros programas y no sólo accederse a ella a través de la interfaz de usuario.

ISO 9000/9001

Conjunto de estándares o normas para procesos de gestión de la calidad definidos por la International Standards Organization (ISO). ISO 9001 es el estándar ISO que resulta más aplicable al desarrollo de software. Puede usarse para certificar los procesos de gestión de calidad en una organización.

ítem de configuración

Unidad legible por máquina, como un documento o un archivo de código fuente, que está sujeto a cambio y donde este último tiene que controlarse mediante un sistema de administración de la configuración.

J2EE

Java 2 Platform Enterprise Edition. Complejo sistema middleware que apoya el desarrollo en Java de aplicaciones Web basadas en componentes. Incluye un modelo de componentes para componentes Java, APIs, servicios, etcétera.

Java

Lenguaje de programación orientado a objetos usado ampliamente, diseñado por Sun con la intención de obtener independencia de plataforma.

lenguaje de consulta estructurado (*Structured Query Language*, SQL)

Lenguaje estándar que se utiliza para programación de bases de datos relacionales.

Lenguaje de Modelado Unificado (*Unified Modeling Language*, UML)

Lenguaje gráfico que se utiliza en el desarrollo orientado a objetos e incluye varios tipos de modelos de sistema que ofrecen diferentes visiones de un sistema. El UML se convirtió en el estándar de facto para el modelado orientado a objetos.

lenguaje de restricción de objetos (*Object Constraint Language*, OCL)

Lenguaje que es parte del UML, que se usa para definir predicados que se aplican a clases de objetos e interacciones en un modelo UML. El uso del OCL para especificar componentes es parte fundamental del desarrollo dirigido por modelo.

leyes de Lehman

Conjunto de hipótesis acerca de los factores que influyen en la evolución de sistemas de software complejos.

línea de productos de software

Véase familia de aplicación.

make

Una de las primeras herramientas de construcción de sistemas; todavía se usa ampliamente en sistemas Unix/Linux.

manifiesto ágil

Conjunto de principios que incluyen las ideas subyacentes de los métodos ágiles de desarrollo de software.

mantenimiento

Proceso de hacer cambios a un sistema después de ponerlo en operación.

mejora de proceso

Cambio en un proceso de desarrollo de software con la intención de hacer que dicho proceso sea más eficiente o mejore la calidad de sus resultados. Por ejemplo, si la intención es reducir el número de defectos en el software entregado, es posible mejorar un proceso al agregar nuevas actividades de validación.

método estructurado

Método de diseño de software que define los modelos de sistema que deben desarrollarse, las reglas y los lineamientos que deben aplicarse a dichos modelos, y un proceso a seguir en el desarrollo del diseño.

métodos ágiles

Métodos de desarrollo de software que se combinan para una entrega rápida del software. El software se desarrolla y se entrega en incrementos, y se minimizan la documentación del proceso y la burocracia. El foco del desarrollo está en al código en sí, y no en los documentos de apoyo.

métodos formales

Métodos de desarrollo de software donde el software se modela usando sentencias matemáticas formales como predicados y conjuntos. La transformación formal convierte este modelo en código. Se usa principalmente en la especificación y el desarrollo de sistemas críticos.

métrica de control

Métrica de software que permite a los administradores tomar decisiones de planeación con base en información acerca del proceso de software o el producto de software que se desarrollará. La mayoría de las métricas de control son métricas de proceso.

métrica de predicción

Métrica de software que se usa como base para realizar predicciones acerca de las características de un sistema de software, como su fiabilidad o mantenibilidad.

métrica del software

Atributo de un sistema o proceso de software que puede expresarse numéricamente y medirse. Las métricas de proceso son atributos del proceso, como el tiempo que tarda en completarse una tarea; las métricas de producto son atributos del software en sí, como el tamaño o la complejidad.

MHC-PMS

Sistema de gestión de pacientes de atención a la salud mental; se usó como estudio de caso en varios capítulos.

middleware

Software de infraestructura en un sistema distribuido. Ayuda a gestionar las interacciones entre las entidades distribuidas en el sistema y las bases de datos del sistema. Ejemplos de middleware son un intermediario de solicitud de objetos y un sistema de gestión de transacciones.

modelado algorítmico de costo

Un enfoque a la estimación de costos del software donde se usa una fórmula para estimar el costo del proyecto. Los parámetros en la fórmula son atributos del proyecto y el software en sí.

modelado de crecimiento de fiabilidad

Desarrollo de un modelo de cómo cambia (mejora) la fiabilidad de un sistema conforme se prueba y se eliminan defectos del programa.

modelo constructivo de costos (*Constructive Cost Modeling*, COCOMO)

Familia de modelos algorítmicos de estimación de costos. COCOMO se propuso por primera vez a principios de la década de 1980 y, desde entonces, se modificó y actualizó para reflejar la nueva tecnología y las cambiantes prácticas en la ingeniería de software.

modelo de componentes

Conjunto de estándares para implementación, documentación y despliegue de componentes. Cubre las interfaces específicas que pueden proporcionar un componente, nomenclatura, interoperación y composición de componentes. Los modelos de componentes brindan la base para que el middleware soporte componentes de ejecución.

modelo de componentes CORBA

Modelo de componentes diseñado para usar en la plataforma CORBA.

modelo de dominio

Definición de abstracciones de dominio, como políticas, procedimientos, objetos, relaciones y eventos. Sirve como base de conocimiento acerca de alguna área problema.

modelo de madurez de proceso

Modelo de la medida en la que un proceso incluye buenas prácticas y capacidades de medición que se integran para mejorar el proceso.

modelo de madurez de capacidad del personal (*People Capability Maturity Model*, P-CMM)

Modelo de madurez de proceso que refleja cuán efectiva es una organización para administrar las habilidades, la capacitación y la experiencia de su personal.

modelo de objeto

Modelo de un sistema de software que se estructura y organiza como un conjunto de clases de objetos y las relaciones entre dichas clases. Pueden existir varias perspectivas diferentes del modelo, como una perspectiva de estado y una de secuencia.

modelo de proceso

Representación abstracta de un proceso. Los modelos de proceso pueden desarrollarse desde varias perspectivas y muestran las actividades implicadas en un proceso, los artefactos usados en éste, las restricciones que se aplican al proceso y los roles de las personas que lo ejecutan.

modelo en cascada

Modelo de proceso de software que comprende etapas de desarrollo discretas: especificación, diseño, implementación, pruebas y mantenimiento. En principio, una etapa debe completarse antes de que sea posible el avance a la siguiente etapa. En la práctica, existe significativa iteración entre etapas.

modelo en espiral

Modelo de un proceso de desarrollo donde el proceso se representa como una espiral; cada vuelta de la espiral incorpora las diferentes etapas del proceso. Conforme uno se mueve de una vuelta de la espiral a otra, se repiten todas las etapas del proceso.

.NET

Marco de trabajo muy extenso que se usa para desarrollar aplicaciones para sistemas Microsoft Windows; incluye un modelo de componentes que define estándares para componentes en sistemas Windows y middleware asociado para apoyar la ejecución de componentes.

Object Management Group (OMG)

Grupo de compañías constituido con la finalidad de desarrollar estándares para el desarrollo orientado a objetos. Los ejemplos de estándares promovidos por el OMG son CORBA, UML y MDA.

ocultamiento de información

Uso de sentencias de lenguaje de programación para ocultar la representación de las estructuras de datos y controlar el acceso externo a dichas estructuras.

patrón arquitectónico (estilo)

Descripción abstracta de una arquitectura de software que se ensayó y puso a prueba en algunos sistemas de software distintos. La descripción del patrón incluye información acerca de dónde es adecuado usar el patrón y la organización de los componentes de la arquitectura.

patrón de diseño

Solución bien probada a un problema común que conjunta experiencia y buena práctica en una forma que pueda reutilizarse. Es una representación abstracta que puede ejemplificarse en varias formas.

plan de calidad

Plan que define los procesos y procedimientos de calidad que deben usarse. Esto implica seleccionar e instanciar los estándares para productos y procesos, y definir los atributos de calidad del sistema que son más importantes.

prevención de fallas

Desarrollo de software en tal forma que no se introduzcan fallas en el desarrollo de dicho software.

probabilidad de falla a pedido (*Probability Of Failure On Demand*, POFOD)

Métrica de fiabilidad que se basa en la probabilidad de que un sistema de software caiga cuando se hace una petición de sus servicios.

proceso de software

Conjunto de actividades y procesos relacionados implicados en el desarrollo y la evolución de un sistema de software.

Proceso Racional Unificado (*Rational Unified Process*, RUP)

Modelo de proceso de software genérico que presenta el desarrollo del software como una actividad iterativa de cuatro fases: concepción, elaboración, construcción y transición. La concepción establece un caso empresarial para el sistema, la elaboración define la arquitectura, la construcción implementa el sistema, y la transición implementa el sistema en el entorno del cliente.

programación en pares

Situación de desarrollo donde los programadores trabajan en pares, y no individualmente, para desarrollar código; es parte fundamental de la programación extrema.

programación extrema (XP)

Método ágil de desarrollo de software usado ampliamente, que incluye prácticas como requerimientos basados en escenarios, desarrollo de primera prueba y programación en pares.

propiedad emergente

Propiedad que sólo se vuelve evidente una vez que se integran todos los componentes para crear el sistema.

protección

Capacidad de un sistema para operar sin falla catastrófica.

pruebas de caja blanca

Enfoque a las pruebas de programa, donde las pruebas se basan en el conocimiento de la estructura del programa y sus componentes. El acceso al código fuente es esencial para las pruebas de caja blanca.

pruebas de caja negra

Un enfoque a las pruebas donde los examinadores no tienen acceso al código fuente de un sistema o sus componentes. Las pruebas se derivan de la especificación del sistema.

Python

Lenguaje de programación con tipos dinámicos, que es particularmente adecuado para el desarrollo de sistemas basados en Web; Google lo usa de manera extensa.

reingeniería

Modificación de un sistema de software para facilitar su comprensión y cambio. Con frecuencia, la reingeniería implica reestructuración y organización de software y datos, simplificación del programa y redocumentación.

reingeniería, procesos empresariales

Cambio de un proceso empresarial para satisfacer un nuevo objetivo organizacional, como costo reducido y ejecución más rápida.

requerimiento funcional

Enunciado de cierta función o característica que debe implementarse en un sistema.

requerimiento no funcional

Enunciado de una restricción o un comportamiento esperado que se aplica a un sistema. Esta restricción puede referirse a las propiedades emergentes del software que se desarrolla o al proceso de desarrollo.

requerimientos de confiabilidad

Requerimiento de sistema que se incluye para ayudar a lograr la confiabilidad requerida para un sistema. Los requerimientos no funcionales de confiabilidad especifican valores de atributo de confiabilidad; los requerimientos de confiabilidad funcional son requerimientos funcionales que especifican cómo evitar, detectar, tolerar o recuperarse de fallas en el desarrollo y la operación del sistema.

REST

REST se deriva de Representational State Transfer (transferencia de estado representacional), que es un estilo de desarrollo basado simplemente en interacción cliente/servidor, y que usa el protocolo HTTP. REST se basa en la idea de un recurso identificable, que tiene una URI. Toda interacción con los recursos se basa en HTTP POST, GET, PUT y DELETE. Ahora se usa ampliamente para implementar servicios Web de carga baja.

riesgo

Resultado indeseable que plantea una amenaza al logro de cierto objetivo. Un riesgo de proceso amenaza la calendarización o el costo de un proceso; un riesgo de producto

es un riesgo que puede significar que algunos de los requerimientos del sistema no se logren.

Ruby

Lenguaje de programación con tipos dinámicos que es particularmente adecuado para programación de aplicaciones Web.

SAP

Compañía alemana que desarrolló un sistema ERP bien conocido y ampliamente usado. También se refiere al nombre del sistema ERP en sí.

