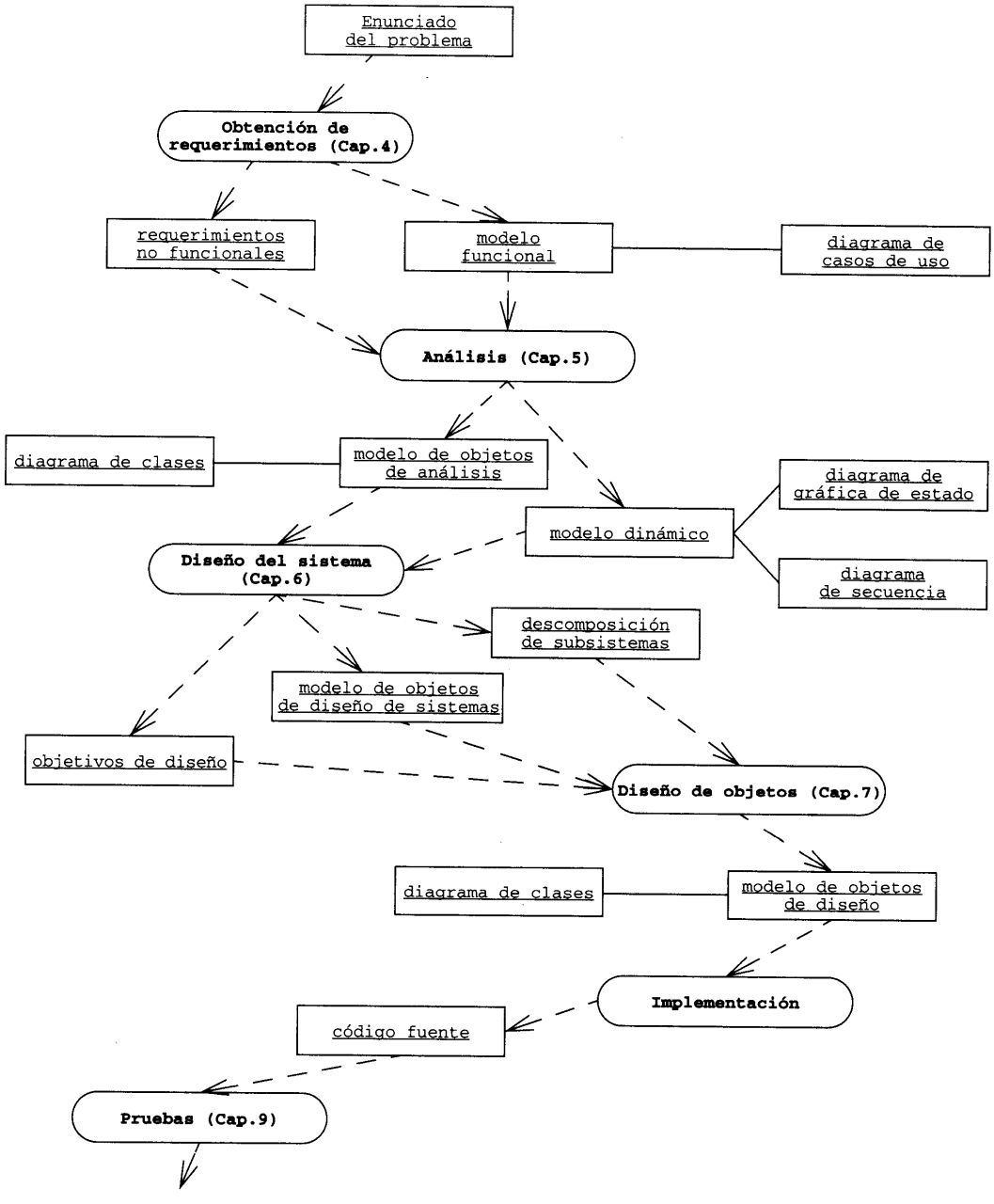
INGENIERÍA DE SOFTWARE ORIENTADO A OBJETOS

BERND BRUEGGE • ALLEN H. DUTOIT

Prentice Hall ®

Manejo de la complejidad



Ingeniería de software orientado a objetos

Bernd Bruegge

Technical University of Munich
Munich, Germany

Allen H. Dutoit

Carnegie Mellon University
Pittsburgh, PA, United States

TRADUCCIÓN:

Sergio Luis María Ruiz Faudón
Ingeniero Químico, Facultad de Ciencias Químicas, Universidad Veracruzana, campus Orizaba

REVISIÓN TÉCNICA:

Rafael Gamboa Hirales
Maestro en Ingeniería, Universidad Politécnica de Madrid
Departamento Académico de Computación, Instituto Tecnológico Autónomo de México

Martha Rosa Cordero López
Maestra en Ciencias, Centro de Investigación y de Estudios Avanzados, Instituto Politécnico Nacional
Escuela Superior de Cómputo
Instituto Politécnico Nacional

Marco Antonio Dorantes González
Maestro en Ciencias, Centro de Investigación y de Estudios Avanzados, Instituto Politécnico Nacional
Escuela Superior de Cómputo
Instituto Politécnico Nacional

PEARSON Educación ®

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

Datos de catalogación bibliográfica

BRUEGGE, BERND y DUTOIT, ALLEN H.
Ingeniería de software orientado a objetos

PEARSON EDUCACIÓN, México, 2002

ISBN: 970-26-0010-3
Área: Universitarios

Formato: 18.5 x 23.5 cm Páginas: 576

Versión en español de la obra titulada *Object-Oriented Software Engineering*, de Bernd Bruegge y Allen H. Dutoit, publicada originalmente en inglés por Prentice Hall Inc., Upper Saddle River, New Jersey, E.U.A.

Esta edición es la única autorizada.

Original English language title by
Prentice Hall Inc.

ISBN 0-13-489725-0

Edición en español:
Editor: Guillermo Trujano Mendoza
Editor de desarrollo: Jorge Alberto Velázquez Arellano
Supervisor de producción: José D. Hernández Garduño

Edición en inglés:
Publisher: Alan Apt
Project Manager: Ana Arias Terry
Editorial Assistant: Toni Holm
Editorial/production supervision: Ann Marie Kalajian and Scott Disanno
Editor-in-Chief: Marcia Horton
Executive Managing Editor: Vince O'Brien
Cover Design: Amy Rosen
Art Director: Heather Scott
Assistant to Art Director: John Christiana
Cover Image: © Superstock, Tamserky. A Himalayan Peak. Khumbu Nepal
Book Photos: Bernd Bruegge and Blake Ward
Manufacturing Buyer: Pat Brown

PRIMERA EDICIÓN, 2002

Atlacomulco Núm. 500-5° Piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México

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

ISBN 970-26-0010-3

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 – 04 03 02

Para Goeg, Toby y Clara, y mis padres.
—B.B.

Para mi mujercita, con amor.
—A.H.D.

Prefacio

El K2 se eleva a 8,611 metros en la cordillera Karakorum en el Himalaya occidental. Es el segundo pico más grande del mundo y está considerado como el más difícil de escalar entre los que miden más de 8,000 metros. Una expedición al K2 dura, por lo general, varios meses y se realiza en el verano, cuando el clima es más favorable. Aun en verano son frecuentes las tormentas de nieve. Una expedición requiere miles de kilogramos de equipo, incluyendo instrumentos para escalar, pertrechos de protección para clima más severo, tiendas, comida, equipo de comunicación y paga y zapatos para cientos de porteadores. La planeación de una expedición de éstas requiere una cantidad de tiempo significativa en la vida de un escalador y requiere docenas de participantes en funciones de apoyo. Una vez que están en el lugar, muchos eventos inesperados, como avalanchas, huelgas de porteadores o fallas de equipo forzarán a los escaladores a adaptarse, encontrar nuevas soluciones o retirarse. La tasa de éxito de las expediciones al K2 en la actualidad es menor a 40%.

El Sistema Aeroespacial Nacional (NAS, por sus siglas en inglés) de Estados Unidos supervisa y controla el tráfico aéreo en ese país. El NAS incluye más de 18,300 aeropuertos, 21 centros de control de tráfico de rutas aéreas y más de 460 torres de control. A esto se añaden más de 34,000 piezas de equipo que incluyen radares, interruptores de comunicaciones, radios, sistemas de cómputo y pantallas. La infraestructura actual está envejeciendo con rapidez. Las computadoras que dan apoyo a los 21 centros de control de tráfico de rutas aéreas, por ejemplo, son mainframes 3083 de IBM que se remontan a principios de los años ochenta. En 1996, el gobierno de Estados Unidos inició un programa para modernizar la infraestructura del NAS, incluyendo mejoras como navegación por satélite, comunicaciones digitales entre controladores y pilotos, y un grado más alto de automatización para el control de las rutas aéreas, el orden en que aterrizan los aviones y el control del tráfico terrestre conforme los aviones se aproximan y se alejan de las pistas. Sin embargo, la modernización de tal infraestructura compleja sólo puede realizarse en forma gradual. En consecuencia, mientras se introducen nuevos componentes que proporcionen nueva funcionalidad, todavía hay que dar soporte a los componentes más antiguos. Por ejemplo, durante el periodo de transición, un controlador tendrá que ser capaz de usar canales de voz tanto analógicos como digitales para comunicarse con los pilotos. Por último, la modernización del NAS

coincide con un incremento dramático del tráfico aéreo global, el cual se predice que se duplicará dentro de los próximos 10 a 15 años. El esfuerzo de modernización anterior del NAS, llamado Sistema de Automatización Avanzado (AAS, por sus siglas en inglés), fue suspendido en 1994 debido a problemas relacionados con software después de haber fallado por varios años en su tiempo de entrega inicial y excederse varios miles de millones de dólares en su presupuesto.

Los ejemplos anteriores tratan sobre sistemas complejos en donde las condiciones externas pueden disparar cambios inesperados. La complejidad pone el problema más allá del control de un solo individuo. El cambio fuerza a los participantes a apartarse de soluciones bien conocidas e inventar nuevas. En ambos ejemplos, varios participantes necesitan cooperar y desarrollar nuevas técnicas para manejar esos retos. No hacerlo así dará como resultado que no se alcance el objetivo.

Este libro trata acerca de la conquista de sistemas de software complejos y cambiantes.

El tema

El dominio de aplicación (planeación de una expedición a una montaña, control de tráfico aéreo, sistemas financieros, procesamiento de textos) incluye, por lo general, muchos conceptos con los cuales no están familiarizados los desarrolladores de software. El dominio de solución (juegos de herramientas de interfaz de usuario, comunicaciones inalámbricas, middleware, sistemas de administración de bases de datos, sistemas de procesamiento de transacciones, computadoras portátiles, etc.) con frecuencia está inmaduro y proporciona a los desarrolladores muchas tecnologías de implementación en competencia. En consecuencia, el proyecto del sistema y su desarrollo son complejos, involucrando muchos componentes, herramientas, métodos y personas diferentes.