Scrum

Método de desarrollo ágil, que se basa en sprints: ciclos de desarrollo cortos. Scrum puede usarse como base para gestión de proyectos ágiles, junto con otros métodos ágiles como XP.

seguridad

Capacidad de un sistema para protegerse a sí mismo contra intrusión accidental o deliberada. La seguridad incluye confidencialidad, integridad y disponibilidad.

SEI

Software Engineering Institute. Centro de investigación y transferencia tecnológica en ingeniería de software, fundado con la intención de mejorar el estándar de la ingeniería de software en las compañías estadounidenses.

servicio

Véase servicio Web.

servicio Web

Componente de software independiente al que puede accederse a través de Internet usando protocolos estándar. Está completamente autocontenido sin dependencias externas. Se han desarrollado estándares basados en XML, como SOAP (Standard Object Access Protocol, protocolo estándar de acceso a objetos), para intercambio de información de servicio Web, y WSDL (Web Service Definition Language, lenguaje de definición de servicio Web), para la definición de interfaces de servicio Web. Sin embargo, el enfoque REST también puede usarse para implementar servicios Web.

servidor

Programa que proporciona servicio a otros programas (clientes).

sistema crítico

Sistema de cómputo cuya falla puede dar por resultado significativas pérdidas económicas, humanas o ambientales.

sistema de planeación de recursos empresariales (*Enterprise Resource Planning*, ERP)

Un sistema de software a gran escala que incluye un rango de capacidades para soportar la operación de las empresas y que ofrece un medio para compartir información a través de dichas capacidades. Por ejemplo, un sistema ERP puede incluir soporte para proporcionar administración, fabricación y distribución en cadena. Los sistemas ERP se configuran con base en los requerimientos de cada compañía que usa el sistema.

sistema de procesamiento de datos

Sistema que se dirige a procesar grandes cantidades de datos estructurados. Dichos sistemas, por lo general, procesan los datos en lotes y siguen un modelo entrada-proceso-salida. Ejemplos de sistemas de procesamiento de datos son los sistemas de boletaje y facturación, y los sistemas de pago.

sistema de procesamiento de lenguaje

Sistema que traduce un lenguaje en otro. Por ejemplo, un compilador es un sistema de procesamiento de lenguaje que traduce el código fuente del programa en código objeto.

sistema de procesamiento de transacciones

Sistema que garantiza que las transacciones se procesen de tal forma que no puedan interferir entre sí y, por lo tanto, que la falla de transacción individual no afecte a otras transacciones o a los datos del sistema.

sistema de tiempo real

Sistema que debe reconocer y procesar eventos externos en "tiempo real". La exactitud del sistema no sólo depende de lo que hace, sino también de qué tan rápido lo hace. Los sistemas de tiempo real por lo general se organizan como un conjunto de procesos secuenciales cooperativos.

sistema distribuido

Sistema de software donde los subsistemas o componentes de software se ejecutan en diferentes procesadores.

sistema heredado

Sistema sociotécnico que es útil o esencial para una organización, pero que se desarrolló usando tecnología o métodos obsoletos. Puesto que los sistemas heredados con frecuencia realizan funciones empresariales críticas, deben mantenerse.

sistema meteorológico a campo abierto

Sistema para recopilar datos acerca de las condiciones meteorológicas en áreas remotas. Se usó como estudio de caso en varios capítulos de este libro.

sistema par a par

Sistema distribuido donde no hay distinción entre clientes y servidores. Las computadoras en el sistema pueden actuar como clientes y como servidores. Las aplicaciones par a par incluyen compartición de archivos, mensajería instantánea y sistemas de apoyo a la cooperación.

sistema sociotécnico

Sistema (incluyendo hardware y componentes de software) con procesos operacionales definidos, que siguen operadores humanos y que funciona dentro de una organización. Por lo tanto, recibe influencia de políticas, procedimientos y estructuras organizacionales.

sistemas basados en eventos

Sistemas donde el control de la operación está determinado por eventos que se generan en el entorno del sistema. La mayoría de los sistemas de tiempo real son sistemas basados en eventos.

sistemas embebidos

Sistema de software que se embebe en un dispositivo de hardware (por ejemplo, el sistema de software en un teléfono celular). Por lo general, los sistemas embebidos son sistemas de tiempo real y, por lo tanto, deben responder en forma oportuna a los eventos que ocurren en su entorno.

Subversión

Herramienta de construcción de sistemas de fuente abierta, ampliamente utilizada, que está disponible para una variedad de plataformas.

tasa de ocurrencia de fallas (*rate of ocurrence of failure*, ROCOF)

Métrica de fiabilidad que se basa en el número de fallas observadas de un sistema en un periodo de tiempo dado.

tejedor de aspectos (*weaver*)

Un programa que por lo general es parte de un sistema de compilación que procesa un programa orientado a aspectos y modifica el código para incluir los aspectos definidos en los puntos especificados del programa.

tiempo medio para falla (MTTF)

Tiempo promedio entre fallas de sistema observadas; se usa en especificación de fiabilidad.

tipo de datos abstractos

Un tipo que se define por sus operaciones y no por su representación. La representación es privada y sólo puede accederse a ella mediante las operaciones definidas.

tolerancia a fallas

Capacidad de un sistema para continuar en ejecución incluso después de que ocurran fallas.

transacción

Unidad de interacción con un sistema de cómputo. Las transacciones son independientes y atómicas (no se descomponen en unidades más pequeñas) y son una unidad fundamental de recuperación, consistencia y concurrencia.

validación

Proceso de comprobar que un sistema satisface las necesidades y expectativas del cliente.

verificación

Proceso de comprobación de que un sistema satisface sus especificaciones.

verificación de modelo

Método de verificación estático donde un modelo de estado de un sistema se analiza exhaustivamente con la intención de descubrir estados inalcanzables.

workbench CASE

Conjunto integrado de herramientas CASE que trabajan en conjunto para apoyar una actividad de proceso principal como el diseño de software o la administración de la configuración.

WSDL

Notación basada en XML para definir la interfaz de servicios Web.

XML

Extended Markup Language, es decir, lenguaje de marcas extensible. XML es un lenguaje de marca de texto que soporta el intercambio de datos estructurados. Cada campo de datos está delimitado por etiquetas que ofrecen información acerca de dicho campo. Ahora XML se usa ampliamente y se ha convertido en la base de protocolos para servicios Web.

XP

Abreviatura utilizada comúnmente para Programación Extrema.

Z

Lenguaje de especificación formal, basado en modelos, desarrollado en la Universidad de Oxford, en Inglaterra.

Índice analítico

A

Abrazar el cambio (manifiesto ágil), 60
Accidentes, 301, 302
Acciones de usuario, bitácora de, 382-83
Aceptabilidad, 8, 24, 315, 316
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices (Fuerza de Trabajo Conjunta sobre Ética y Prácticas Profesionales de la Ingeniería de Software), 15-16
Actividades
  automatizadas, 37
  de ingeniería de software, 6, 9, 28, 36
Activos, 303, 304
Actuadores, 198, 300, 349, 350, 491, 540, 541, 545
Adaptación ambiental, 243-44
Adaptadores, 468-71, 476
ADL (lenguajes de descripción de arquitectura), 154
Administración
  de la configuración, 193, 195-96, 202, 681-704. *Véase también* Administración del cambio
    actividades de, 195-96
    construcción de sistema y, 682, 684, 693-99, 702
    flujo de trabajo para, 52
    gestión de entregas (*release*) y, 682, 699-701, 702
    gestión de versiones y, 195, 682, 690-93, 702
    patrones arquitectónicos y, 155
  de personal, 595, 602-7
  de proyectos, 593-617
    actividades de, 595
    ágil, 72-74
    flujo de trabajo, 52
  del cambio, 77, 196, 348, 682, 685-89, 702
    forma de solicitud (CRF), 686
    métodos ágiles y, 77
    registro de historia, 691
    requerimientos para administración del cambio, 113-14
    terminología, 684
Adquisición
  análisis (de requerimientos), 37, 100-110, 115
  de datos de flujo de neutrones, 554
Advice (consejo) (AOSE), 572, 587
Aeronave, falla de sistemas, 294
Aglomeración de datos, 251
Agregación, 133, 186
Airbus 340, sistema de control de vuelo, 345, 351-52
AJAX, 14, 26, 433, 451, 501, 507, 634
Ajuste del cambio (procesos de cambio de proceso), 720
Alias, 360
Ambientes laborales, 613
Ambigüedad, mediciones y, 676-77, 713
Amenazas, 303, 304
AMI (Análisis, Medición, Mejora), Método, 713
Análisis
  basado en escenarios, 183
  de árbol de fallas, 317-19, 336
  de componentes
    de software, 673-75
    desarrollo basado en reutilización, 35

Análisis (*continúa*)
de red de Petri, 317
de sistemas supervivientes, 387, 388-90
de temporización (sistemas de tiempo real), 554-57
estático, 395-400, 417
automático, 398-400
estructurado de DeMarco, 134
flujo de trabajo de diseño y, 52
Medición, Mejora (AMI), Método, 713
preliminar de riesgos, 312, 330
Analizadores estáticos, 197, 218, 334, 395, 398-400
Antibloqueo, sistema de frenado, 552
AOSE. *Véase* Ingeniería de software orientada a aspectos.
Aplicaciones
independientes, 10-11, 36
interactivas basadas en transacciones, 11
Aprendizaje (análisis de proceso), 716
Apuntadores, 359-60
Argumento(s)
de protección estructurados, 414-17, 418
de seguridad informal (bomba de insulina), 416-17
estructurados, 411-13, 418
Ariane 5, explosión, 344, 394, 445, 467, 468
Armazones
de aplicación, 431-34, 448
empresarial, 432
Arquitectura(s)
cliente-servidor
de dos niveles, 492-93
multinivel, 493-95
de automonitorización, 350-52
de compilador de tubería y filtro, 170, 171
de componentes distribuidos, 495-98
de protección en capas, 378
de referencia, 171
de sistema, 348-55
confiable, 348-55
de aplicación, 164-71, 172
de sistema de recolección de datos (estación meteorológica), 182, 183
compilador de tubería y filtro, 170-71
definición de, 172
distribuidas, 151, 163
ingeniería de sistemas y, 275
referencia, 171
de software
aplicación, 164-71, 172
arquitectura en grande, 148
arquitectura en pequeño, 148
de sistema, 348-55
de sistema de administración de recursos, 437
dirigida por modelo (MDA), 138-41
distribuidas, 151, 163
maestro-esclavo, 490-91
orientadas a servicios (SOA), 498, 508-36
definición de, 534
reutilización de software y, 430
SaaS *versus*, 502
p2p descentralizada, 500
P2P semicentralizada, 501
par a par (p2p), 498-501
replicadas, 348
Arreglos
no enlazados, 360
sin límites, 360
Aseguramiento de confiabilidad/seguridad, 393-421. *Véase también* Procesos de software (aseguramiento de)
Asignación de memoria dinámica, 360
AspectJ, lenguaje de programación, 566, 571, 573, 574, 575, 584
Aspectos, 571-87
Ataque(s), 303, 304, 306, 383, 405, 483
de envenenamiento SQL, 383, 405
de modificación, 483
de negación de servicio, 376
detección y neutralización, 305
evaluación (requerimientos de seguridad), 331
internos, 369
Atención a la salud mental-sistema de gestión de pacientes. *Véase* MHC-PMS
ATM (cajeros automáticos), 166, 167, 324, 326, 327, 493
Atributos de software, 6, 8, 24
Automatización de pruebas, 42, 70, 212, 231, 444, 695

B

"Banda de Cuatro", 190, 191, 192
BMPN (notación para el modelado de procesos de negocios), 530, 351, 532, 535, 356, 717, 718, 730, 731
Bomba
de gas (modelo de máquina de estado), 545
de petróleo (modelo de máquina de estado), 545
Buffer circular, 544
Bugzilla, 196, 689

C

Caída del sistema, 297
Caja blanca, 585, 586
  en XP, 69
Cajeros automáticos. *Véase* ATM
Calendarización (planeación de proyecto), 626-30
Calidad del servicio (QoS), 484, 504
Cambio, 43-45. *Véase también* Procesos de software (cambio de)
  métodos ágiles y, 60, 65, 114
  resistencia al, 720
  social, 10
    empresa y, 10
  XP y, 67
Capa
  de gestión y comunicación de datos, 264-65
  social, 264-65
Capacidades de reinicio, 361
CASE (ingeniería de software auxiliada por computadora), herramientas, 37, 58, 174, 597, 602
Caso
  de protección/confiabilidad, 410-17, 418
  de prueba
    de verificación de dosis, 69, 70
    diseño, 216
    generación, 111
  de uso, 106-8, 142, 180
    adquisición, 108
    modelado de, 100, 120, 124-26, 142, 180, 181
    pruebas, 219
Catálogo de arquitectura de software, de Booch, 150, 173, 174
CBSE. *Véase* Ingeniería de software basada en componentes
Ciclo de vida
  análisis de riesgo, 312, 330
    seguridad, 347
    software, 30-32
  del software, 30-32. *Véase también* Modelo en cascada
CIM (modelos de computación independientes), 140, 141
Clases de objetos, 182-84, 189
Clearcase, 196, 204
Clientes en sitio (XP), 66
CMM, modelo de madurez. *Véase* Modelo de Madurez de Capacidad del Personal; Modelo de Madurez CMM de Software
CMMI, armazón de mejora del proceso, 721-28, 729
COCOMO II, modelo, 248, 464, 637-45
Code Red, gusano, 305
Código(s)
  de conducta, 15-17, 24
  de Ética y Práctica Profesional (ingeniería de software), 15-16
  duplicado, 68, 251
  glue, 468, 469, 476
Compartimentalización, 384
Competencia(s), 14
  separación de, 567-71
Compiladores, 166
Complejidad ciclomática, 247, 668, 669, 670, 672, 673, 679
Componente(s)
  Controlador (MVC), 155
  de software, 35-36, 455-58, 475
    composición, 468-75, 475
    comunicaciones, 152, 198, 453, 460, 482, 505
    diseño, 40
    interfaces, 188-89, 201, 216-18
    modelos, 458-61, 475
    pruebas, 216-19
    reutilización de, 35-36, 194, 426
    servicios *versus*, 514-18
  de visualizador, 497
  Model (MVC), 155
  View (MVC), 155
Comprobación(es)
  basada en herramientas, 405-6
  de aserción, 400
  de error definido por el usuario, 400
  de modelo, 334, 339, 395, 396, 397-98, 417
  de racionalidad, 357
  de rango, 357
  de realismo, 110
  de representación, 357
  de tamaño, 357
  de totalidad, 86, 87, 110
  de validez, 110, 356-57, 383
Computación en nube, 11, 13, 484, 514
Comunicaciones
  de regreso (*callbacks*), 433
  proceso de análisis, 716
Condiciones latentes, 283
Confiabilidad del software, 289-95
  aseguramiento, 393-421
  casos de, 410-17, 418
  de hardware, 270

Confiabilidad del software (*continúa*)
definición de, 291, 306
especificación de, 309-40
ingeniería y, 341-65
propiedades y, 291-94
seguridad y, 8, 12, 24, 292
Confianza, 8, 10, 291-92
Confidencialidad, 14
Configuración de tiempo de implementación, 439-40
Consistencia, 86, 87, 110, 603
Constantes, mención de, 363
Construcción de sistemas, 682, 684, 693-99, 702
Constructos proclives a error, 359-61, 364
Control(es)
armazones de aplicación y, 434
basados en eventos, 164
centralizado, 164
de configuración, 684
patrones arquitectónicos para, 164
seguridad y, 303, 304
CORBA (arquitectura común de intermediarios de peticiones de objetos), 454, 478, 482, 483, 496, 507
Corrección de bugs, 257
Costos. *Véase también* Técnicas basadas en experiencia (técnicas de estimación)
COCOMO II y, 642
confiabilidad y, 294-95
eliminación de fallas en el desarrollo, 343
especificación formal, 336
falla del sistema, 290
generales, 620
ingeniería de software, 6
mantenimiento/desarrollo, 244, 257
reutilización de software y, 448
COTS
reutilización basada en, 36, 440-48
sistemas, 35, 177, 276-77
de solución, 442-44
integrados, 445-48
Creación
de prototipos
de sistema, 44, 45-46, 53, 109, 111
desechables, 46
de sistema ejecutable, 695
"Crisis del software", 5
Criterios de éxito (sistemas sociotécnicos), 268, 272-73
Crystal, 59, 78, 80
Cuestionarios, 714, 715
CVS, 691, 692, 704

D

Daños, 301
Demandas, fallas de software y, 4
Depuración, 41, 211, 223
Derechos de propiedad intelectual, 14
Desafío de heterogeneidad, 10
Desarrollo
anfitrión destino, 193, 196-98, 202
costos de mantenimiento, 244,257
de código abierto, 198-201, 202
de instancia de producto, 438
de primera prueba, 66, 68, 69-70, 223, 231
de sistemas, 273-74, 278-81, 286
de software
adaptativo, 59, 80
brownfield, 75, 235
profesional, 5-14, 24
dirigido por modelo (MDD), 40
dirigido por plan
métodos ágiles *versus*, 62-64, 77, 623
planeación de proyectos y, 623-26
procesos de, 29, 30, 42, 43, 62-64, 77
dirigido por pruebas (TDD), 221-24
/entrega iterativos, 45, 52, 53
evolución *versus*, 43, 235-36, 257
incremental, 30, 32-34, 53, 58, 67, 69
independiente, 691
mantenimiento *versus*, 43
modelo en espiral y, 49, 235-36, 257
puesta a prueba, 41-42, 210-21
rápido de software, 57-58
reutilización y, 35
con (proceso CBSE), 461, 465-68
para (proceso CBSE), 461, 462-65
servicios y, 527-34
sistemas sociotécnicos, 273-74, 278-81, 286
visión, 154, 172
Descripción Universal, Descubrimiento e Integración (UDDI), 510, 511
Despliegue
del sistema, 279, 281
seguro, 390
DFD (diagramas de flujo de datos), 134
Diagramas
de actividades (UML), 19, 29, 120, 135, 143, 718
de bloques, 150, 179
de clase, 120, 129-31
de estado(UML), 120, 135, 136, 143, 186, 187, 188, 203

de flujo de datos (DFD), 134
de secuencia, 120, 124, 126-28, 186
Dinámica de evolución de programa, 240-42
Diseño
arquitectónico, 39, 147-75
catálogo arquitectónico de Booch y, 150, 173, 174
decisiones, 151-53, 172
diseño orientado a objetos y, 181-82
modelo de visión 4+1, 120, 145, 153, 175
seguridad y, 152, 376-80
de base de datos, 40
de interfaz, 39, 188, 189
de catálogo, 523
de servicio, 521-24
de usuario, 45, 274, 275, 312
de seguridad de sistema, 375-86
de sistemas, 279
embebidos, 540-47
seguros, 375-86, 390
de software, 24, 28, 38-41, 53, 176-78
fase de ciclo de vida, 31
flujo de trabajo, 52, 529-33
implementación y, 38-41, 53, 176-78, 193-98, 201
interfaz de usuario, 45, 274, 275, 312
orientado a aspectos, 580-84
para implementación, 384, 385-86
para recuperabilidad, 384-85
patrones de, 189-93
/requerimientos, modelo en espiral de, 279-80
reutilización y, 35
sistemas embebidos, 540-47
y programación orientados a aspectos, 580-84
Dispersión, 569-70, 587
Disponibilidad
de sistema, 152, 292, 295-99
Diversidad
del software, 344, 345, 349, 353-55, 363
redundancia y, 343-44, 363, 383
software, 344, 345, 349, 353-55, 363
Documentación, 5, 24, 246
arquitecturas y, 154-55
capítulo en línea, 246
característica de proceso y, 346
del sistema. *Véase* Documentación
estándares, 658
métodos ágiles y, 61, 63-64, 155
sistema build y, 695
TDD y, 223
Dominio, 103, 214
DSDM, 57, 59, 81
Duración del proyecto y asignación de personal (COCOMO II), 645-46

E

Eficiencia, 8, 12, 24
EJB (Enterprise Java Beans), 432, 454, 455, 458, 496
Ejemplo
OilSoft, 622-23
PharmaSoft, 621-22
Empresas
cambio social y, 10
desarrollo de software rápido y, 57-58
modelado de flujo de trabajo, 52
reingeniería de procesos, 721
reorganización, 276
sistemas de software, 13, 28, 29, 54, 448, 505
software de código abierto, 201
Web y, 13
Enfoque
"big bang", 280
de persona (errores humanos), 282
de sistemas (errores humanos), 283
dirigido por riesgo para derivación de requerimientos, 336
SPICE, 722
Enlaces de arreglo, verificación, 362
Enredos (*tangling*), 569-70, 587
Enrollado
aplicación, 447
sistema heredado, 250, 430, 464, 526
Enterprise Java Beans (EJB), 432, 454, 455, 458, 496
Entorno(s). *Véase también* IDE
de desarrollo interactivo. *Véase* IDE
ECLIPSE, 37, 68, 80, 197, 204, 251, 523
flujos de trabajo de, 52
herramientas CASE y, 37, 58, 174, 597, 602
patrones arquitectónicos y, 156
trabajo y, 613
Entrada(s)
procesamiento de, por omisión, 360-61
verificación de validez de, 356-57, 383
Entrega(s)
de versión pequeñas (XP), 66
incremental, 34, 44, 47-48, 60
(*release*), 684
Entrevistas, 104-5, 715

Enunciados go-to, 359, 364, 585
Envoltura (capa) de aplicación (*wrapper*), 447
Equipos activos, 405, 418
Ergonomía, ingeniería de sistemas y, 275
Error(es)
  aritmético, 320
  de operadores, 263, 270, 282, 291, 312, 321
  de temporización, 217-18
  del sistema, 297
  humanos, 282-84, 286, 297
Escalamiento de métodos ágiles, 74-77, 78
Escenarios, 105-6
  pruebas de, 225-26
Escritura de propuestas, 595
Espacio de trabajo, 684
Especialización
  ambiental (líneas de productos de software), 436
  de plataforma (líneas de producto de software), 436
  funcional (líneas de producto de software), 436
Especificación(es)
  algebraicas, 334
  cuantitativas de la fiabilidad, 324-26
  de confiabilidad funcional, 328
  de interfaz, 188-89
  de requerimientos de software (SRS), 91-94, 115
  de sistema (V-spec), 354
  de software, 6, 9, 12, 24, 28, 36-38, 309-40
  en lenguaje natural, 94, 96-97
  estructuradas, 97-98
  formales, 95, 333-36, 337
  matemáticas, 95. *Véase también* Métodos formales.
Espíritu de equipo (estudio de caso), 608
Establecimiento de objetivos (modelo en espiral), 49
Estaciones meteorológicas a campo abierto, 22-23
  ambiente de, 22-23
  arquitectura
    de alto nivel de, 182
    de sistema de recolección de datos en, 182, 183
  diagrama de estado, 187, 188
  gráfica de secuencia de "datos meteorológicos recopilados" para, 220
  identificación de objetos en, 183-84
  interfaces, 189
  interfaz de objeto de, 212
  modelo
    de caso de uso para, 180, 181
    de contexto para, 180
  objetos, 184
  recolección de datos (diagrama de secuencia) en, 186
Estándares
  de proceso, 29, 346, 658, 659, 709
  de producto, 658, 659
  de servicio Web, 36, 386, 426, 461, 483, 510
  de software, 657-63
  documentación, 658
  marco de estándares ISO 92001, 660-63
Estímulo(s)
  no periódicos, 540-41
  periódicos, 540-41
  /respuesta (sistemas embebidos), 540-41
Estrategia
  de minimización (gestión del riesgo), 601
  de reconocimiento (supervivencia), 387, 389
  de recuperación (supervivencia), 387-88, 389
  de resistencia (supervivencia), 387, 389
Estudios de factibilidad, 37, 100
Etapa
  de adquisición/procuración (sistemas sociotécnicos), 273-74, 275-77, 286
  de operación (sistemas sociotécnicos), 273-74, 281-85, 286
  de procuración/adquisición (sistemas sociotécnicos), 273-74, 275-77, 286
Etnografía, 108-9, 715
Evaluación, 256
Evitación
  de cambio, 44
  de riesgos, 301-2, 319
  estrategias (gestión del riesgo), 601
  fallas de desarrollo, 299, 342
  punto de falla único, 381-82
  vulnerabilidad, 305
Evolución del software, 6, 9, 24, 28, 43, 234-60
  definición de, 53
  desarrollo *versus*, 43, 235-36, 257
  diagrama, 44
  dinámica de evolución de programa, 240-42
  evolución de sistema *versus*, 284-85
  mantenimiento y, 8, 24, 43
  modelo en espiral de, 235-36, 257
  prestación de servicios y, 236-37
  procesos de, 237-40
  refactorización y, 44, 66, 71, 250-52
Excepciones
  de proceso, 718
  manejador para, 357-59
Exposición, 303, 304
Extensiones
  de la calidad del servicio, 576
  sistemas centrales con, 576-77