Conforme los desarrolladores aprenden más acerca del dominio de aplicación a partir de sus usuarios, actualizan los requerimientos de sistema. Conforme los desarrolladores aprenden más acerca de las tecnologías que surgen, o acerca de las limitaciones de las tecnologías actuales, adaptan el diseño del sistema y su implementación. Conforme el control de calidad encuentra defectos en el sistema y los usuarios solicitan nuevas características, los desarrolladores modifican el sistema y sus productos de trabajo asociados, dando como resultado un cambio continuo.

La complejidad y el cambio representan retos que hacen que sea imposible para una sola persona controlar el sistema y su evolución. Si se les controla en forma inadecuada, la complejidad y el cambio invalidan la solución antes de que se entregue, aunque el objetivo esté a la vista. Demasiados errores en la interpretación del dominio de aplicación hacen que la solución sea inútil para los usuarios, forzando una retirada de la ruta o del mercado. Tecnologías de implementación inmaduras o incompatibles dan como resultado una baja confiabilidad y retrasos. La falla en el manejo de los cambios introduce nuevos defectos en el sistema y degrada el desempeño más allá de su utilidad.

Este libro refleja más de diez años de construcción de sistemas y de enseñanza de cursos de proyectos de ingeniería de software. Hemos observado que a los estudiantes se les enseñan técnicas de programación y de ingeniería de software aisladas, con frecuencia usando como ejemplos problemas pequeños. En consecuencia, son capaces de resolver con eficiencia problemas bien definidos, pero son sobrepasados por la complejidad de su primera experiencia de desarrollo real cuando es necesario usar muchas técnicas y herramientas diferentes y necesitan colaborar con personas diferentes. Como reacción a esta situación, el plan de estudios típico de los estudiantes de licenciatura incluye ahora, a menudo, un curso de proyectos de ingeniería de software organizado como un solo proyecto de desarrollo.

Las herramientas: UML, Java y patrones de diseño

Escribimos este libro con un curso de proyectos en mente. Sin embargo, este libro también puede usarse en otras situaciones, como talleres cortos o intensivos, o proyectos de investigación y desarrollo de corto plazo. Usamos ejemplos de sistemas reales y examinamos la interacción entre las técnicas más actuales, como lenguaje de modelado unificado (UML, por sus siglas en inglés), tecnologías basadas en Java, patrones de diseño, fundamentación del diseño, administración de la configuración y control de calidad. Además tratamos temas relacionados con la administración de proyectos que están relacionados con estas técnicas y su impacto en la complejidad y el cambio.

Los principios

Enseñamos ingeniería de software siguiendo cinco principios:

Experiencia práctica. Creemos que la educación de ingeniería de software debe estar vinculada con la experiencia práctica. La comprensión de la complejidad sólo puede obtenerse trabajando con un sistema complejo; esto es, un sistema que no puede comprender por completo un solo estudiante.

Resolución de problemas. Creemos que la educación de ingeniería de software debe estar basada en la resolución de problemas. En consecuencia, no hay soluciones correctas o equivocadas, sino sólo soluciones que son mejores o peores con relación a un criterio establecido. Aunque examinamos las soluciones existentes a problemas reales y recomendamos su reutilización, también recomendamos la crítica y mejora de las soluciones estándar.

Recursos limitados. Si tuviéramos suficiente tiempo y recursos tal vez podríamos construir el sistema ideal. Hay varios problemas en esta situación. Primero, no es realista. Segundo, aunque tuviéramos suficientes recursos, si el problema original cambia con rapidez durante el desarrollo podríamos entregar al final un sistema que resuelve el problema equivocado. En consecuencia, suponemos que nuestro proceso de resolución de problemas está limitado respecto a los recursos. Además, la conciencia de la escasez de recursos motiva un enfoque basado en componentes, reutilización de software, diseño y código. En otras palabras, apoyamos un enfoque de ingeniería para el desarrollo de software.

Interdisciplinariedad. La ingeniería de software es un campo interdisciplinario. Requiere contribuciones de áreas que abarcan la ingeniería eléctrica y de computación, la ciencia de la computación, la administración de empresas, el diseño gráfico, el diseño industrial, la arquitectura, el teatro y la redacción. La ingeniería de software es un campo aplicado. Cuando se trata de comprender y modelar el dominio de aplicación, los desarrolladores interactúan en forma regular con otros, incluyendo usuarios y clientes, quienes a veces saben muy poco acerca del desarrollo de software. Esto requiere que se vea y ataque el sistema desde varias perspectivas y terminologías.

Comunicación. Aunque los desarrolladores construyeran software sólo para desarrolladores, todavía tendrían que comunicarse entre ellos mismos. Como desarrolladores no podemos darnos el lujo de sólo poder comunicarnos con nuestros iguales. Necesitamos comunicar alternativas, articular soluciones, negociar compromisos y revisar y criticar el trabajo de los demás. Una gran cantidad de fallas en los proyectos de ingeniería de software puede deberse a la comunicación imprecisa de la información o a la falta de información. Debemos aprender a comunicarnos con todos los participantes en el proyecto, incluyendo, en forma muy importante, al cliente y los usuarios finales.

Estos cinco principios son la base de este libro. Motivan y permiten que el lector ataque problemas complejos y cambiantes con soluciones prácticas y más actuales.

El libro

Este libro está basado en técnicas orientadas a objetos aplicadas a la ingeniería de software. No es un libro de ingeniería de software general que investiga todos los métodos disponibles ni un libro de programación acerca de algoritmos y estructuras de datos. En vez de ello nos enfocamos en un conjunto limitado de técnicas y explicamos su aplicación en un ambiente razonablemente complejo, como un proyecto de desarrollo de varios equipos que incluya de 20 a 60 participantes. En consecuencia, este libro también refleja nuestras predisposiciones, nuestras virtudes y nuestros defectos. Sin embargo, esperamos que todos los lectores encuentren algo que puedan usar. El libro está estructurado en 12 capítulos organizados en cuatro partes, que pueden enseñarse en un curso de un semestre.

La parte I, *Comenzando*, incluye tres capítulos. En esta parte nos enfocamos en las herramientas básicas que necesita un desarrollador para funcionar en un contexto de ingeniería de software.

- En el capítulo 1, *Introducción a la ingeniería de software*, describimos la diferencia entre la programación y la ingeniería de software, los retos actuales en nuestra disciplina y definiciones básicas de los conceptos que usaremos a lo largo del libro.
- En el capítulo 2, *Modelado con UML*, describimos los elementos básicos de un lenguaje de modelado, lenguaje de modelado unificado (UML), que se usa en las técnicas orientadas a objetos. Presentamos el modelado como una técnica para manejar la complejidad. Este capítulo le enseña al lector la manera de leer y comprender los diagramas UML. Capítulos subsiguientes le enseñan al lector la manera de construir diagramas UML para modelar diversos aspectos del sistema. Usamos UML a lo largo del libro para modelar diversos artefactos, desde sistemas de software hasta procesos y productos de trabajo.
- En el capítulo 3, *Comunicación de proyectos*, tratamos la actividad más crítica que realizan los desarrolladores. Los desarrolladores y administradores ocupan más de la mitad de su tiempo comunicándose con otros, ya sea en persona o por medio de correo electrónico, groupware, videoconferencias o documentos escritos. Mientras el modelado trata con la complejidad, la comunicación trata con el cambio. Describimos los medios principales de comunicación y su aplicación, y tratamos lo que constituye una comunicación efectiva.

En la parte II, *Manejo de la complejidad*, nos enfocamos en métodos y tecnologías que permiten que los desarrolladores especifiquen, diseñen e implementen sistemas complejos.

- En el capítulo 4, *Obtención de requerimientos*, y en el capítulo 5, *Análisis*, describimos la definición del sistema desde el punto de vista de los usuarios. Durante la obtención de requerimientos, los desarrolladores determinan la funcionalidad que necesitan los usuarios y una forma útil para proporcionarla. Durante el análisis, los desarrolladores formalizan este conocimiento y aseguran que esté completo y sea consistente. Nos enfocamos en la manera en que se usa UML para manejar la complejidad del dominio de aplicación.

- En el capítulo 6, *Diseño del sistema*, describimos las definiciones del sistema desde el punto de vista del desarrollador. Durante esta fase, los desarrolladores definen la arquitectura del sistema en función de objetivos de diseño y descomposición en subsistemas. Tratan asuntos globales como la correspondencia del sistema con el hardware, el almacenamiento de datos persistentes y el flujo de control global. Nos enfocamos en la manera en que los desarrolladores pueden usar patrones de diseño, componentes y UML para manejar la complejidad del dominio de solución.
- En el capítulo 7, *Diseño de objetos*, describimos el modelado detallado y las actividades de construcción que están relacionadas con el dominio de solución. Refinamos los requerimientos y el modelo del sistema, especificamos con precisión las clases que constituyen el sistema y definimos la frontera de la biblioteca de clases existentes y marcos de trabajo. Para la especificación de las interfaces de clases usamos el lenguaje de restricción de objetos de UML.

En la parte III, *Manejo del cambio*, nos enfocamos en métodos y tecnologías que apoyan el control, la valoración e implementación de los cambios a lo largo del ciclo de vida.

- En el capítulo 8, *Administración de la fundamentación*, describimos la captura de las decisiones de diseño y su justificación. Los modelos que desarrollamos durante la obtención de requerimientos, el análisis y el diseño del sistema nos ayudan a manejar la complejidad, proporcionándonos diferentes perspectivas sobre *qué* debe hacer el sistema y *cómo* debe hacerlo. Para poder manejar bien el cambio también necesitamos conocer *por qué* el sistema es de la forma que es. La captura de las decisiones de diseño, las alternativas evaluadas y su argumentación nos permiten el acceso a la fundamentación del sistema.
- En el capítulo 9, *Pruebas*, describimos la validación del comportamiento del sistema contra los modelos del sistema. Las pruebas detectan fallas en el sistema, incluyendo aquellas que se introducen durante los cambios al sistema o a sus requerimientos. Las actividades de prueba incluyen la prueba unitaria, la prueba de integración y la prueba del sistema. Describimos varias técnicas de prueba, como caja blanca, caja negra, prueba de ruta basada en estado e inspecciones.
- En el capítulo 10, *Administración de la configuración del software*, describimos técnicas y herramientas para el modelado de la historia del proyecto. La administración de la configuración complementa la fundamentación para ayudarnos a manejar el cambio. La administración de versiones registra la evolución del sistema. La administración del lanzamiento asegura la consistencia y calidad a través de los componentes de una publicación. La administración del cambio asegura que las modificaciones al sistema sean consistentes con los objetivos del proyecto.
- En el capítulo 11, *Administración del proyecto*, describimos las técnicas necesarias para iniciar un proyecto de desarrollo de software, llevar cuenta de su avance y manejar los riesgos y eventos no planeados. Nos enfocamos en las actividades de organización, papeles y administración que permiten que un gran número de participantes colaboren y entreguen un sistema de alta calidad dentro de restricciones planeadas.

En la parte IV, *Vuelta a empezar*, se revisan los conceptos que describimos en los capítulos anteriores desde una perspectiva del proceso. En el capítulo 12, *Ciclo de vida del software*, se

describen los ciclos de vida del software, como el modelo espiral de Boehm y el proceso de desarrollo de software unificado, que proporciona un modelo abstracto de actividades de desarrollo. En este capítulo también describimos el modelo de madurez de capacidades, que se usa para valorar la madurez de las organizaciones. Concluimos con dos ejemplos de ciclo de vida del software que pueden aplicarse en un proyecto de clase.