F

Fallas
  de hardware, 290
  de sistema
    computadora, 291
    confiablidad/seguridad y, 8, 274, 290
    costos de, 290
    definición, 297
    disponibilidad y, 295-97
    errores del sistema y, 298
    errores humanos y, 281, 282-84, 286
    fallas de software y, 265
    fallas en el desarrollo del sistema y, 299
    fiabilidad y, 271, 295-97
    no determinismo y, 272
    pruebas de rendimiento y, 227
    reparabilidad y, 293
    seguridad en caso de, 382
    sistemas críticos para la protección y, 300
    sistemas críticos y, 291
    sistemas de aeronaves y, 294
    tipos de 322
  de software, 4, 12, 48, 265-66, 270, 290, 291, 294, 297, 300, 320, 321, 327, 333, 349, 411, 467, 480
  en el desarrollo, 297
    detección y eliminación de, 299, 342
    evitación de, 299, 342
    reparación de, 243-44
    tolerancia a, 299, 342
  errores humanos y, 282-84, 286, 297
  operacionales, 263, 270, 282, 291, 312, 321
Fase
  de concepción (RUP), 50-51
  de elaboración (RUP), 51
  de transición (RUP), 51
Fiabilidad
  del operador, 270
  del sistema, 4, 5, 8, 269, 270-71, 295-99, 306
  especificación de, 320-28
  métrica, 322-24, 336
  modelado de crecimiento, 402, 403
  pruebas de, 401-4
  requerimientos de, 324-28, 336
Fijación de precio al software (planeación de proyectos), 621-23
Filmoteca, 162
Flujo(s)
  de trabajo, 51-52, 529-33
    paquete de vacaciones, 528
Formato de petición de cambio (CRF), 686
Fototeca, 472-74
Framework de aplicación Web (WAF), 433
Fronteras
  del sistema, 121, 122, 142, 179, 296
Fusión, 684, 693

G

Generación de rutinas (*scripts*) de construcción, 695
Generadores de programa, 430
Generalidad especulativa, 251
Generalización, 131-33, 185-6, 190, 201
Gestión
  calidad del software y, 655-57
  de almacenamiento, 691
  de entregas (*release*), 682, 699-701, 702
  de la calidad, 651-80
  de versiones (VM), 195, 682, 690-93, 702
  del software, 12, 24, 591-92. *Véase también* Administración de la configuración; Administración de personal; Administración de proyectos; Gestión de la calidad; Mejora de procesos; Planeación de proyectos
  estándares de software y, 657-63
  medición/métricas de software y, 668-77
Gnutella, 499
GQM (Meta-Pregunta-Métrica), 674, 712-13, 730
Gráfica(s)
  de actividades (planeación), 627, 630
  de asignación de personal, 631
  de estado, 135, 545
Grupo(s), 607-14
  composición (estudio de caso), 610
  comunicaciones, 613-14
  de Gestión de Objetos (OMG), 138, 139
  organización de, 610-13
Gusano, 305, 308, 329, 383, 392, 404
  de Internet, 305, 308, 383, 392

H

Herencia, 132, 184, 185, 186, 190, 203, 212, 360, 364, 432, 435, 670, 674

Herramientas
de desarrollo de software. *Véase* CASE, herramientas
de ingeniería de software auxiliada por computadora. *Véase* CASE, herramientas
de traducción, 37, 140, 141, 249
automatizadas, 140
Historia
de derivación, 689
de usuario, 65, 68, 91, 230, 239, 632
Honestidad (administración de personal), 603
Horno de microondas, 11, 136-38, 538, 545

I

ICASE, 639
IDE (Entornos de Desarrollo Interactivos)
arquitectura de repositorio para, 159, 160
definición de, 37, 197
desarrollo anfitrión destino y, 193, 196-98, 197, 202
entorno ECLIPSE y, 37, 68, 80, 197, 204, 251, 523
propósito general, 197
Identificación
de amenaza (requerimientos de seguridad), 331
de control (requerimientos de seguridad), 331
de servicio candidato, 518-21
y evaluación de activos (requerimientos de seguridad), 331
y priorización de mejora (proceso de cambio de proceso), 719
Identificadores Universales de Recurso (URI), 512
IEC (International Electrotechnical Commission), estándar para gestión de seguridad, 313
IEC 61508, ciclo de vida de seguridad, 347
Implementación
conflictos de, 193-98
de cambio, 238, 239
de servicio, 524-25
del sistema, 10, 12, 24, 28, 38-41, 53, 193-98, 279, 281
despliegue de servicio, 524-25
diagramas de implementación UML, 129, 197, 198
diseño e, 38-41, 53, 176-78, 193-98, 201
diseño para, 384, 385-86
fase de ciclo de vida, 31
flujo de trabajo, 52, 529-33
segura, 390
Incompatibilidad
de operación, 469
de parámetro, 469
Infracción del copyright, 14, 158, 450, 501
Ingeniería
de requerimientos, 6, 9, 12, 24, 28, 36-38, 53, 82-117
definición de, 53
métodos ágiles y, 63
orientada a competencias, 577-80
de sistemas, 6, 9, 266, 273-75
críticos, 60, 336, 347, 348, 407, 419
de software
actividades de, 6, 9, 28, 36
análisis de procesos e, 716
avanzada, 423-24
basada en componentes (CBSE), 189, 452-78
beneficios de la, 566, 587
ciencias de la computación *versus*, 6, 9
con aspectos, 576-87
costos de, 6
definición de, 6, 7-8
distribuida, 479-507
diversidad de, 10-12
historia de la, 5
importancia de la, 8-9
ingeniería de sistemas *versus*, 6, 9, 266, 274
licenciamiento y, 408
MHC-PMS y, 571-75
nociones fundamentales en, 12, 24, 28
orientada a aspectos (AOSE), 430, 565-89
orientada a reutilización, 30, 35-36, 53, 426-28, 453
responsabilidad ética y, 14-17, 24
retos para la, 4, 6, 10
separación de competencias y, 567-71
terminología, 572
Web y, 6, 13-14
de subsistemas, 279, 280
dirigida por modelo (MDE), 138-42, 143, 430
inversa, 249, 250
social, 369, 383, 391
Inspecciones, 208-9, 218, 585, 663-68. *Véase también* Revisiones
Instancia de versión, 684
Integración
continua, 65, 66, 68, 76, 78, 697, 698
del sistema, 195, 279
incremental, prueba e, 219
reutilización e, 35
Interacción de usuario (WAF), 433
Intercepción, 483
Interfaces
componentes de, 188-89, 201, 216-18
de memoria compartida, 217

de parámetro, 216
de servicio, 534
de transmisión de mensajes, 217
procedimentales, 217
servicio e, 534
International Electrotechnical Commision (IEC), estándar para gestión de seguridad, 313
Interpretación de mediciones, 676-77, 713
Interrupciones, 360, 362, 483
Introspección (análisis de proceso), 716
Inversión de control, en armazones, 434
Involucramiento del cliente (métodos ágiles), 60, 62, 65, 239, 633
ISO 92001, marcos de estándares, 660-63
Ítem
de configuración, 684
de software (SCI), 684

J

J2EE, 36, 141, 454
Java
desarrollo de sistemas
de tiempo real y, 547
embebidos y, 546
prueba de programa, 222
Jerarquía
de afirmación de protección (sistema de control de bomba de insulina), 414
de necesidades humanas, 604
Juego de planeación, 632-33
JUnit, 42, 55, 70, 81, 197, 212, 222, 232, 444, 695

L

Lenguaje
de Consulta Estructurado. *Véase* SQL
de Definición de Servicio Web. *Véase* ESDL
de descripción
de arquitectura (ADL), 154
de diseño, 95
de Modelado Unificado. *Véase* UML
de restricción de objeto (OCL), 142, 189, 472-73
natural estructurado, 95
SPARK/Ada, 336
Leyes de Lehman, 240-42, 257
Librerías de programa, 430
LIBSYS, 158-59
Licencia
BSD (Berkeley Standard Distribution), 200
de Público General (GPL) GNU, 200
Pública General Reducida (Lesser General Public License), 200
Licenciamiento
código abierto, 200-201
ingenieros de software y, 408
Limitación
de daño, 302, 319
y recuperación de exposición, 305
Lineamientos
contratación, 611
de programación confiable, 355-63
seguridad de sistema, 380-85
Línea(s)
base (*baseline*), 684, 690
de código (*codeline*), 684, 690
de producto de software, 434-40, 448
principal (*mainline*), 684
Linux, 199, 200, 204, 367, 383, 392, 436, 558, 563
Lista(s) de verificación
inspección de, 667
revisiones (análisis de riesgo), 317
seguridad de, 406
Lógica formal (análisis de riesgos), 317
Logs (bitácoras), 382-83, 409-10
Llamadas a procedimiento remoto (RPC), 486, 498
Lluvia de ideas, 314, 598

M

MAD. *Véase* Arquitectura dirigida por modelo
Mal uso
de computadoras, 14
de interfaz, 217
Mala interpretación de interfaz, 217
"Malos olores", 251
Malware, 14, 290, 501
Manejador de excepciones, 357-59
Manifiesto ágil, 56, 59, 60, 141, 688, 708
Mantenibilidad, 8, 18, 24, 90, 149, 152, 178, 209, 244, 245, 247, 293, 709

Mantenimiento
de software, 8, 24, 43, 242-52
costos de, desarrollo, 244, 257
desarrollo *versus*, 43
diseño arquitectónico y, 152-53
distribución de esfuerzo, 244
fase de ciclo de vida, 31
métodos ágiles y, 61-62
predicción, 246-48
tipos de, 243, 257
preventivo. *Véase* Refactorización
Mapeo entrada/salida, 298
Marca de tiempo (*timestamp*) de modificación, 696
Marco .NET, 36, 141, 430, 432, 454, 458, 459, 466, 496, 584
Marcos
de aplicación Web (WAF), 432, 433
de infraestructura del sistema, 432
MDD. *Véase* Desarrollo dirigido por modelo
MDE. *Véase* Ingeniería dirigida por modelo. *Véase también* Métricas
ambigüedad en, 676-77, 713
controlador/predicción, 669
interpretación, 676-77, 713
medición de proceso, 711-14, 729
Medición/métricas de software, 668-77
Mejora. *Véase* Procesos de software (mejora de)
de estructura de programa, 249
Mensajes de entrada/salida (UML), 524
Meta-Pregunta-Métrica (GQM), 674, 712-13, 730
Método(s)
ágiles, 29, 56-81
administración del proyecto, 72-74
B, 32, 55, 335, 396
cambio y, 60, 65, 114
desarrollo incremental y, 33, 58
diseño arquitectónico y, 148
documentación y, 61, 63-64, 155
enfoque dirigido por plan *versus*, 62-64, 77, 623
enfoque Scrum y, 56, 57, 59, 72-74, 78, 631, 632
escalamiento, 74-77, 78
especificación formal y, 336
evolución y, 239
involucramiento del cliente y, 60, 62, 65, 239, 633
MAD y, 141
manifiesto, 56, 59, 60, 141, 688, 708
mejora del proceso y, 706, 728
panorama, 58-62
"personas, no procesos" y, 60, 65, 688
planeación del proyecto, 631-33
principios de, 60
requerimientos y, 84
resultados de diseño y, 40
sistemas críticos y, 60, 348
de análisis estructurado, 100, 134
de diseño estructurado, 40, 178
de ingeniería de requerimientos VOLERE, 97, 115
formales (desarrollo de software), 32, 49, 95, 139, 333, 337, 396
MDE y, 139
método B, 32, 55, 335, 396
modelos de sistema, 334
verificación y, 395-97
Objectory, 106
Métrica(s)
AVAIL, 332-34, 336
de control/de predicción, 669
de disponibilidad (AVAIL), 332-34, 336
de eventos, 711-12, 729
de productos, 672-73
de tiempo, 711-12, 729
de utilización de recursos, 711-12, 729
enfoque GQM, 674, 712-13, 730
estáticas de producto de software, 673
medición de software y, 668-77
para fiabilidad, 322-24, 336
para requerimientos no funcionales, 90
tiempo, 711-12, 729
utilización de recursos, 711-12, 729
MHC-PMS (Mental Health Care-Patient Management System: Atención a la Salud Mental-Sistema de Gestión de Pacientes), 20-22
análisis
de activos (reporte de valoración de riesgos) para, 332-33
de amenazas y control para, 333
AOSE y, 571-75
asociación de agregación en, 133
características principales del, 21
caso de prueba de comprobación de dosis, 69, 70
casos de uso para, 107
conceptos de seguridad y, 303-4
criterios de éxito y, 272-73
diagramas de clase y, 130, 131
diagramas de secuencia y, 126-28
escenario en, 106
jerarquía de generalización y, 132, 33
metas de, 20-21, 272
modelado de caso de uso y, 124, 125, 126
modelo
de contexto de, 122
de proceso de detención involuntaria en, 123

organización (diagrama) de, 20
participantes para, 103
patrón arquitectónico en capas en, 158, 167-69
privacidad y, 21-22
prueba de escenario y, 226
pruebas basadas en requerimientos y, 225
requerimientos
funcionales en, 83, 84, 85, 86
no funcionales en, 89
seguridad y, 21-22
tarjetas
de historia y, 66, 67
de tarea y, 68, 69
Middleware, 196, 264, 432, 487-88
Minería de datos, 497
Modelado
ágil, 59, 80, 120, 145
algorítmico De Costo, 634, 635-37
arquitectónico, 40, 122, 129
de crecimiento, confiabilidad, 402-3
de sistema. *Véase* Modelos
dirigido
por datos, 134-35
por eventos, 135-38
Modelo(s), 11, 95, 118-46
actividad, 19, 29, 120, 135, 143, 718
caso de uso, 100, 120, 124-26, 142, 180, 181
cliente liviano, 492
cliente pesado, 492
CMMI
continuo, 727-28
en etapas, 725-27
contextuales, 121-24, 142, 179-81
crecimiento de fiabilidad, 402, 403
datos semánticos, 93, 130, 145
de Boehm. *Véase* Modelos en espiral
de Capacidad de Ingeniería de Sistemas, 722, 731
de componentes, 458-61
de comportamiento, 133-38, 143
de composición de aplicación, 639
de computación independientes (CIM), 140, 141
de diseño temprano, 639-40
de interacción, 124-28, 179-81, 485-86
de Madurez CMM de Software, 644, 683, 721, 722, 727
de Madurez de Capacidad del Personal (P-CMM), 606, 722
de máquinas de estado, 186, 187, 201, 397, 398, 545
de proceso, 29-36, 53, 718, 729
de negocios, 534
de prueba, 210
de queso suizo, 283-84, 286
de reutilización (COCOMO II), 640-42
de subsistema, 185, 186
de transmisión, 182
de visión 4+1, 120, 145, 153, 175
dinámico, 129, 179, 185, 186, 201
dirigido
por datos, 134-35
por eventos, 135-38
en cascada, 29, 30-32, 53
en espiral, 48-50, 99-100, 235-36, 257, 279-80
específicos de plataforma (PSM), 140, 141
estáticos, 129, 186, 201
estructurales, 129-33, 143, 185
generalización, 131-33, 201
gráficos. *Véase* Modelos
independientes de plataforma (PIM), 140, 141
interacción, 124-28, 179-81, 485-86
máquina de estado, 186, 187, 201, 397, 398 545
modelado
ágil, 59, 80, 120, 145
algorítmico de costos, 635-37
P-CMM, 606, 722
post-arquitectónico, 642-45
semánticos de datos, 93, 130, 145
sistema
de tiempo real, 544-46
formal, 334
subsistema, 185, 186
transmisión, 182
Modelo-Vista-Controlador. *Véase* MVC
Motivación (administración de personal), 603-7
MVC (Modelo-Vista-Controlador), 155, 156, 157, 159, 432-33
MySQL, 199, 433

N

Necesidades
de autorrealización, 604
de estima, 604
de protección (jerarquía de necesidades humanas), 604
fisiológicas, 604
sociales, 604
Nivel
de abstracción (reutilización), 194
de sistema (reutilización), 194
No determinismo, 268, 271-72, 282

No funcional
propiedades emergentes, 269-71
requerimientos, 85, 87-91, 115, 310
de fiabilidad, 324-28
Notación para el Modelado de Procesos de Negocios (BPMN), 530-2, 535-6, 717-8, 730-1
Números de punto flotante, 359

O

OCL (lenguaje de restricción de objetos), 142, 189, 472-73
OMG (Grupo de Gestión de Objetos), 138, 139
Operación
de servicio de catálogo, 522
del sistema, 273-74, 281-85
incompleta, 469
y mantenimiento (fase de ciclo de vida), 31
Oracle, 164
Orientado a objetos
análisis de requerimientos, 100, 129
diseño, 178-89, 201
OWL-S, 525, 536

P

Páginas Web dinámicas, 433
Paquetes de software vertical, 164, 430
Paralelismo, 360
Pares programador/examinador, 210-11
Partición de equivalencia, 214-15
Participantes o partes interesadas, 103
Patrón(es)
arquitectónico de repositorio, 159-60, 172
arquitectónicos (estilos), 155-64, 172, 490-501
software embebido y, 547-53
de arquitectura en capas, 157-59, 167-69, 172
de control ambiental, 548, 550-52
de diseño, 189-93, 430
organizativo, 155
decorador, 192
façade (fachada), 192
iterador, 192
Observador, 190, 191, 192. 433
Observar y Reaccionar, 547, 548-60
productor-consumidor, 182, 544
tubería y filtro, 162, 163-64, 172
P-CMM (Modelo de Madurez de Capacidades del Personal), 606, 722
Perfiles operativos, 227, 402-3, 404, 417, 418
Persistencia del cambio, 720
Persona(s)
autoorientadas, 606
orientadas a tareas, 606
Personal orientado a interacción, 606
Personalización
COTS y, 440, 442
modelo de componentes y, 459
productos de software y, 7, 440
sistemas embebidos y, 699
Perspectiva
dinámica (RUP), 50
estática (RUP), 50
práctica (RUP), 50, 52-53
Picos, 67
PIM (modelo independiente de plataforma), 140, 141
Planeación
de gestión de requerimientos, 112-13
de proyectos, 595, 618-50
calendarización y, 626-30
complementos, 624
desarrollo dirigido por plan y, 623-26
fijación de precio al software y, 621-23
métodos ágiles y, 631-33
proceso de, 624-26
técnicas de estimación, 633-46
de prueba, 209, 349, 407
incremental (XP), 66
modelo en espiral y, 49
riesgo, 597, 600-602
Scrum y, 56, 57, 59, 72-74, 78, 631, 632
Planes
de contingencia (gestión del riesgo), 601
de proyecto, 623-24
Plug-ins, 197, 440
POFOD (probabilidad de falla a pedido), 322-24, 336
Polimorfismo, 190, 251, 435
Política(s)
de seguridad organizacional, 332
explícitas de seguridad, 380-81
Pre/post-condiciones (procesos de software), 28
Predicción, mantenimiento, 246-48
Privacidad
MHC-PMS y, 21-22
requerimientos de, 330
Probabilidad de falla a pedido (POFOD), 322-24, 336
Problemas “malvados” (que no se pueden definir por completo), 111, 272, 287

Procesamiento de entrada por omisión, 360-61
Proceso(s)
  auditables, 346
  check-in/check-out, 692
  consumidor/productor (buffer circular), 544
  de cuarto limpio (Cleanroom), 32, 209, 233, 308, 396, 401, 421, 680
  de desarrollo
    genérico, 53, 178
    transformacional, 396
  de Desarrollo de Software Unificado, 50, 55
  de ingeniería de servicio, 518-27, 534
  de software confiables, 345-48
  operativos confiables, 345
  Racional Unificado. *Véase* RUP
Procesos de software, 12, 27-55
  actividades, 6, 9, 28, 36
  análisis, 710, 714, 715-18
  aseguramiento de, 406,10
  calidad (con base en proceso), 657
  cambio de, 710, 718-21
    características, 346, 709
    CBSE, 461-68
    definida, 9, 24, 53
    fiable, 345-48
    manifiesto ágil y, 60,65
  ciclo de mejora, 710-11, 729
  enfoque de madurez, 706, 728
  especialización (líneas de producto de software), 436
  estándares de, 29, 346, 658, 659
  etapa de capacitación, 720
    estandarización y, 29, 346, 658, 659, 709
    evolución, 237-40
    para aseguramiento de seguridad, 408-10
  excepciones, 718
  medición, 710, 711-14, 729
  mejora de proceso, 29, 705-31
    enfoques al, 706, 728
    marco de mejoramiento de procesos CMMI, 721-28, 729
    metas de, 709, 728
    métodos ágiles y, 706, 728
  modelos, 29-36, 53, 718,729
  patrón de segmentación de proceso (*process pipeline*), 548, 552-53
  proceso de mejora, 708-11
  visión, 154, 172
Procuración del sistema (adquisición del sistema), 273-74, 275-77, 286
Productividad del software, 638
Producto(s)
  a la medida, 7
  de software genéricos, 6-7
  estándares, 658, 659
  métricas de, 672-73
  requerimientos de, 88-89
  resultados de procesos de software, 28
  riesgos de, 596-97
Programación. *Véase también* Programación extrema
  de n-versión, 95, 352-53
  en pares, 66, 71-72
  extrema (XP), 64-72, 77
    ciclo de liberación en, 65
    desarrollo de primera prueba y, 66, 68, 69-70, 223, 231
    ingeniería de requerimientos y, 36, 38
    integración continua y, 65, 66, 68, 76, 78, 697, 698
    métodos ágiles y, 59
    planeación de proyecto y, 632-33
    principios/prácticas, 66
    prueba de aceptación y, 230
    requerimientos del usuario en, 69. 78
  lineamientos de, 355-63
  orientada a aspectos, 580-84
  sin ego, 71
  sin personalismo, 71
  técnicas/actividades, 12, 40-41
  tiempo real, 546
Propagación de falla, 270
Propiedad(es)
  colectiva, 65, 66, 71
  de sistema emergente, 268, 269-71, 286
  funcionales emergentes, 269-71
Proposiciones de cambio, 62, 237-38
Protección, 299-302
  a nivel
    de aplicación, 377
    de plataforma, 377, 378
  definición de, 306
  diseño arquitectónico y, 152
  especificación de, 313-20
  ética y, 16-17
  gestión de riesgos a la seguridad y, 330
  MHC-PMS y, 20-21
  procesos de aseguramiento, 408-10
  requerimientos de, 320, 337
  terminología de, 301
Prototipo Mago de Oz, 46
Proyecto Bootstrap, 722, 731
Prueba(s)
  alfa, 42, 228
  automatizadas, 42, 69, 70, 77, 212-13, 231, 444, 695