Los temas mencionados anteriormente están muy interrelacionados. Para enfatizar sus relaciones seleccionamos un enfoque iterativo. Cada capítulo consta de cinco secciones. En la primera sección presentamos los asuntos relevantes del tema con un ejemplo ilustrativo. En la segunda sección describimos en forma breve las actividades del tema. En la tercera sección explicamos los conceptos básicos del tema con ejemplos simples. En la cuarta sección detallamos las actividades técnicas con ejemplos de sistemas reales. Por último, describimos las actividades administrativas y tratamos los compromisos típicos. Al repetir y detallar los mismos conceptos usando ejemplos cada vez más complejos, esperamos proporcionar al lector un conocimiento operacional de la ingeniería de software orientado a objetos.

Los cursos

Escribimos este libro para un curso de proyectos de ingeniería de software de un semestre para estudiantes de último año o graduados. Suponemos que los estudiantes ya tienen experiencia con lenguajes de programación como C, C++, Ada o Java. Esperamos que los estudiantes tengan las habilidades de solución de problemas necesarias para atacar problemas técnicos, pero no esperamos que hayan estado expuestos a situaciones complejas o cambiantes típicas del desarrollo de sistemas. Sin embargo, este libro también puede usarse para otros tipos de cursos, como cursos profesionales, intensivos y cortos.

Cursos en un nivel de proyecto y para graduación. Un curso de proyectos debe incluir todos los capítulos del libro casi en el mismo orden. Un instructor puede considerar enseñar al principio del curso los conceptos introductorios de administración de proyectos del capítulo 11, *Administración del proyecto*, para que de esta forma los estudiantes estén familiarizados con la planeación y los reportes de estado.

Curso introductorio. Un curso introductorio con tareas en casa debe enfocarse en las tres primeras secciones de cada capítulo. La cuarta sección puede usarse como material para trabajar en casa, y puede simularse la construcción de un sistema pequeño usando papel para los diagramas UML, documentos y código.

Curso técnico corto. Este libro también puede usarse para un curso corto e intensivo dirigido hacia profesionales. Un curso técnico enfocado en UML y métodos orientados a objetos puede usar la secuencia de capítulos 1, 2, 4, 5, 6, 7, 8 y 9, cubriendo todas las fases de desarrollo desde la obtención de requerimientos hasta las pruebas. Un curso avanzado podría incluir también el capítulo 10, *Administración de la configuración del software*.

Curso administrativo corto. Este libro también puede usarse para un curso intensivo corto orientado hacia administradores. Un curso de administración enfocado en aspectos administrativos, como la comunicación, la administración de riesgos, la fundamentación, los modelos maduros y UML, podría usar la secuencia de capítulos 1, 2, 11, 3, 4, 8, 10 y 12.

Acerca de los autores

El Dr. Bernd Bruegge ha estudiado y enseñado ingeniería de software en la Universidad Carnegie Mellon durante 20 años, en donde obtuvo su maestría y doctorado. Recibió su diploma en la Universidad de Hamburgo. Ahora es profesor universitario de ciencias de la computación con una cátedra de ingeniería de software aplicada en la Universidad Técnica de Munich y es miembro adjunto del cuerpo docente de la Universidad Carnegie Mellon. Durante 10 años ha enseñado cursos de proyectos de ingeniería de software orientado a objetos con los materiales del texto y el sitio Web descrito en este libro. Ganó el premio a la excelencia en la enseñanza Herbert A. Simmon en la Universidad Carnegie Mellon en 1995. Bruegge también es consultor internacional y ha usado las técnicas de este libro para diseñar e implementar muchos sistemas reales que incluyen: un sistema de retroalimentación de ingeniería para Daimler-Chrysler, un sistema de modelado ambiental para el E.P.A. y un sistema para administración de accidentes para un departamento de policía municipal, por nombrar unos cuantos.

El Dr. Allen H. Dutoit es un investigador científico en la Universidad Técnica de Munich. Obtuvo su maestría y doctorado en la Universidad Carnegie Mellon y su diploma de ingeniero en el Instituto Federal Suizo de Tecnología en Lausana. Ha enseñado cursos de proyectos de ingeniería de software con el profesor Bruegge desde 1993, tanto en la Universidad Carnegie Mellon como en la Universidad Técnica de Munich, en donde usaron y refinaron los métodos descritos en este libro. La investigación de Dutoit cubre varias áreas de ingeniería de software y sistemas orientados a objetos, incluyendo administración del conocimiento, administración de la fundamentación, soporte de decisiones distribuido y sistemas basados en prototipos. Antes perteneció al Instituto de Ingeniería de Software y al Instituto de Sistemas de Ingeniería Compleja en la Universidad Carnegie Mellon.

Contenido

Agradecimientos

Este libro ha atestiguado mucha complejidad y cambio durante su desarrollo. En 1989, el primer autor comenzó a enseñar ingeniería de software con un formato de un curso de un solo proyecto. El objetivo era exponer a los estudiantes los asuntos importantes de la ingeniería de software resolviendo problemas reales, descritos por clientes reales, con herramientas reales, bajo restricciones reales. El primer curso, listado como 15-413 en el catálogo de cursos de Carnegie Mellon, tuvo 19 estudiantes, usó SA/SD como metodología de desarrollo y produjo 4,000 líneas de código. Desde entonces hemos probado en forma satisfactoria muchos métodos, herramientas y notaciones (por ejemplo, OMT, OOSE, UML). En la actualidad estamos enseñando una versión distribuida del curso que involucra hasta 80 estudiantes de Carnegie Mellon y la Universidad Técnica de Munich, teniendo como resultado sistemas con hasta 500 páginas de documentación y 50,000 líneas de código.

La desventaja de los cursos de proyectos es que los instructores no escapan a la complejidad ni al cambio que experimentan sus estudiantes. Los instructores se convierten con rapidez en participantes del desarrollo, actuando a menudo como gerentes de proyecto. Esperamos que este libro ayude tanto a los instructores como a los estudiantes a conquistar este nivel de complejidad y cambio.

De alguna forma, a pesar del gran gasto de energía en el curso, hemos encontrado tiempo para escribir y terminar este libro de texto gracias a la ayuda y paciencia de muchos estudiantes, colegas, asistentes de enseñanza, personal de soporte, colegas instructores, revisores, personal de Prentice Hall y, sobre todo, de nuestras familias. Algunos han contribuido para mejorar el curso, otros han proporcionado retroalimentación constructiva en borradores sucesivos y otros más simplemente estuvieron allí cuando las cosas se pusieron difíciles. A lo largo de los últimos 10 años, hemos quedado en deuda con muchas personas a las cuales damos aquí nuestro agradecimiento.

A los participantes de Workstation Fax (1989) Mark Sherman (cliente). Keys Botzum, Keith Chapman, Curt Equi, Chris Fulmer, Dave Fulmer, James Gilbert, Matt Jacobus, John Kalucki, David Kosmal, Jay Libove, Chris Maloney, Stephen Mahoney, Bryan Schmersal, Rich Segal, Jeff Shufelt, Stephen Smith, Marco Urbanic, Chris Warner y Tammy White.

A los participantes de Interactive Maps (1991) David Garlan (cliente). Jeff Alexander, Philip Bronner, Tony Brusseau, Seraq Eldin, Peter French, Doug Ghormley, David Gillen, Mike Ginsberg, Mario Goertzel, Tim Gottschalk, Susan Knight, Bin Luo, Mike Mantarro, Troy Roysedorph y Stephen Sadowsky.

A los participantes de Interactive Pittsburgh (1991) Charles Baclawski, Ed Wells y David Wild (clientes). Jason Barshay, Jon Bennett, Jim Blythe, Brian Bresnahan, John Butare, Jon Caron, Michael Cham, Ross Comer, Eric Ewanco, Karen Fabrizius, Kevin Gallo, Ekard Ginting, Stephen Gifford, Kevin Goldsmith, Jeff Jackson, Gray Jones, Chris Kirby, Brian Kircher, Susan Knight, Jonathan Levy, Nathan Loofbourrow, Matthew Lucas, Benjamin McCurtain, Adam Nemitoff, Lisa Nush, Sean O'Brien, Chris Paris, David Patrick, Victoria Pickard, Jeon Rezvani, Erik Riedel, Rob Ryan, Andy Segal y David VanRyzin.

A los participantes de FRIEND (1992, 1993, 1994) Jefe Michael Bookser (cliente). Soma Agarwal, Eric Anderson, Matt Bamberger, Joe Beck, Scott Berkun, Ashish Bisarya, Henry Borysewicz, Kevin Boyd, Douglas Brashear, Andrew Breen, James Brown, Kevin Chen, Li-Kang Chen, Daniel Cohn, Karl Crary, Dan Dalal, Laurie Damianos, Mark Delsesto, Court Demas, Carl Dockham, Ezra Dreisbach, Jeff Duprey, Tom Evans, Anja Feldman, Daniel Ferrel, Steve Fink, Matt Flatt, Pepe Galmes, Zafrir Gan, Steven Gemma, Dave Gillespe, Jon Gillaspie, Shali Goradia, Rudolf Halac, John Henderson, Brian Hixon, Mike Holling, Zach Hraber, Thomas Hui, Asad Iqbal, Sergey Iskotz, Isam Ismail, Seth Kadesh, Parlin Kang, George Kao, Paul Karlin, Shuntaro Kawakami, Dexter Kobayashi, Todd Kulick, Stephen Lacy, Jay Laefer, Anna Lederman, Bryan Lewis, Wendy Liau, Ray Lieu, Josephene Lim, Edward Liu, Kenneth Magnes, Patrick Magruder, Stephanie Masumura, Jeff Mishler, Manish Modh, Thomas Mon, Bill Nagy, Timothy Nali, Erikas Napjus, Cuong Nguyen, Kevin O'Toole, Jared Oberhaus, Han Ming Ong, David Pascua, David Pierce, Asaf Ronen, David Rothenberger, Sven Schiller, Jessica Schmidt, Hollis Schuller, Stefan Sherwood, Eric Snider, Paul Sonier, Naris Siamwalla, Art Sierzputowski, Patrick Soo Hoo, David Stager, Wilson Swee, Phil Syme, Chris Taylor, James Uzmack, Kathryn van Stone, Rahul Verma, Minh Vo, John Wiedey, Betty Wu, Bobby Yee, Amit Zavery y Jim Zelenka.