Prueba(s) (*continúa*)
basadas
en experiencia, 405
en lineamientos, 213
beta, 42, 228
de aceptación, 42, 70, 228-30
de caja blanca, 585, 586
de caja negra, 215
de defecto, 41, 206, 211, 224, 227, 401, 402, 585
de entregas (*release*), 224-27
de esfuerzo, 218, 227, 232
de interfaz, 217
de partición, 213-15
de regresión, 223
de ruta, 216
de software, 205-33
aceptación, 42, 70, 228-30
alfa, 42, 228
automatizadas, 42, 69, 70, 77, 212-13, 231, 444, 695
beta, 42, 228
de sistema, 42, 219-21
defecto, 41, 206, 211, 224, 227, 401, 402, 585
depuración (*debugging*) *versus*, 41
desarrollo, 41-42, 210-21
estructural, 585, 586
etapas en, 41-43
fiabilidad, 401-3
inspecciones *versus*, 208-9
metas de, 206
métodos ágiles y, 77
modelo de, 210
seguridad, 404-6, 418
validación y, 41
de usuario, 228-30
del sistema, 42, 219-21, 279
estadísticas, 417
estructurales, 585, 586
unitarias, 31, 211-16
PSM (modelos específicos de plataforma), 140, 141
Punto(s)
de corte (*pointcuts*), 571-75, 587
de enlace, 571-75, 587
de vista, 103, 104, 578, 580, 587
individual de falla, evitar, 381-82
Python, 79, 160, 170, 177, 178, 432

Q

QoS. *Véase* Calidad del servicio

R

Ramificación (*branching*), 684, 693
Rastreabilidad (de requerimientos), 113, 114, 225, 409, 601
Rastreo de problemas, 196
Receptor GPS, 512, 513
Recuperabilidad, diseño para, 384-85
Recursión, 360
Rediseño, 31, 39, 44, 57, 58, 72, 110, 278, 599, 600, 623, 643
Redundancia
diversidad y, 343-44, 363, 383
modular triple (TMR), 352
Refactorización, 44, 66, 71, 250-52, 257
de diseño, 252
Regulación
de contabilidad Sarbanes-Oxley, 34, 275
de software, 407
Reingeniería
de datos, 249, 250
de software, 248-50, 257
Relaciones de software interna/externa, 670
Rendimiento
diseño arquitectónico y, 152
pruebas, 227
Reparabilidad, 269, 293
Reportes, 595, 695
Repositorio de versión, 692
Representación del calendario, 627-30
Requerimientos, 86, 101, 104, 108-9
de autenticación, 329
de autorización, 329
de detección de intrusión, 330
de identificación, 329
de inmunidad, 329
de integridad, 329
de la organización, 88-89
de no repudio, 330
de recuperación (requerimiento de fiabilidad funcional), 328
de redundancia (requerimiento de fiabilidad funcional), 328
de seguridad para mantenimiento de sistema, 330
de software, 12, 83
administración del cambio, 113-14
adquisición/análisis, 37, 100-110, 115
análisis y definición (fase de ciclo de vida), 31
clasificación y organización, 101
definición de, 83, 115

desarrollo, 278
descubrimiento, 101, 103-4
documento (especificación de requerimientos de software), 91-94, 115
duraderos, 112
especificación de, 37-38, 84, 85, 86, 94-96, 102
espiral de diseño, 279-80
evolución, 111
funcional, 84-87, 115, 310
gestión de, 100, 111-14, 115
métodos ágiles y, 84
modificación de (desarrollo basado en reutilización), 35
no funcionales, 85, 87-91, 115, 310
priorización y negociación, 101
pruebas (con base en requerimientos), 224-25
rastreabilidad, 113, 114, 225, 409, 601
revisiones, 110, 111, 346, 347
riesgos, 598, 599
validación, 38, 99, 110-11, 115
volátiles, 112
de usuario, 38, 69, 78, 83
del sistema, 38, 83
duraderos, 112
externos, 88-89
funcionales, 84-87, 115, 310
volátiles, 112
Respeto (administración de personal), 603
Responsabilidad ética/profesional, 14-17, 24
Restricciones organizacionales (análisis de procesos), 716
Reutilización
a nivel de objetos, 194, 426
basada en generador, 431
de concepto, 192, 426-27
de función, 426
de sistemas de aplicación, 426
de software, 12, 24, 30, 35-36, 53, 190, 193-95, 201, 425-51
Revisiones, 208, 218
inspecciones y, 663-68
listas de comprobación (análisis de peligros), 317
proceso de revisión, 664-65
Riesgo(s), 301. *Véase también riesgos específicos*
ALARP (Tan Bajos Como Sea Razonablemente Práctico), 315
análisis de, 311-12, 313, 321, 330-32, 336, 597, 598-600
bitácora de, 409-10
de estimación, 598, 599
de herramientas, 598, 599
de proyecto, 596-97
del personal, 598, 599
detección y eliminación de, 302, 319
empresariales, 596-97
especificación de requerimientos dirigida por, 311-12
etapa de descomposición, 312, 313, 322
evitación de, 301-2, 319
gestión de, 49, 50, 330, 595-602
identificación de, 311, 313, 314, 321, 597, 598
indicadores de, 602
intolerables, 315
monitorización de, 597, 602
organizacionales, 598, 599
planeación, 597, 600-602
probabilidad de, 301
proceso de requerimientos de seguridad dirigidos por, 330-32
reducción de, 312, 313, 319-20, 322
severidad de, 301
tecnológicos, 598, 599
tipos de, 600
triángulo, 315-16
valoración de, 314-17
Ritmo sostenible (XP), 66
Robustez (característica de proceso), 346, 709
ROCOF (tasa de ocurrencia de fallas), 332-34, 336
Roles (procesos de software), 28
RPC (llamadas a procedimiento remoto), 486, 498
RTOS. *Véase* Sistemas operativos de tiempo real
Ruby, 12, 79, 432
RUP (Proceso Racional Unificado), 50-53, 178

S

SaaS. *Véase* Software como servicio
SAP, 7, 164, 442
SCI (ítem de configuración de software), 684
Scrum, 56, 57, 59, 72-74, 78, 631, 632
Seguridad, 302-5
amenazas a la, 390
aseguramiento, 393-421
brechas, 305
como propiedad emergente, 269
confiabilidad y, 8, 12, 24, 292
confianza y, 8, 10
definición de, 306

Seguridad (*continúa*)
diseño
arquitectónico y, 152, 376-80
para, 375-86
especificación de, 329-33
fallas, 382
gestión del riesgo, 330, 369-75, 390
ingeniería, 366-92
lineamientos de, 380-85
y usabilidad, 382
lista de verificación, 406
políticas de, 380-81
pruebas de, 404-6, 418
requerimientos de, 329-33, 337
de auditoría, 330
terminología de, 303
validación de, 418
Separación de competencias, 567-71
Servicio(s), 13, 36, 509, 514
de coordinación, 519, 534
de utilidad, 519, 534
evolución y, 236-37
empresariales, 519, 534
orientados
a entidades, 519
a tareas, 519
RESTful, 483, 511, 512, 536
Web, 13, 36, 509, 514
clasificación de, 520, 534
como componentes reutilizables, 514-18
componentes *versus*, 514-18
construcción (mediante composición) de, 528-29
coordinación de, 519, 534
definición de, 13, 509, 515
desarrollo de software y, 527-34
empresariales, 519, 534
enfoque RESTful y, 483, 511, 512, 536
estándares de, 36, 386, 426, 461, 483, 510
interfaces, 11
modelo de proceso empresarial y, 534
pruebas de, 533-34
utilidad de, 519, 534
WSDL y, 458
Simplicidad (métodos ágiles), 60, 65, 66
Simulación, 11
Simuladores, 196, 626, 694
Sistema(s)
bancario, Internet, 494
basados en Web, 13-14
build GNU, 196
cliente-servidor, 160-63, 172, 488-89, 505
complejos, 266-73
críticos
aseguramiento de proceso y, 407
definición de, 291
falla de, 290-91, 306
falla de sistema y, 300
lenguaje SPARK/Ada y, 336
métodos ágiles y, 60, 348
negocios, 248, 291, 324, 387, 390
para la protección, 299-302
proceso de desarrollo y, 418
triángulo de riesgo para, 315-16
de administración de inventario, 581
de adquisición de datos, 553, 554, 560, 569
de alta rapidez, 553, 554, 560, 569
de alarma contra robo, 540, 541, 549, 550, 551, 555, 556
de alta disponibilidad, 152, 198, 295, 325, 351
de aplicación comerciales. *Véase* COTS, sistemas
de asignación de recursos, arquitectura de, 437
de banca por Internet, 494
de comercio equitativo, 379 389
de control de bomba de insulina, 18-20
análisis de árbol de fallas para, 317-19
argumento de seguridad informal y, 416-17
argumentos estructurados y, 412-13
bitácora de riesgo para, 410
cálculo de dosis de insulina (con verificaciones de seguridad) en, 415
clasificación de riesgo para, 316-17
componentes de hardware (diagrama), 19
diagrama de procesamiento de pedido en, 135
especificación de requerimiento estructurado de, 96, 97
especificación formal para, 334
especificación tabular y, 98
falla en el, 327
jerarquía de afirmación de seguridad para, 414
modelo de actividad de, 19, 135
propiedades de confiabilidad para, 293
requerimientos de seguridad para, 320
riesgos en, 314
de control de robot de empacado, 148, 149
de control de tráfico aéreo, 139, 274
de despacho de vehículos, 437
de entretenimiento, 11
de equipajes, aeropuerto de Denver, 266
de frenado antibloqueo, 552
de gestión
de recursos, 167-69, 172, 436, 437, 460, 546, 558, 561
de tráfico, 491

de información, 18, 167-69
automotriz, 512-13
de pacientes. *Véase* MHC-PMS
dentro del auto, 512-13
meteorológica, 22
de inventario de equipo, 579
de planeación de recursos empresariales. *Véase* Sistema(s) ERP
de procesamiento
de lenguaje, 166, 169-72
de transacciones, 165-67, 172, 227, 348, 493
por lotes, 11, 163
de protección, 349-50
de recolección de datos, 11
con base en sensores, 18
de registros
de pacientes (PRS), 125, 127, 128
médicos. *Véase* MHC-PMS
de software. *Véase también* Sistemas distribuidos
complejo, 266-73
con extensiones, 576-77
definición de, 266-67
sociotécnico, 263-88
sistemas de, 11
tipos de, 4, 6-7, 10-12, 24, 266-67
de software embebidos, 11, 17-18, 537-64. *Véase también* Sistemas de tiempo real
desarrollo anfitrión destino y, 198
patrones arquitectónicos y, 547-53
personalización y, 699
RUP y, 53
simuladores y, 196
de tiempo real, 538-39, 561
análisis de temporización de, 554-57
diseño, 540-44
modelado de, 544-46
programación de, 546
de verificación de controlador, 336, 400
determinista, 271
distribuidos, 480-81
conflictos, 481-88
patrones arquitectónicos para, 490-501
sistemas cliente-servidor, 160-63, 172, 488-89, 505
ventajas de, 480, 505
empresariales críticos, 248, 291, 324, 387, 390
ERP (planeación de recursos empresariales), 7, 429, 430, 442, 443, 444, 448, 450, 451, 671
heredados
gestión de, 245, 252-57, 285
interfaces de servicio y, 525-27, 534
wrapping (envoltura), 250, 430, 464, 526
nucleares, 17
operativos de tiempo real (RTOS), 558-61
sociotécnicos, 263-88
tipo E, 240
Skype, 499
SOA. *Véase* Arquitecturas orientadas a servicios
SOAP, 509, 510, 511, 512, 513, 516, 535
Software
atributos de, 6, 8, 24
cero defectos, 32, 396
como servicio (SaaS), 13, 501-5
conflictos con, 10
definición de, 5, 6, 24
eficiencia de, 8, 12, 24
fallas de, 4, 12
primario crítico para la seguridad, 300
regulación de, 407
relaciones interna/externa, 670
tipos de producto, 6-7, 10-12, 24
Soporte de herramientas, 598, 599
análisis de proceso, 716
Sprint (Scrum), 73, 78, 665
SQL (Lenguaje de Consulta Estructurado), 174, 199, 383, 389, 405, 433, 494
SRS (especificación de requerimientos de software), 91-94, 115
Subsistemas, 267
Subversion, 196, 204, 691, 704
Suite de métricas de Chidamber y Kemerer (CK), 673-4
Sumas de verificación (checksums) de código fuente, 696
Supervivencia, 293, 386-90

T

Tablero de control de cambio (CCB), 685, 686, 687
Tarjetas
de historia, 65, 66, 67, 69
de tarea, 68, 69
Tasa de ocurrencia de fallas (ROCOF), 332-34, 336
TDD. *Véase* Desarrollo dirigido por pruebas
Técnicas
basadas en experiencia (técnicas de estimación), 634
de estimación (planeación del proyecto), 633-46
modelado algorítmico de costos, 634, 635-37
modelo COCOMO II, 248, 464, 637-45
técnicas basadas en experiencia, 634

Tejido (*weaving*) (AOSE), 572
Tipos de personalidad, 606
TMR (redundancia modular triple), 352
Tolerancia
  al cambio, 44
  al error, 293
Trabajo en equipo, 607-14
Traducción de código fuente, 249

U

UML (Lenguaje de Modelado Unificado)
  definición de, 121
  diagramas de despliegue, 129, 197, 198
  diseño
    arquitectónico y, 154
    orientado a objetos con, 178-89
  mensajes entrada/salida y, 524
  modelos de estado y, 545
  RUP y, 50, 52
  tipos de diagrama, 120, 185
  XUML y, 142
UML ejecutable (XUML), 142
URI (identificadores universales de recurso), 512
Usabilidad
  como propiedad emergente, 269
  lineamientos de, y seguridad, 382
  patrones de, 155, 175
Utilidad make Unix, 196

V

V&V (verificación y validación), 208, 394
  AOSE y, 584-87
Validación de software, 6, 9, 24, 28, 41-43
  AOSE y, 584-87
  definición de, 53
  meta de, 207
  modelo en espiral y, 49
  validación de requerimientos, 38, 99, 110-11, 115
  verificación *versus*, 206-7
Valoración
  de exposición (requerimientos de seguridad), 331
  de factibilidad (requerimientos de seguridad), 331
  del entorno, 255
Verificabilidad, 110
Verificación
  de enlaces de arreglo, 362
  de error característico, 399-400
  de requerimientos (requerimiento de confiabilidad funcional), 328
  de software
    AOSE y, 584-87
    formal, 340, 396, 397, 398, 405, 406
    meta de, 207
    métodos formales y, 395-97
    validación *versus*, 206-7
  estática, 336
  formal, 340, 396, 397, 398 405, 406
Verificadores
  de contraseña, 405
  de modelo SPIN, 397
Virus, 14, 294, 303, 329, 367, 404
Visión
  arquitectónica, 153-55,172
  conceptual, 154, 172
  lógica, 153, 173
Vista(s)
  arquitectónicas, 153-55, 172
  física, 154, 172
VM. *Véase* Gestión de versiones
V-spec, 354
  evitar vulnerabilidad, 305

W

WS-BPEL, 510, 511, 529, 530, 532
WSDL (Lenguaje de Definición de Servicio Web), 458, 509, 510, 511, 512, 515, 516, 517, 522, 523, 525, 529, 534, 535

X

XML, 510
  mensaje (ejemplo), 485
  protocolos basados en XML, 509
  sistemas de procesamiento de lenguaje y, 166, 169, 171
XP. *Véase* Programación extrema
XUML (UML ejecutable), 142

Índice de autores

A

Abbot, R., 183, 203
Abrial, J. R., 396, 420
Abts, C., 258, 447, 448, 450, 477, 650
Ackroyd, S., 269, 288
Adams, E. N., 298, 308
Addy, E., 449, 478
Ahern, D. M., 683, 704, 722, 730
Aksit, M., 588
Alberts, C., 370, 391
Alexander, C., 189, 203
Alexander, I., 371, 391
Ambler, S. W., 59, 80, 120, 145
Amelot, A., 336, 339
Anderson, E. A., 340
Anderson, R., 367, 391, 392, 404, 420
Anderson, R. J., 269, 288
Andrea, J., 223, 232
Andrews, M., 418
Andrews, T., 533, 536
Appleton, B., 156, 174, 702
Arisholm, E., 72, 80
Arlow, J., 50, 55
Armour, P., 647
Aron, J. D., 612, 616
Arthur, L. J., 238, 258
Artus, D. J. N., 535
Astels, D., 69, 80
Atlee, J. M., 115
Avizienis, A. A., 353, 354, 365
Ayewah, N., 418

B

Badeau, F., 336, 339
Baier, C., 397, 420
Baker, F. T., 612, 616
Baker, T., 441, 450
Balcer, M. J., 141, 142, 145
Balk, L. D., 441, 450
Ball, T., 336, 339, 400, 420, 421
Bamford, R., 661, 679, 683, 704
Baniassad, E., 583, 589
Banker, R. D., 247, 258
Barnard, J., 670, 679
Barnes, J. P., 336, 339
Basili, V. R., 674, 679, 712, 730
Bass, B. M., 606, 617
Bass, L., 149, 150, 154, 173, 174
Bate, R., 722, 731
Baumer, D., 432, 450
Bayersdorfer, M., 200, 203
Beck, K., 57, 59, 64, 78, 80, 91, 117, 183, 204, 221, 232, 259, 611, 617, 631, 649
Beedle, M., 57, 59, 72, 81
Belady, L., 240, 241, 259
Bell, R., 315, 339
Bellagio, D. E., 196, 204
Bennett, K. H., 236, 257, 260
Berczuk, S. P., 155, 174, 702
Berghel, H., 305, 308, 383, 392
Bernstein, P. A., 175, 487, 506, 507
Berry, G., 539, 564
Bezier, B., 214, 232

Birrer, I., 575, 589
Bishop, M., 332, 339, 367, 392
Bishop, P., 411, 420
Bloomfield, R. E., 411, 420
Boehm, B. W., 25, 29, 48, 49, 55, 62, 78, 80, 207, 232, 248, 258, 300, 308, 348, 365, 447, 448, 450, 464, 477, 600, 617, 634, 635, 637, 639, 640, 641, 644, 646, 647, 649, 650, 656, 679
Booch, G., 55, 120, 145, 146, 150, 173, 174
Bosch, J., 149, 153, 160, 174
Bott, F., 24
Bowen, J. P., 337
Brazendale, J., 315, 339
Brereton, P., 506, 536
Brilliant, S. S., 354, 365
Brinch-Hansen, P., 543, 564
Brooks, F. P., 24, 612, 615, 616
Brownsword, L., 441, 450
Budgen, D., 40, 55, 506, 536
Burns, A., 504, 557, 561, 562, 564
Buschmann, F., 156, 174, 175, 190, 204

C

Cabrera, L. F., 533, 536
Carlson, D., 68, 80, 197, 204
Carr, N., 514, 536
Chandra, S., 397, 421
Checkland, P., 269, 288
Chen, P., 130, 145
Cheng, B. H. C., 115
Chidamber, S., 673, 679
Chrissis, M. B., 704, 722, 729, 731
Clarke, E. M., 334, 339
Clarke, K., 286
Clarke, S., 583, 588
Clayberg, E., 197, 204
Clement, A., 566, 567, 589
Clements, P., 154, 173, 174, 175
Clouse, A., 704, 730
Coad, P., 183, 204
Cockburn, A., 59, 72, 80
Codd, E. F., 130, 145
Cohn, M., 59, 80, 631, 647
Coleman, A., 24
Coleman, D., 247, 259
Collins-Sussman, B., 204, 704
Colyer, A., 566, 567, 584, 589
Connaughton, C., 678
Conradi, R., 729
Constantinos, C., 586, 589
Cooling, J., 554, 562, 564
Coplien, J. H., 155, 175
Coulouris, G., 480, 507
Councill, W. T., 451, 455, 476, 477, 478
Crabtree, A., 108, 117
Cranor, L., 382, 392
Crnkovic, I., 476
Crosby, P., 655, 680
Croxford, M., 400, 421
Cummings, R., 306
Cunningham, W., 81, 183, 204
Curtis, B., 678, 704, 722, 731
Cusamano, M., 210, 232, 428, 450

D

Dahl, O. J., 233, 589
Davis, A. M., 83, 117
Deibler, W. J., 661, 679, 683, 704
Demarco, T., 62, 80, 134, 145, 615
Denger, C., 678
Denning, P. J., 75, 80
Dibble, P. C., 546, 564
Dijkstra, E. W., 206, 233, 359, 365, 543, 564, 585, 589
Dorofee, A., 370, 391
Douglass, B. P., 545, 548, 564
Drobna, J., 60, 80
Dunn, W. R., 306
Dunteman, G., 606, 617
Dybå, T., 729
Dyer, M., 81, 308, 680

E

Easterbrook, S., 577, 589
Eaton, J., 24
Edwards, J., 507
El-Amam, K., 673, 680
Ellison, R., 293, 308, 387, 388, 390, 392
Elrad, T., 588
Endres, A., 300, 308, 671, 680
Erickson, J., 120, 145

Erl, T., 510, 519, 534, 535, 536
Erlikh, L., 235, 259
Evans, D., 400, 421

F

Fagan, M. E., 209, 233, 665, 666, 680
Fayad, M. E., 432, 450
Felsing, J. M., 59, 81
Fenton, N., 671, 680
Filman, R. E., 588
Finkelstein, A., 577, 589
Firesmith, D. G., 329, 337, 339
Fitzpatrick, B. W., 204, 704
Fogel, K., 202
Fowler, M., 251, 259

G

Galin, D., 678
Gamma, E., 155, 175, 190, 191, 202, 204, 432, 433, 450
Garfinkel, S., 382, 392
Garlan, D., 152, 156, 171, 173, 175, 447, 449, 450
Gilb, T., 666, 680
Gomaa, H., 545, 564
Goodenough, J. B., 418
Goth, G., 506
Gotterbarn, D., 15, 24, 26
Gradecki, J. D., 584, 589
Grady, R. B., 670, 680
Graham, D., 666, 680
Graydon, P. J., 411, 421
Green, S., 712, 730
Griss, M. L., 428, 450, 451, 477
Guimaraes, T., 243, 259
Gunning, R., 668, 680