A los participantes de JEWEL, GEMS (1991, 1994, 1995) Greg McRay, Joan Novak, Ted Russel y James Wilkinson (clientes). William Adelman, Robert Agustin, Ed Allard, Syon Bhattacharya, Dan Bothell, Kin Chan, Huifen Chan, Yuzong Chang, Zack Charmoy, Kevin Chea, Kuo Chiang Chiang, Peter Chow, Lily Chow, Joe Concelman, Gustavo Corral, Christopher Ekberg, David Fogel, Bill Frank, Matthew Goldberg, Michael Halperin, Samuel Helwig, Ben Holz, Jonathan Homer, Sang Hong, Andrew Houghton, Dave Irvin, James Ivers, Christopher Jones, Harry Karatassos, Kevin Keck, Drew Kompanek, Jen Kirstein, Jeff Kurtz, Heidi Lee, Jose Madriz, Juan Mata, Paul McEwan, Sergio Mendiola, David Mickelson, Paul Mitchell, Jonathan Moore, Sintha Nainggolan, Donald Nelson, Philip Nemec, Bill Ommert, Joe Pekny, Adrian Perrig, Mark Pollard, Erik Riedel, Wasinee Rungsarityotin, Michael Sarnowski, Rick Shehady, Hendrick Sin, Brian

Solganick, Anish Srivastava, Jordan Tsvetkoff, Gabriel Underwood, Kip Walker, Alex Wetmore, Peter Wieland, Marullus Williams, Yau Sheng, Mark Werner y David Yu.

A los participantes de DIAMOND (1995, 1996) Bob DiSilvestro y Dieter Hege (clientes). Tito Benitez, Bartos Blacha, John Chen, Seth Covitz, Chana Damarla, Dmitry Dakhnovsky, Sunanda Dasai, Xiao Gao, Murali Haran, Srinivas Inguva, Joyce Johnstone, Chang Kim, Emile Litvak, Kris McQueen, Michael Peck, Allon Rauer, Stephan Schoenig, Ian Schreiber, Erik Siegel, Ryan Thomas, Hong Tong, Todd Turco, A. J. Whitney y Hiyung Yu.

A los participantes de OWL (1996, 1997) Volker Hartkopf (cliente). Paige Angstadt, Ali Aydar, David Babbitt, Neeraj Bansal, Jim Buck, Austin Bye, Seongju Chang, Adam Chase, Roberto DeFeo, Ravi Desai, Kelly DeYoe, John Dorsey, Christopher Esko, Benedict Fernandes, Truman Fenton, Ross Fubini, Brian Gallew, Samuel Gerstein, John Gillies, Asli Gulcur, Brian Hutsell, Craig Johnson, Tim Kniveton, John Kuhns, Danny Kwong, DeWitt Latimer, Daniel List, Brian Long, Gregory Mattes, Ceri Morgan, Jeff Mueller, Michael Nonemacher, Chris O'Rourke, Iroro Orife, Victor Ortega, Philipp Oser, Hunter Payne, Justus Pendleton, Ricardo Pravia, Robert Raposa, Tony Rippy, Misha Rutman, Trevor Schadt, Aseem Sharma, Mark Shieh, Caleb Sidel, Mark Silverman, Eric Stein, Eric Stuckey, Syahrul Syahabuddin, Robert Trace, Nick Vallidis, Randon Warner, Andrew Willis, Laurence Wong y Jack Wu.

A los participantes de JAMES (1997, 1998) Brigitte Pihulak (cliente). Malcolm Bauer, Klaus Bergner, Reinhold Biedermann, Brian Cavalier, Gordon Cheng, Li-Lun Cheng, Christopher Chiappa, Arjun Cholkar, Uhyon Chung, Oliver Creighton, Aveek Datta, John Doe, Phillip Ezolt, Eric Farng, William Ferry, Maximilian Fischer, Luca Girado, Thomas Gozolits, Alfonso Guerrero-Galan, Sang Won Ham, Kevin Hamlen, Martin Hans, Pradip Hari, Russel Heywood, Max Hoefner, Michael Karas, Yenni Kwek, Thomas Letsch, Tze Bin Loh, Alexander Lozupone, Christopher Lumb, Vincent Mak, Darren Mauro, Adam Miklaszewicz, Hoda Moustapha, Gerhard Mueller, Venkatesh Natarajan, Dick Orgass, Sam Perman, Stan Pavlik, Ralf Pfleghar, Marek Polrolniczak, Michael Poole, Wolfgang Popp, Bob Poydence, Kalyana Prattipati, Luis Rico-Gutierrez, Andreas Rausch, Thomas Reicher, Michael Samuel, Michael Scheinholtz, Marc Sihling, Joel Slovacek, Ann Sluzhevsky, Marc Snyder, Steve Sprang, Paul Stadler, Herbert Stiel, Martin Stumpf, Patrick Toole, Isabel Torres-Yebra, Christoph Vilsmeier, Idan Waisman, Aaron Wald, Andrew Wang, Zhongtao Wang, Ricarda Weber, Pawel Wiktorza, Nathaniel Woods, Jaewoo You y Bin Zhou.

A los participantes de PAID (1998, 1999) Helmut Ritzer y Richard Russ (clientes). Ralf Acker, Luis Alonso, Keith Arner, Bekim Bajraktari, Elizabeth Bigelow, Götz Bock, Henning Burdack, Orly Canlas, Igor Chernyavskiy, Jörg Dolak, Osman Durrani, John Feist, Burkhard Fischer, David Garmire, Johannes Gramsch, Swati Gupta, Sameer Hafez, Tom Hawley, Klaas Hermanns, Thomas Hertz, Jonathan Hsieh, Elaine Hyder, Florian Klaschka, Jürgen Knauth, Guido Kraus, Stefan Krause, James Lampe, Robin Loh, Dietmar Matzke, Florian Michahelles, Jack Moffett, Yun-Ching Lee, Wing Ling Leung, Andreas Löhr, Fabian Loschek, Michael Luber, Kent Ma, Asa MacWilliams, Georgios Markakis, Richard Markwart, Dan McCarriar, Istvan Nagy, Reynald Ong, Stefan Oprea, Adam Phelps, Arnaldo Piccinelli, Euijung Ra, Quiang Rao, Stefan Riss, William Ross, Pooja Saksena, Christian Sandor, Johannes Schmid, Ingo Schneider, Oliver Schnier, Florian Schönherr, Gregor Schraegle, Rudy Setiawan, Timothy Shirley, Michael Smith, Eric Stein,

Daniel Stodden, Anton Tichatschek, Markus Tönnis, Ender Tortop, Barrett Trask, Ivan Tumanov, Martin Uhl, Bert van Heukelkom, Anthony Watkins, Tobias Weishäupl, Marko Werner, Jonathan Wildstrom, Michael Winter, Brian Woo, Bernhard Zaun, Alexander Zeilner, Stephane Zermatten y Andrew Zimdars.

A las personas que apoyaron los proyectos por su compromiso, su amabilidad y por resolvernos los problemas cuando lo necesitamos: Catherine Copetas, Oliver Creighton, Ava Cruse, Barry Eisel, Dieter Hege, Joyce Johnstone, Luca Girardo, Monika Markl, Pat Miller, Ralf Pfleghar, Barbara Sandling, Ralph Schiessl, Arno Schmackpfeffer y Stephan Schoenig.

A los colegas, compañeros instructores y amigos que influyeron en nosotros con ideas e innumerables horas de apoyo: Mario Barbacci, Len Bass, Ben Bennington, Elizabeth Bigelow, Roberto Bisiani, Harry Q Bovik, Sharon Burks, Marvin Carr, Mike Collins, Robert Coyne, Douglas Cunningham, Michael Ehrenberger, Kim Faught, Peter Feiler, Allen Fisher, Laura Forsyth, Eric Gardner, Helen Granger, Thomas Gross, Volker Hartkopf, Bruce Horn, David Kauffer, Kalyka Konda, Suresh Konda, Rich Korf, Birgitte Krogh, Sean Levy, K. C. Marshall, Dick Martin ("Tang Soo"), Horst Mauersberg, Roy Maxion, Russ Milliken, Ira Monarch, Rhonda Moyer, Robert Patrick, Mark Pollard, Martin Purvis, Raj Reddy, Yoram Reich, James Rumbaugh, Johann Schlichter, Mary Shaw, Jane Siegel, Daniel Siewiorek, Asim Smailagic, Mark Stehlik, Eswaran Subrahmanian, Stephanie Szakal, Tara Taylor, Michael Terk, Günter Teubner, Marc Thomas, Jim Tomayko, Blake Ward, Alex Waibel, Art Westerberg y Jeannette Wing.

A los revisores que nos dieron retroalimentación constructiva y que nos ayudaron a corregir muchos detalles: Martin Barret, Thomas Eichhorn, Henry Etlinger, Ray Ford, Gerhard Mueller, Barbara Paech, Joan Peckham, Ingo Schneider y Eswaran Subrahmanian. Todos los errores restantes son nuestros.

A todo el personal de Prentice Hall que nos ayudó para que este libro fuera una realidad, en particular Alan Apt, nuestro editor, por no haber perdido nunca la fe; Sondra Scott y Ana Terry, nuestros gerentes de proyecto, por llevar cuenta del millón de problemas asociados con este proyecto y facilitar su solución; Eileen Clark, nuestra gerente de producción, Ann Marie Kalajian y Scott Disanno, nuestros editores de producción, y Heather Scott, nuestro director artístico, por diseñar una portada de libro e inicios de capítulo elegantes y atractivos, y por hacer que muchas cosas maravillosas e imposibles sucedieran a tiempo; Joel Berman, Toni Holm, Eric Weisser y muchos otros que trabajaron mucho para la terminación de este libro pero que no tuvimos la oportunidad y el placer de conocer en persona.

Y por último a nuestras familias, a quienes les dedicamos este libro y sin cuyo infinito amor y paciencia la empresa nunca habría sido posible.

PARTE I
Comenzando

1

1

Introducción a la ingeniería de software

El ingeniero de software aficionado siempre está a la búsqueda de algo mágico, algún método o herramienta sensacional que prometa convertir el desarrollo de software en algo trivial. Está en el ingeniero de software profesional saber que no existe tal panacea.

—Grady Booch, *en Object-Oriented Analysis and Design*

El término ingeniería de software fue acuñado en 1968 como una respuesta al nivel de progreso desolador del objetivo de desarrollar software de calidad a tiempo y dentro del presupuesto. Los desarrolladores de software no fueron capaces de definir objetivos concretos, predecir los recursos necesarios para lograr esos objetivos y manejar las expectativas de los clientes. Con mucha frecuencia se prometía la luna y la construcción de un vehículo lunar, y se entregaba un par de ruedas cuadradas.

El énfasis de la ingeniería de software está en las dos palabras: *ingeniería* y *software*. Un ingeniero es capaz de construir un producto de alta calidad usando componentes ya elaborados e integrándolos bajo restricciones de tiempo y presupuesto. El ingeniero se enfrenta a menudo con problemas mal definidos y con soluciones parciales, y tiene que apoyarse en métodos empíricos para evaluar soluciones. Los ingenieros que trabajan en campos de aplicación como el diseño de aeronaves de pasajeros y la construcción de puentes han resuelto en forma satisfactoria retos similares. Los ingenieros de software no han tenido tanto éxito.

Se ha investigado de manera activa el problema de construir y entregar a tiempo sistemas de software complejos. Se ha culpado a todo, desde el cliente ("¿qué quiere decir usted con que no puedo obtener la luna por $50?") hasta lo "blando" en software ("si pudiera añadir una última característica...") y la juventud de esta disciplina. ¿Cuál es el problema?

Complejidad y cambio.

Los sistemas de software útiles son complejos. Para seguir siendo útiles necesitan evolucionar con las necesidades de los usuarios finales y el ambiente de destino. En este libro describimos técnicas orientadas a objetos para conquistar sistemas de software complejos y cambiantes. En este capítulo proporcionamos una motivación para las técnicas orientadas a objetos y definimos los conceptos básicos que se usan a lo largo de este libro.

1.1 Introducción: fallas de la ingeniería de software

Considere los siguientes ejemplos [Neumann, 1995]:

El error del año 1900

En 1992, Mary, de Winona, Minnesota, recibió una invitación para que asistiera a un jardín de niños. En ese entonces Mary tenía 104 años.

Error del año bisiesto

A un supermercado le pusieron una multa de 1,000 dólares por tener carne que había caducado por un día, el 29 de febrero de 1988. El programa de computadora que imprimió la fecha de caducidad en las etiquetas de la carne no tomó en cuenta que 1988 era año bisiesto.

Mal uso de interfaz

El 10 de abril de 1990, en Londres, un tren subterráneo salió de la estación sin su conductor. El conductor había oprimido el botón que arrancaba el tren confiando en el sistema que impedía que el tren se moviera mientras las puertas estuvieran abiertas. El operador del tren había abandonado su lugar para cerrar una puerta que estaba atorada. Cuando finalmente la puerta se cerró, el tren simplemente se fue.

Seguridad

El 2 de noviembre de 1988, un programa que se autopropaga, al que posteriormente se le llamó el gusano de Internet, fue lanzado en Internet. El programa explotó la vulnerabilidad de los servicios de red, como la del programa de envío de correo de Unix, para replicarse a sí mismo de una computadora a otra. Por desgracia, al llegar el gusano de Internet a una máquina, consumía todos los recursos de cómputo disponibles y hacía que la máquina infectada dejara de funcionar. Se estima que se afectó 10% de todos los nodos de Internet. Se necesitaron varios días para erradicar la infección.

Con retraso y excedido en el presupuesto

En 1995 los errores en el sistema de manejo de equipaje automatizado del nuevo aeropuerto internacional de Denver causaron que se dañaran las maletas. El aeropuerto abrió con 16 meses de retraso, con un exceso de gasto de 3.2 mil millones de dólares y con un sistema para el manejo de equipaje en su mayor parte manual.

Entrega a tiempo

Después de 18 meses de desarrollo, se entregó un sistema de 200 millones de dólares a una compañía de seguros de salud en Wisconsin en 1984. Sin embargo, el sistema no funcionó en forma correcta: se expidieron 60 millones de dólares de pagos extras. Se necesitaron tres años para componer el sistema.

Complejidad innecesaria

El avión de carga C-17 de McDonnell Douglas se excedió 500 millones de dólares en el presupuesto a causa de problemas en su software de electrónica de aviación. El C-17 incluía 19 computadoras a bordo, 80 microprocesadores y seis lenguajes de programación diferentes.

Cada una de las fallas descritas antes se debió a un problema relacionado con software. En algunos casos los desarrolladores no anticiparon situaciones que ocurren rara vez (una persona que vive más de cien años, años bisiestos que tienen un impacto en las fechas de caducidad). En otros casos los desarrolladores no anticiparon que el usuario haría mal uso del sistema (la opresión de un botón, la exploración de las facilidades de depuración del envío de correo). En otros casos las fallas del sistema resultaron por fallas de administración (entrega tardía y con presupuesto excedido, entrega a tiempo de un sistema incorrecto, complejidad innecesaria).

Los sistemas de software son creaciones complejas: realizan muchas funciones, están construidos para lograr muchos objetivos diferentes y con frecuencia conflictivos, comprenden muchos componentes, muchos de sus componentes son en sí mismos complejos y hechos a la medida, muchos participantes de disciplinas diferentes intervienen en el desarrollo de estos componentes, el proceso de desarrollo y el ciclo de vida del software a menudo abarcan muchos años y, por último, los sistemas complejos son difíciles de comprender por completo para una sola persona. Muchos sistemas son tan difíciles de comprender, incluso durante su fase de desarrollo, que nunca llegan a ser terminados: a éstos se les llama vaporware.

Los proyectos de desarrollo de software están sujetos a cambios constantes. Debido a que los requerimientos son complejos, necesitan ser actualizados cuando se descubren errores y cuando los desarrolladores tienen una mejor comprensión de la aplicación. Si el proyecto dura muchos años, la rotación de personal es alta y requiere entrenamiento constante. El tiempo entre los cambios tecnológicos con frecuencia es más corto que la duración del proyecto. La suposición ampliamente difundida entre los gerentes de proyecto de software de que hay que abordar todos los cambios y que los requerimientos pueden congelarse, conduce a que se despliegue un sistema irrelevante.

En la siguiente sección presentamos una vista de alto nivel de la ingeniería de software. Describimos la ingeniería de software bajo la perspectiva de la ciencia, la ingeniería y la adquisición y formalización de conocimientos. En la sección 1.3 describimos con mayor detalle los términos y conceptos principales que usamos en este libro. En la sección 1.4 proporcionamos una panorámica de las actividades del desarrollo de la ingeniería de software. En la sección 1.5 damos una panorámica de las actividades administrativas de la ingeniería de software.

1.2 ¿Qué es la ingeniería de software?

La ingeniería de software es una actividad de **modelado**. Los ingenieros de software manejan la complejidad mediante el modelado, enfocándose siempre sólo en los detalles relevantes e ignorando todo lo demás. En el curso del desarrollo, los ingenieros de software construyen muchos modelos diferentes del sistema y del dominio de aplicación.

La ingeniería de software es una actividad para la **solución de problemas**. Se usan los modelos para buscar una solución aceptable. Esta búsqueda es conducida por la experimentación. Los ingenieros de software no tienen recursos infinitos, y están restringidos por presupuestos y tiempos de entrega. Dada la falta de una teoría fundamental, con frecuencia tienen que apoyarse en métodos empíricos para evaluar los beneficios de alternativas diferentes.

La ingeniería de software es una actividad para la **adquisición de conocimiento**. En el modelado de los dominios de la aplicación y la solución, el ingeniero de software recopila datos, los organiza en información y los formaliza en conocimiento. La adquisición de conocimiento no es lineal, ya que un solo dato puede invalidar modelos completos.

La ingeniería de software es una actividad **dirigida por una fundamentación**. Cuando se adquiere conocimiento y se toman decisiones acerca del sistema o sus dominios de aplicación, los ingenieros de software también necesitan captar el contexto en el que se tomaron las decisiones y las razones que hay tras las mismas. La información de la fundamentación, representada como un conjunto de modelos de problemas, permite que los ingenieros de software comprendan las implicaciones de un cambio propuesto cuando revisan una decisión.

En esta sección describimos con mayor detalle la ingeniería de software desde la perspectiva del modelado, la solución de problemas, la adquisición de conocimientos y la fundamentación. Para cada una de estas actividades, los ingenieros de software tienen que trabajar bajo restricciones de personal, tiempo y presupuesto. Además, suponemos que los cambios pueden suceder en cualquier momento.

1.2.1 Modelado

El propósito de la ciencia es describir y comprender sistemas complejos, como un sistema de átomos, una sociedad de seres humanos o un sistema solar. Por tradición se hace una distinción entre *ciencias naturales* y *ciencias sociales* para distinguir entre los dos tipos principales de sistemas. El propósito de las ciencias naturales es comprender la naturaleza y sus subsistemas. Las ciencias naturales incluyen, por ejemplo, biología, química, física y paleontología. El propósito de las ciencias sociales es comprender a los seres humanos. Las ciencias sociales incluyen psicología y sociología.

Hay otro tipo de sistemas a los que llamamos sistemas artificiales. Ejemplos de sistemas artificiales incluyen el transbordador espacial, los sistemas de reservación de boletos de avión y los sistemas de comercialización de acciones. Herbert Simon acuñó el término *ciencias de lo artificial* para describir las ciencias que tratan sobre los sistemas artificiales [Simon, 1970]. Mientras que las ciencias naturales y sociales han existido durante siglos, las ciencias de lo artificial son recientes. La ciencia de la computación, por ejemplo, la ciencia que trata de la comprensión de los sistemas de computadora, es una hija del siglo pasado.

Muchos métodos que se han aplicado de modo satisfactorio en las ciencias naturales y en las humanidades también pueden aplicarse en las ciencias de lo artificial. Observando las demás ciencias podemos aprender un poco. Uno de los métodos básicos de la ciencia es el modelado. Un modelo es una representación abstracta de un sistema que nos permite responder preguntas acerca del sistema. Los modelos son útiles cuando se manejan sistemas que son demasiado grandes, demasiado pequeños, demasiado complicados o demasiado caros para tener una experiencia de primera mano. Los modelos también nos permiten visualizar y comprender sistemas que ya no existen o que sólo se supone que existen.

Los biólogos dedicados a los fósiles desentierran unos cuantos huesos y dientes que se han conservado de algún dinosaurio que nadie ha visto. A partir de los fragmentos de hueso reconstruyen un modelo del animal siguiendo las reglas de la anatomía. Entre más huesos encuentran más clara es su idea de la manera en que se reúnen las piezas y mayor es la confianza de que su modelo concuerda con el dinosaurio original. Si encuentran una cantidad suficiente de huesos, dientes y garras, casi pueden estar seguros de que su modelo refleja la realidad con precisión y pueden hacer suposiciones sobre las partes faltantes. Las piernas, por ejemplo, por lo general vienen en pares. Si se encuentra la pierna izquierda, pero falta la derecha, el biólogo de fósiles tiene una idea bastante buena de cómo debe verse la pierna faltante y en qué parte del modelo encaja. Éste es un ejemplo de un modelo de un sistema que ya no existe.

Los físicos actuales dedicados a la alta energía están en una posición similar a la de los biólogos de fósiles que han encontrado la mayoría de los huesos. Los físicos están construyendo un modelo de la materia y la energía y la manera en que se reúnen en el nivel subatómico más básico. Su herramienta es el acelerador de partículas de alta energía. Muchos años de experimentación con los aceleradores de partículas han dado a los físicos de alta energía la suficiente confianza para pensar que sus modelos reflejan la realidad y que las piezas faltantes que todavía no han encon-

trado se acomodarán en el modelo llamado estándar. Éste es un ejemplo de un modelo para un sistema que se supone que existe.

Ambos modeladores de sistemas, los biólogos de fósiles y los físicos de alta energía, manejan dos tipos de entidades: el sistema del mundo real observado en función de un conjunto de fenómenos, y el modelo del dominio de problema representado como un conjunto de conceptos interdependientes. El sistema en el mundo real es un dinosaurio o partículas subatómicas. El modelo del dominio de problema es una descripción de aquellos aspectos del sistema del mundo real que son relevantes para el problema que se está considerando.