H

Haase, V., 722, 731
Hall, A., 335, 336, 339
Hall, E., 596, 617
Hall, T., 411, 420, 671, 680
Hamilton, S., 397, 418, 421
Hammer, M., 130, 145, 721, 731
Hanssen, G. K., 662, 680
Hardstone, G., 286
Harel, D., 135, 145, 187, 204, 545, 564
Harkey, D., 488, 507
Harold, E. R., 166, 175
Harrison, N. B., 155, 175
Hass, A. M. J., 702
Hatton, L., 355, 365
Heineman, G. T., 451, 455, 476, 477, 478
Helm, R., 155, 175, 190, 191, 202, 204, 432, 433, 450
Henney, K., 174, 204
Heslin, R., 614, 617
Highsmith, J. A., 59, 80
Hinchey, M. G., 337
Hnich, B., 476
Hoare, C. A. R., 233, 543, 564, 589
Hofmeister, C., 150, 154, 175
Holcombe, M., 78
Holdener, A. T., 14, 26, 433, 451, 501, 507
Holzmann, G. J., 397, 421
Hopkins, R., 75, 80, 235, 259
Hudak, J. J., 418
Huff, C., 17, 26
Huisman, J. W., 61, 81, 240, 259
Hull, R., 130, 145
Humphrey, W., 653, 666, 680, 707, 711, 731
Hunter, D., 166, 175
Husted, T., 42, 55, 70, 81, 222, 233

I

Ince, D., 661, 680

J

Jaatun, M. G., 337
Jacobson, I., 55, 106, 117, 124, 145, 146, 428, 451, 466, 477, 568, 576, 581, 583, 588, 589
Jahanian, F., 317, 339
Jain, P., 156, 175, 190, 204
Janoff, N. S., 74, 81
Jeffrey, R., 671, 678, 680

Jeffries, R., 59, 80, 120, 145, 221, 223, 233
Jenkins, K., 75, 80, 235, 259
Johnson, D. G., 26
Johnson, R., 155, 175, 190, 191, 202, 204, 432, 433, 450
Johnson, R. E., 250, 259
Jonsson, P., 117, 145, 451, 477
Jonsson, T., 476

K

Kafura, D., 247, 259
Kan, S. H., 678
Kaner, C., 225, 233
Katoen, J. -P., 397, 420
Katz, S., 586, 589
Kavantzas, N., 533, 536
Kazman, R., 173, 174
Kedia, A., 441, 450
Kemerer, C., 258, 673, 679
Kent, S., 138, 145
Kerievsky, J., 252, 259
Kiczales, G., 566, 589
Kilpi, T., 670, 680
King, R., 130, 145
Kircher, M., 156, 175, 190, 204
Kitchenham, B., 670, 678, 680
Kiziltan, Z., 476
Kleppe, A., 139, 145, 472, 478
Knight, J. C., 354, 365, 408, 421
Konrad, M., 722, 729, 731
Kotonya, G., 91, 117, 461, 477, 577, 589
Kozlov, D., 247, 259
Krogstie, J., 243, 259
Krutchen, P., 50, 54, 55, 120, 153, 175
Kume, H., 300, 308
Kuvaja, P., 722, 731

L

Lange, C. F. J., 154, 175
Laprie, J. -C., 290, 308
Larman, C., 59, 80, 100, 117, 202
Larochelle, D., 400, 421
Larus, J. R., 400, 421
Lau, K. -K., 454, 458, 476, 477
Laudon, K., 17, 26
Lee, E. A., 539, 564
Leffingwell, D., 75, 76, 78, 80
Lehman, M. M., 240, 241, 242, 257, 258, 259, 700
Leveson, N. G., 312, 317, 337, 338, 340, 354, 365, 408, 421
Lewis, B., 536
Lewis, G. A., 257
Lewis, P. M., 166, 175
Lezeiki, N., 584, 589
Lientz, B. P., 243, 259
Lindvall, M., 76, 80, 348, 365
Linger, R. C., 81, 233, 308, 390, 392, 421, 680
Lister, T., 615
Littlewood, B., 297, 308
Lomow, G., 514, 536
Londeix, B., 646, 650
Longstaff, T., 308, 390, 392
Lovelock, C., 509, 536
Lutz, R. R., 217, 233, 300, 308
Lyu, M. R., 364, 365

M

Maciaszek, L., 143
Marshall, J. E., 614, 617
Martin, C. D., 17, 26
Martin, D., 109, 117, 155, 175
Martin, J., 57, 80
Martin, R. C., 223, 233
Maslow, A. A., 603, 617
Massol, V., 42, 55, 70, 81, 197, 204, 222, 233
Matsumoto, Y., 427, 451
McCabe, T. J., 247, 259
McConnell, S., 615, 666, 680
McDougall, P., 499, 507
McGraw, G., 375, 380, 391, 392
McIlroy, M. D., 426, 451
McLeod, D., 130, 145
Mead, N. R., 308, 390, 392
Means, W. S., 166, 175
Meland, P. H., 337
Mellor, S. J., 135, 139, 141, 142, 143, 145, 146, 183, 204

Melnik, G., 221, 224, 233
Meyer, B., 473, 477
Miers, D., 530, 536
Mili, A., 449, 464, 478
Mili, H., 449, 464, 478
Miller, K., 24, 26
Miller, S. A., 731
Miller, S. P., 336, 340
Milligan, T. J., 196, 204
Mills, H. D., 51, 81, 298, 308, 666, 680
Mok, A. K., 317, 339
Moore, A., 308, 392
Moore, E., 75, 81
Morisio, M., 440, 447, 449, 451
Morris, E., 441, 450
Mumford, E., 269, 288
Musa, J. D., 403, 418, 421
Myers, W., 646, 650

N

Nakajo, T., 300, 308
Naur, P., 5, 26
Neuman, B. C., 483, 507
Neustadt, I., 50, 55
Newcomer, E., 514, 536
Ng, P. -W., 568, 576, 581, 583, 588, 589
Nguyen, T., 400, 421
Nii, H. P., 160, 175
Nosek, J. T., 243, 259
Nuseibeh, B., 445, 451, 577, 589

O

O'Connell, E., 729
Offen, R. J., 671, 680
O'Leary, D. E., 442, 451
Opdahl, A. L., 371, 392
Opdyke, W. F., 250, 259
Oram, A., 499, 506, 507
Orfali, R., 488, 496, 507
Osterweil, L., 731
Ould, M., 54, 596, 617, 721, 731
Ourghanlian, A., 400, 421
Overgaard, G., 117, 145
Owl_Services_Coalition, 536

P

Palmer, S. R., 59, 81
Palvia, P., 243, 259
Parnas, D. L., 345, 365
Parrish, A., 72, 81
Paulk, M. C., 683, 704, 713, 722, 731
Pautasso, C., 512, 536
Peach, R. W., 683, 704
Perrow, C., 302, 308
Peterson, J. L., 317, 340
Pfarr, T., 441, 451
Pfleeger, C. P., 303, 308, 367, 371, 392, 406, 421
Pfleeger, S. L., 303, 308, 367, 371, 392, 406, 421
Pilato, C., 196, 204, 691, 704
Plakosh, D., 257
Poole, C., 61, 81, 240, 259
Pooley, R., 107, 117, 143
Pope, A., 454, 478, 483, 507
Price, A., 670, 679
Prowell, S. J., 209, 233, 401, 421
Pshigoda, G., 74, 81
Pulford, K., 713, 731
Pullum, L. L., 348, 364, 365

R

Rajlich, V., 236, 257, 260
Randell, B., 5, 26
Raymond, E. S., 198, 204
Reason, J., 282, 283, 284, 286, 288
Reddy, G. R., 247, 259
Regan, P., 397, 418, 421
Reis, J. E., 441, 451
Rettig, M., 46, 55
Richardson, L., 511, 536
Rising, L., 74, 81
Rittel, J., 272, 288
Robertson, J., 97, 115, 117
Robertson, S., 97, 115, 117

Rogerson, S., 24, 26
Rombach, H. D., 671, 674, 679, 680, 712, 730
Rosenberg, D., 62, 81
Rouncefield, M., 286
Rowland, D., 24
Royce, W. W., 30, 55, 637, 650
Rubel, D., 197, 204
Ruby, S., 511, 536
Rumbaugh J., 50, 55, 120, 124, 145, 146

S

Saiedian, H., 729
Sametinger, J., 458, 460, 478
Sawyer, P., 117, 577, 589, 714, 719, 731
Scacchi, W., 54
Schmidt, D. C., 40, 55, 138, 146, 156, 174, 175, 190, 204, 431, 432, 433, 450, 451
Schneider, S., 32, 55, 396, 421
Schneier, B., 306, 331, 340, 380, 392
Scholes, J., 269, 288
Schuh, P., 703
Schwaber, K., 57, 59, 72, 81, 631, 650
Scott, J. E., 444, 451
Scott, K., 143, 145
Seacord, R. C., 257
Seiwald, C., 702
Selby, R. W., 210, 232
Shaw, M., 152, 156, 171, 173, 175
Shlaer, S., 183, 204
Shrum, S., 729, 731
Shull, F., 678
Siau, K., 120, 145
Silberschatz, A., 543, 564
Sindre, G., 371, 392
Sjøberg, D., 729
Smits, H., 74, 81
Sommerville, I., 91, 109, 115, 117, 155, 175, 286, 449, 577, 589, 685, 714, 719, 731
Sousa, M. J., 243, 260
Spafford, E., 305, 308, 383, 392
Spens, J., 75, 81
St. Laurent, A., 200, 204
Stahl, T., 139, 146
Stalhane, T., 662, 680
Stapleton, J., 57, 59, 81
Stephens, M., 62, 81
Stevens, P., 107, 117, 143
Stevens, R., 266, 288
Stolzy, J., 317, 340
Storey, N., 317, 340, 345, 365
Suchman, L., 108, 117, 269, 288
Sutherland, J., 74, 81
Sutton, J., 400, 421
Swanson, E. B., 243, 259
Swartz, A. J., 266, 288
Szyperski, C., 455, 462, 476, 478

T

Tanenbaum, A. S., 480, 506, 507, 543, 564
Thayer, R. H., 266, 286, 288
Thomé, B., 266, 288
Tøndel, I. A., 337
Torchiano, M., 440, 447, 449, 451
Torres-Pomales, W., 348, 365
Tracz, W., 441, 451
Turner, M., 506, 509, 536
Turner, R., 29, 55, 704, 730

U

Ulrich, W. M., 249, 260
Ulsund, T., 729

V

Valeridi, R., 648
Van Steen, M., 480, 506, 507
Vesperman, J., 704
Viega, J., 375, 380, 391, 392
Viller, S., 109, 117, 589
Visser, W., 398, 421
Vlissides, J., 155, 175, 190, 191, 202, 204, 432, 433, 450
Voas, J., 402, 421
Voelter, M., 139, 146

W

Wang, Z., 454, 458, 476, 477
Ward, P., 135, 146
Warmer, J., 145, 472, 478
Warren, I. E., 253, 260
Weber, C. V., 704, 731
Weigers, K. M., 115
Weinberg, G., 71, 81
Weinreich, R., 458, 460, 478
Weinstock, C. B., 418
Weise, D., 143, 145
Wellings, A., 557, 561, 562, 564
Westmark, V. R., 387, 392
Wheeler, D. A., 380, 392
White, S., 266, 288
White, S. A., 530, 536, 718, 731
Whittaker, J. A., 216, 221, 231, 233, 418
Williams, L., 72, 80, 81, 421
Wingerd, L., 702
Wirfs-Brock, R., 183, 204
Wordsworth, J., 32, 55, 340, 396, 421
Wosser, M., 428, 450

Y

Yacoub, S., 449, 478
Yamaura, T., 231
Yourdon, E., 183, 204

Z

Zheng, J., 400, 421