Los ingenieros de software enfrentan retos similares a los de los biólogos de fósiles y físicos de alta energía. Primero, los ingenieros de software necesitan comprender el ambiente en el que va a operar el sistema. Para un sistema de control de tráfico de trenes, los ingenieros de software necesitan conocer los procedimientos de señalización de los trenes. Para un sistema de comercialización de acciones, los ingenieros de software necesitan conocer las reglas de comercio. Los ingenieros de software no necesitan llegar a ser por completo despachadores de trenes certificados o corredores de bolsa, sino que sólo necesitan aprender los conceptos del dominio de problema que son *relevantes* para el sistema. En otras palabras, necesitan construir un modelo del dominio de problema.

Segundo, los ingenieros de software necesitan comprender los sistemas que podrían construir para evaluar diferentes soluciones y compromisos. La mayoría de los sistemas son demasiado complejos para que sean comprendidos por una sola persona y cuesta mucho construirlos. Para atacar estos retos, los ingenieros de software describen los aspectos importantes de los sistemas alternativos que investigan. En otras palabras, necesitan construir un modelo del dominio de solución.

Los métodos orientados a objetos combinan las actividades de modelado de los dominios de problema y de solución en uno solo. Primero se modela el dominio de problema como un conjunto de objetos y relaciones. Luego, el sistema usa este modelo para representar los conceptos del mundo real que manipula. Un sistema de control de tráfico de trenes incluye objetos de trenes que representan a los trenes que supervisa. Un sistema de comercialización de acciones incluye objetos de transacción que representan la compra y venta de acciones. Luego, también se modelan como objetos los conceptos del dominio de solución. El conjunto de líneas que se usan para representar un tren o una transacción son objetos que son parte del dominio de solución. La idea de los métodos orientados a objetos es que el modelo del dominio de solución es una extensión del modelo del dominio de problema. El desarrollo de software se traduce en las actividades necesarias para identificar y describir un sistema como un conjunto de modelos que abordan el problema del usuario final. En el capítulo 2, *Modelado con UML*, describimos con mayor detalle el modelado y el concepto de objetos.

1.2.2 Solución de problemas

La ingeniería es una actividad para la solución de problemas. Los ingenieros buscan una solución adecuada, a menudo mediante ensayo y error, evaluando alternativas en forma empírica con recursos limitados y con conocimiento incompleto. En su forma más simple, el método de la ingeniería incluye cinco pasos:

1. Formular el problema
2. Analizar el problema
3. Buscar soluciones

4. Decidir cuál es la solución adecuada
5. Especificar la solución

La ingeniería de software es una actividad de ingeniería. No es algorítmica. Requiere experimentación, la reutilización de soluciones patrón y la evolución creciente del sistema hacia una solución que sea aceptable para el cliente.

El desarrollo de software incluye, por lo general, cinco actividades de desarrollo: obtención de requerimientos, análisis, diseño del sistema, diseño de objetos e implementación. Durante la obtención de requerimientos y el análisis, los ingenieros de software formulan el problema junto con el cliente y construyen el modelo del dominio de problema. La obtención de requerimientos y el análisis corresponden a los pasos 1 y 2 del método de ingeniería. Durante el diseño del sistema los ingenieros de software analizan el problema, lo dividen en partes más pequeñas y seleccionan estrategias generales para el diseño del sistema. Durante el diseño de objetos seleccionan soluciones detalladas para cada parte y deciden la solución más adecuada. El diseño del sistema y el diseño de objetos dan como resultado el modelo del dominio de solución. Los diseños de sistema y objetos corresponden a los pasos 3 y 4 del método de ingeniería. Durante la implementación, los ingenieros de software realizan el sistema trasladando el modelo del dominio de solución hacia una representación ejecutable. La implementación corresponde al paso 5 del método de ingeniería. Lo que hace que la ingeniería de software sea diferente a la solución de problemas en otras ciencias es que los cambios suceden mientras se está resolviendo el problema.

El desarrollo de software también incluye actividades cuyo propósito es evaluar lo adecuado de los modelos respectivos. Durante la revisión de análisis, el modelo del dominio de aplicación se compara con la realidad del cliente, la cual, a su vez, puede cambiar como resultado del modelado. Durante la revisión del diseño se evalúa el modelo del dominio de solución contra los objetivos del proyecto. Durante las pruebas se valida el sistema contra el modelo del dominio de solución, el cual puede cambiar a causa de la introducción de nuevas tecnologías. Durante la administración del proyecto, los administradores comparan su modelo del proceso de desarrollo (es decir, la calendarización y presupuesto del proyecto) contra la realidad (es decir, los productos de trabajo entregados y los recursos gastados).

1.2.3 Adquisición de conocimiento

Un error común que cometen los ingenieros y gerentes de software es suponer que la adquisición del conocimiento necesario para desarrollar un sistema es lineal. Este error no sólo lo cometen los gerentes de software, sino también se le puede encontrar en otras áreas. En el siglo XVII se publicó un libro que ofrecía enseñar todos los poemas alemanes vertiéndolos con un embudo en la cabeza del estudiante en seis horas.[1] La idea de usar un embudo para el aprendizaje está basada en la suposición ampliamente difundida de que nuestra mente es una cubeta que al principio está vacía y que se puede llenar en forma lineal. El material entra a través de nuestros sentidos, se acumula y es digerido. Popper llama a este modelo de adquisición lineal del conocimiento "la teoría de la cubeta de la mente". Entre las muchas cosas que están equivocadas en esta teoría

1. G. P. Harsdoerfer (1607–1658) "Poetischer Trichter, die teutsche Dicht- und Reimkunst, ohn Behuf der lateinischen Sprache, in 6 Stunden einzugießen", Nuerenberg, 1630.

(descrita en [Popper, 1992]) está la suposición de que se concibe el conocimiento como consistente en cosas que pueden llenar una cubeta, y entre más llena esté la cubeta más sabemos.

La adquisición de conocimientos es un proceso no lineal. La adición de una nueva parte de información puede invalidar todo el conocimiento que hemos adquirido para la comprensión de un sistema. Aunque ya hubiéramos documentado esta comprensión en documentos y código ("el sistema está codificado 90% y lo terminaremos la semana próxima") debemos estar preparados mentalmente para comenzar a partir de cero. Esto tiene implicaciones importantes en el conjunto de actividades y sus interacciones que definimos para desarrollar el sistema de software. El equivalente de la teoría de la cubeta de la mente es el modelo de cascada lineal para el desarrollo de software, en donde todos los pasos del método de ingeniería se realizan en forma secuencial.

La falta de linealidad del proceso de adquisición de conocimiento tiene implicaciones severas en el desarrollo del sistema. Hay varios procesos de software que manejan este problema evitando las dependencias lineales inherentes al modelo de cascada. *El desarrollo basado en el riesgo* trata de anticipar sorpresas tardías en un proyecto mediante la identificación de los componentes de alto riesgo. *El desarrollo basado en problemas* también trata de eliminar la linealidad. Cualquier actividad de desarrollo, ya sea análisis, diseño del sistema, diseño de objetos, implementación, pruebas o entrega, puede influir en cualquier otra actividad. En el desarrollo basado en problemas todas estas actividades se ejecutan en paralelo. Sin embargo, el problema con los modelos de desarrollo que no son lineales es que son difíciles de manejar.

1.2.4 Administración de la fundamentación

Cuando describimos la adquisición o evolución del conocimiento, estamos mucho menos equipados que cuando describimos el conocimiento de un sistema existente. ¿Cómo deriva un matemático una prueba o demostración? Los libros de texto de matemáticas están llenos de pruebas, pero rara vez proporcionan pistas acerca de la derivación de pruebas. Esto se debe a que los matemáticos no creen que sea importante. Una vez que se han establecido los axiomas y las reglas de deducción, no tiene caso la prueba. Solo tendrá que ser revisada cuando cambien las suposiciones básicas o las reglas de deducción. En las matemáticas esto sucede muy rara vez. La aparición de geometrías no euclidianas 2000 años después de Euclides es un ejemplo. Las geometrías no euclidianas están basadas en los primeros cuatro axiomas de Euclides, pero suponen un axioma alterno en vez del quinto axioma de Euclides (dada una línea `l` y un punto `p` que no es parte de `l`, hay solamente una línea `l'` paralela a `l` que incluye el punto `p`). Posteriormente, Einstein usó una geometría no euclidiana en su teoría general de la relatividad. En astronomía, el cambio tampoco es un evento diario. Se necesitaron 1500 años para pasar del modelo geocéntrico del universo de Ptolomeo al modelo heliocéntrico del universo de Copérnico.

Para los ingenieros de software la situación es diferente. Las suposiciones que hacen los desarrolladores acerca de un sistema cambian en forma constante. Aunque los modelos del dominio de aplicación se estabilizan a la larga una vez que los desarrolladores adquieren una comprensión adecuada del problema, los modelos del dominio de solución están en flujo constante. Las fallas de diseño e implementación descubiertas durante las pruebas, y los problemas de utilización descubiertos durante la evaluación del usuario activan cambios a los modelos de solución. Los cambios también pueden ser causados por nuevas tecnologías. La disponibilidad de baterías de vida larga y las comunicaciones de gran ancho de banda, por ejemplo, pueden

provocar que se revisen los conceptos de una terminal portátil. El cambio introducido por nuevas tecnologías permite, con frecuencia, la formulación de nuevos requerimientos funcionales o no funcionales. Una tarea típica que deben realizar los ingenieros de software es cambiar un sistema operativo actualmente operacional para que incorpore esta nueva tecnología que lo permite. Para cambiar el sistema no es suficiente comprender sus componentes y comportamientos actuales. También es necesario capturar y comprender el contexto en el cual se tomó cada decisión de diseño. A este conocimiento adicional se le llama la **fundamentación** del sistema.

La captura y el acceso a la fundamentación de un sistema no es trivial. Primero, para cada decisión tomada se pueden haber considerado, evaluado y argumentado varias alternativas. En consecuencia, la fundamentación representa una cantidad de información mucho mayor que la que tienen los modelos de solución. Segundo, con frecuencia la información sobre la fundamentación no es explícita. Los desarrolladores toman muchas decisiones con base en su experiencia e intuición, sin evaluar de manera explícita diferentes alternativas. Cuando se pide que expliquen una decisión, puede ser que los desarrolladores tengan que pasar una buena cantidad de tiempo recuperando su fundamentación. Sin embargo, para manejar los sistemas cambiantes, los ingenieros de software deben enfrentar el reto de capturar y tener acceso a la fundamentación.

Hasta ahora hemos presentado una visión de alto nivel de la ingeniería de software desde las perspectivas del modelado, la solución de problemas, la adquisición de conocimiento y la fundamentación. En la siguiente sección definimos los principales términos y conceptos que usamos en el libro.

1.3 Conceptos de ingeniería de software

En esta sección describimos los conceptos principales que usamos a lo largo del libro.[2] Un `Proyecto`, cuyo propósito es desarrollar un sistema de software, está compuesto por varias `Actividades`. Cada `Actividad` está compuesta, a su vez, de varias `Tareas`. Una `Tarea` consume `Recursos` y origina un `Producto de Trabajo`. Un `Producto de Trabajo` puede ser un `Sistema`, un `Modelo` o un `Documento`. Los `Recursos` son `Participantes`, `Tiempo` o `Equipo`. En la figura 1-1 se muestra una representación gráfica de estos conceptos. Esta figura está representada en la notación del lenguaje de modelado unificado (UML, por sus siglas en inglés). Usamos UML a lo largo del libro para representar modelos de software y otros sistemas. Usted deberá ser capaz de comprender de manera intuitiva este diagrama sin tener un conocimiento completo de la semántica del UML. En forma similar, también puede usar diagramas UML cuando interactúe con un cliente o usuario, aunque tal vez él no tenga ningún conocimiento del UML. En el capítulo 2, *Modelado con UML*, describimos con detalle la semántica de estos diagramas.

2. En lo posible, seguimos las definiciones de las normas de la IEEE sobre la ingeniería de software [IEEE, 1997].

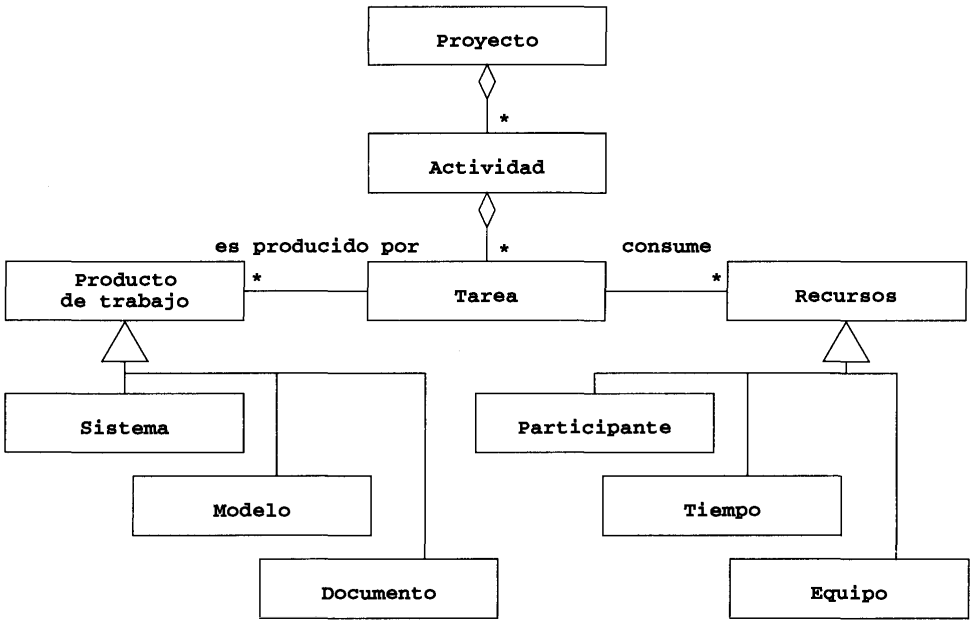


Figura 1-1 Conceptos de ingeniería de software mostrados como un diagrama de clase UML (OMG, 1998).

1.3.1 Participantes y papeles

El desarrollo de un sistema de software requiere la colaboración de muchas personas con diferentes formaciones o intereses. El cliente ordena y paga el sistema. Los desarrolladores construyen el sistema. El gerente del proyecto planea y calcula el presupuesto del proyecto y coordina a los desarrolladores y al cliente. Los usuarios finales son apoyados por el sistema. Nos referimos a todas estas personas que están involucradas en el proyecto como **participantes**. Nos referimos al conjunto de responsabilidades en el proyecto o en el sistema como un **papel**. Un papel está asociado con un conjunto de tareas y se asigna a un participante. El mismo participante puede cumplir varios papeles.

Considere una máquina que distribuye boletos para un tren. Los viajeros tienen la opción de seleccionar un boleto para un solo viaje, para varios viajes o una tarjeta temporal para un día o una semana. El distribuidor de boletos calcula el precio del boleto solicitado con base en el área en donde se realizará el viaje y si el viajero es niño o adulto. En este ejemplo, los viajeros que compran boletos son usuarios finales. La compañía de trenes que contrata el desarrollo del distribuidor de boletos es el cliente del proyecto. Los ingenieros que realizan el sistema (y el software), Juan, Marcos y Zoe son desarrolladores. Su jefa, Alicia, que coordina el trabajo y la comunicación con el cliente, es la gerente del proyecto. Usuario final, cliente, desarrollador y gerente de proyecto son papeles. Alicia, Juan, Marcos, Zoe, la compañía de trenes y los viajeros son participantes.

1.3.2 Sistemas y modelos

Usamos el término **sistema** para referirnos a la realidad subyacente, y el término **modelo** para referirnos a cualquier abstracción de la realidad. Un distribuidor de boletos para un tren subterráneo es un sistema. Los planos para el distribuidor de boletos, los esquemas de su alambrado eléctrico y los modelos de objetos de su software son modelos del distribuidor de boletos. Observe que un proyecto de desarrollo es, en sí mismo, un sistema que puede ser modelado. La calendarización del proyecto, su presupuesto y su tiempo de entrega planeado son modelos del proyecto de desarrollo.

1.3.3 Productos de trabajo

Un **producto de trabajo** es un artefacto que se produce durante el desarrollo, como un documento o un fragmento de software para los demás desarrolladores o para el cliente. Nos referimos a un producto de trabajo para el consumo interno del proyecto como **producto de trabajo interno**. Nos referimos a un producto de trabajo para un cliente como una **entrega**. Las entregas se definen, por lo general, antes del inicio del proyecto, y están especificadas en un contrato que enlaza a los desarrolladores con el cliente.

En el ejemplo del distribuidor de boletos, los manuales de operación y mantenimiento que necesita la compañía de trenes son entregas. Los distribuidores de boletos y su software también son entregas. Los prototipos de demostración, escenarios de prueba y resultados de prueba producidos por los desarrolladores para el gerente del proyecto son productos de trabajo internos, a menos que estén especificados en el contrato como artículos que serán entregados al cliente.

1.3.4 Actividades, tareas y recursos

Una **actividad** es un conjunto de tareas que se realiza con un propósito específico. Por ejemplo, la obtención de requerimientos es una actividad cuyo propósito es definir con el cliente lo que hará el sistema. La entrega es una actividad cuyo propósito es instalar el sistema en una ubicación operacional. La administración es una actividad cuyo propósito es supervisar y controlar el proyecto en forma tal que cumpla sus objetivos (por ejemplo, tiempo de entrega, calidad, presupuesto). Las actividades pueden estar compuestas por otras actividades. La actividad de entrega incluye la actividad de instalación del software y una actividad de entrenamiento del operador. A las actividades a veces también se les llama **fases**.

Una **tarea** representa una unidad atómica de trabajo que puede ser administrada: un gerente la asigna a un desarrollador, el desarrollador la realiza y el gerente supervisa el avance y terminación de la tarea. Las tareas consumen recursos, dan como resultado productos de trabajo y dependen de productos de trabajo que son producidos por otras tareas.

Los **recursos** son bienes que se usan para realizar el trabajo. Los recursos incluyen tiempo, equipo y mano de obra. Cuando se planea un proyecto, un gerente divide el trabajo en tareas y les asigna recursos.

1.3.5 Objetivos, requerimientos y restricciones

Un **objetivo** es un principio de alto nivel que se usa para guiar el proyecto. Los objetivos definen los atributos del sistema que son importantes. Proyectos diferentes tienen objetivos dife-

rentes. El objetivo principal del desarrollo del software de guía del transbordador espacial es producir un sistema que sea seguro (es decir, que no ponga en peligro la vida humana). El objetivo principal del distribuidor de boletos es producir un sistema que sea altamente confiable (es decir, que funcione en forma correcta la mayor parte del tiempo).

Los objetivos a menudo entran en conflicto, esto es, es difícil lograrlos en forma simultánea. Por ejemplo, la producción de un sistema seguro, como una aeronave de pasajeros, es cara. Sin embargo, los fabricantes de aeronaves también necesitan poner atención en el costo de venta de la aeronave, esto es, producir una aeronave que sea más barata que las de la competencia. La seguridad y el bajo costo son objetivos en conflicto. Una buena parte de la complejidad en el desarrollo de software viene de objetivos mal definidos o en conflicto.

Los requerimientos son características que debe tener el sistema. Un **requerimiento funcional** es un área de funcionalidad que debe soportar el sistema y, en cambio, un **requerimiento no funcional** es una restricción sobre la operación del sistema.

Por ejemplo, *proporcionar al usuario información sobre los boletos* es un requerimiento funcional. *Proporcionar retroalimentación en menos de un segundo* es un requerimiento no funcional. *Proporcionar un sistema confiable* es un objetivo de diseño. *Producir un sistema a bajo costo* es un objetivo gerencial. Otras restricciones incluyen que se requiera una plataforma de hardware especifica para el sistema o compatibilidad retrospectiva con un sistema heredado que el cliente no quiere retirar.

1.3.6 Notaciones, métodos y metodologías

Una **notación** es un conjunto de reglas gráficas o textuales para representar un modelo. El alfabeto romano es una notación para representar palabras. El UML (lenguaje de modelado unificado [OMG, 1998]), la notación que usamos a lo largo de este libro, es una notación orientada a objetos para la representación de modelos. Z [Spivey, 1989] es una notación para la representación de sistemas basada en la teoría de conjuntos.

Un **método** es una técnica repetible para la resolución de un problema específico. Una receta es un método para cocinar un plato específico. Un algoritmo de ordenamiento es un método para ordenar elementos de una lista. La administración de la fundamentación es un método para la justificación de los cambios. La administración de la configuración es un método para el seguimiento de los cambios.

Una **metodología** es una colección de métodos para la resolución de una clase de problemas. Un libro de cocina de mariscos es una metodología para la preparación de mariscos. El proceso de desarrollo de software unificado [Jacobson *et al.*, 1999], la técnica de modelado de objetos (OMT, por sus siglas en inglés [Rumbaugh *et al.*, 1991]), la metodología Booch [Booch, 1994] y Catalysis [D'Souza y Wills, 1999] son metodologías orientadas a objetos para el desarrollo de software.

Las metodologías de desarrollo de software descomponen el proceso en actividades. La OMT proporciona métodos para tres actividades: *análisis*, que se enfoca en la formalización de los requerimientos del sistema en un modelo de objeto, *diseño del sistema*, que se enfoca en decisiones estratégicas, y *diseño del objeto*, que transforma el modelo de análisis en un modelo de objeto que puede ser puesto en práctica. La metodología OMT supone que ya se han definido los requerimientos y no proporciona métodos para la obtención de los mismos. El proceso de desa-

rrollo de software unificado también incluye una actividad de *análisis* y trata al *diseño del sistema* y al *diseño del objeto* como una sola actividad llamada *diseño*. El proceso unificado, a diferencia del OMT, incluye una actividad de *captura de requerimientos* para la obtención y modelado de requerimientos. Catalysis, aunque usa la misma notación del proceso unificado, se enfoca más en la reutilización del diseño y el código usando patrones y marcos. Todas estas metodologías se enfocan en el manejo de sistemas complejos.

En este libro presentamos una metodología para el desarrollo de sistemas complejos y también para los cambiantes. Durante el curso de nuestra enseñanza e investigación ([Bruegge, 1992], [Bruegge y Coyne, 1993], [Bruegge y Coyne, 1994], [Coyne *et al.*, 1995]) hemos adaptado y refinado métodos de varias fuentes. Para actividades que modelan el dominio de aplicación, como la obtención de requerimientos y el análisis, describimos métodos similares a los de OOSE [Jacobson *et al.*, 1992]. Para actividades de modelado del dominio de solución, como el diseño del sistema y el diseño de objetos, describimos actividades orientadas a objetos similares a las de OMT. Para actividades relacionadas con el cambio nos enfocamos en la administración de las razones que se originaron con la investigación de las razones de diseño [Moran y Carroll, 1996] y la administración de la configuración que se originó para el mantenimiento de sistemas grandes [Babich, 1986].

1.4 Actividades de desarrollo de ingeniería de software

En esta sección damos una panorámica de las actividades técnicas asociadas con la ingeniería de software. Las actividades de desarrollo manejan la complejidad mediante la construcción de modelos de los dominios del problema o del sistema. Las actividades de desarrollo incluyen:

- Obtención de requerimientos (sección 1.4.1)
- Análisis (sección 1.4.2)
- Diseño del sistema (sección 1.4.3)
- Diseño de objetos (sección 1.4.4)
- Implementación (sección 1.4.5)

En la sección 1.5 damos una panorámica de las actividades administrativas asociadas con la ingeniería de software.

1.4.1 Obtención de requerimientos

Durante la obtención de requerimientos, el cliente y los desarrolladores definen el propósito del sistema. El resultado de esta actividad es una descripción del sistema en términos de actores y casos de uso. Los actores representan las entidades externas que interactúan con el sistema. Los actores incluyen papeles como los usuarios finales, otras computadoras con las que necesite tratar el sistema (por ejemplo, un banco de computadoras central, una red) y el ambiente (por ejemplo, un proceso químico). Los casos de uso son secuencias de eventos generales que describen todas las acciones posibles entre un actor y el sistema para un fragmento de funcionalidad dado. La figura 1-2 muestra un caso de uso para el ejemplo del distribuidor de boletos que tratamos antes.

Nombre del caso de uso	`CompraBoletoSencillo`.
Actor participante	Iniciado por `Viajero`.
Condición inicial	1. El `Viajero` se para enfrente del distribuidor de boletos, que puede estar ubicado en la estación de origen o en otra estación.
Flujo de eventos	2. El `Viajero` selecciona las estaciones de origen y destino. 3. El `DistribuidorDeBoletos` despliega el precio del boleto. 4. El `Viajero` inserta una cantidad de dinero que, por lo menos, es igual al precio del boleto. 5. El `DistribuidorDeBoletos` emite el boleto especificado al `Viajero` y regresa el cambio si es el caso.
Condición de salida	6. El `Viajero` toma un boleto válido y el cambio, en su caso.
Requerimientos especiales	Si la transacción no se termina después de un minuto de inactividad, el `DistribuidorDeBoletos` regresa todo el dinero insertado.

Figura 1-2 Un ejemplo de un caso de uso: `CompraBoletoSencillo`.

Durante la obtención de requerimientos, el cliente y los desarrolladores también se ponen de acuerdo sobre un conjunto de requerimientos no funcionales. Los siguientes son ejemplos de requerimientos no funcionales:

- El distribuidor de boletos deberá estar disponible para los viajeros al menos 95% del tiempo.
- El distribuidor de boletos debe proporcionar retroalimentación al viajero (por ejemplo, desplegar el precio del boleto, regresar el cambio) menos de un segundo después de que se haya seleccionado la transacción.

En el capítulo 4, *Obtención de requerimientos*, describimos con mayor detalle la descripción de los requerimientos, incluyendo casos de uso y requerimientos no funcionales.

1.4.2 Análisis

Durante el análisis, los desarrolladores tratan de producir un modelo del sistema que sea correcto, completo, consistente, claro, realista y verificable. Los desarrolladores transforman los casos de uso producidos durante la obtención de requerimientos en un modelo de objeto que describa por completo al sistema. Durante esta actividad, los desarrolladores descubren ambigüedades e inconsistencias en el modelo de caso de uso y las resuelven con el cliente. El resultado del análisis es un modelo de objeto comentado con atributos, operaciones y asociaciones. La figura 1-3 muestra un ejemplo de un modelo de objeto para el `DistribuidorDeBoletos`.

En el capítulo 5, *Análisis*, describimos con detalle el análisis, incluyendo modelos de objetos. En el capítulo 2, *Modelado con UML*, describimos con detalle la notación UML para la representación de modelos.

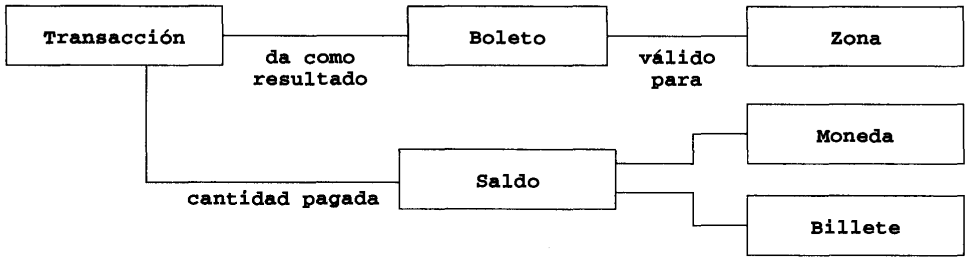


Figura 1-3 Un modelo de objeto para el distribuidor de boletos (diagrama de clase UML). En el caso de uso `CompraBoletoSencillo`, un `Viajero` inicia una transacción que dará como resultado un `Boleto`. Un `Boleto` es válido solo para una `Zona` especificada. Durante la `Transacción`, el sistema calcula el `Saldo` contando las `Monedas` y `Billetes` insertados.

1.4.3 Diseño del sistema

Durante el diseño del sistema, los desarrolladores definen los objetivos de diseño del proyecto y descomponen el sistema en subsistemas más pequeños que pueden realizar los equipos individuales. Los desarrolladores también seleccionan estrategias para la construcción del sistema, como la plataforma de hardware y software en la que ejecutará el sistema, la estrategia de almacenamiento de datos persistentes, el flujo de control global, la política de control de acceso y el manejo de las condiciones de frontera. El resultado de un diseño de sistema es una descripción clara de cada una de estas estrategias, una descomposición en subsistemas y un diagrama de organización que representa el mapeo en hardware y software del sistema. La figura 1-4 muestra un ejemplo de una descomposición del sistema para el distribuidor de boletos.

En el capítulo 6, *Diseño del sistema*, describimos con detalle el diseño del sistema y sus conceptos relacionados.

1.4.4 Diseño de objetos

Durante el diseño de objetos, los desarrolladores definen objetos personalizados para cubrir el hueco entre el modelo de análisis y la plataforma de hardware y software definida durante el diseño del sistema. Esto incluye definir con precisión los objetos e interfaces de subsistemas, la selección de componentes hechos, la reestructuración del modelo de objeto para lograr los objetivos de diseño, tales como extensibilidad o comprensión, y la optimización del modelo de objetos para el desempeño. El resultado de la actividad de diseño de objetos es un modelo de objetos detallado, comentado con restricciones y descripciones precisas para cada elemento. En el capítulo 7, *Diseño de objetos*, describimos con detalle el diseño de objetos y sus conceptos relacionados.

1.4.5 Implementación

Durante la implementación, los desarrolladores traducen el modelo de objetos en código fuente. Esto incluye la implementación de los atributos y métodos de cada objeto y la integración de todos los objetos de forma tal que funcionen como un solo sistema. La actividad de implementación cubre el hueco entre el modelo de diseño de objetos detallado y el conjunto completo de archivos de código fuente que pueden ser compilados juntos. En este libro no trata-

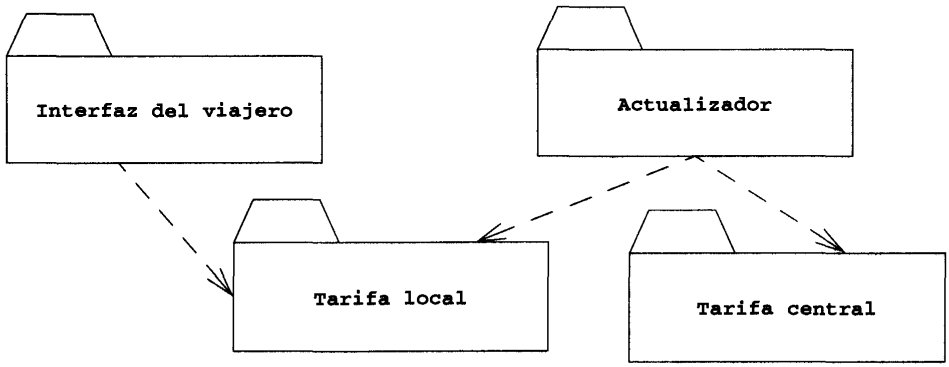


Figura 1-4 Una descomposición en subsistemas del `DistribuidorDeBoletos` (diagrama de clase UML, las carpetas representan subsistemas y las líneas de guiones representan dependencias). El subsistema `Interfaz del viajero` es responsable de la recolección de entrada del `Viajero` y proporciona retroalimentación (por ejemplo, despliega el precio del boleto, regresa el cambio). El subsistema `Tarifa local` calcula el precio de boletos diferentes de acuerdo con una base de datos local. El subsistema `Tarifa central`, que se encuentra en una computadora central, conserva una copia de referencia de la base de datos de tarifas. Un subsistema `Actualizador` es responsable de la actualización de las bases de datos locales en cada `DistribuidorDeBoletos` mediante una red cuando cambian los precios de los boletos.

mos ninguna actividad de implementación, ya que suponemos que el lector ya está familiarizado con los conceptos de la programación.

1.5 Administración del desarrollo de software

En esta sección describimos en forma breve las actividades relacionadas con la administración de un proyecto de ingeniería de software. Las actividades de administración se enfocan en la planeación del proyecto, la supervisión de su estado, el seguimiento de los cambios y la coordinación de los recursos para que se entregue un producto de alta calidad a tiempo y dentro del presupuesto. Las actividades de administración no involucran sólo a los gerentes, sino también a la mayoría de los demás participantes del proyecto. Las actividades de administración incluyen:

- Comunicación (sección 1.5.1)
- Administración de la fundamentación (sección 1.5.2)
- Pruebas (sección 1.5.3)
- Administración de la configuración del software (sección 1.5.4)
- Administración del proyecto (sección 1.5.5)
- Actividades de modelado del ciclo de vida del software (sección 1.5.6)

1.5.1 Comunicación

La comunicación es la actividad más crítica y la que consume más tiempo en la ingeniería de software. La falta de comprensión y las omisiones con frecuencia conducen a fallas y retrasos cuya corrección posterior durante el desarrollo es costosa. La comunicación incluye el intercambio de modelos y documentos acerca del sistema y su dominio de aplicación, reportando el

estado de los productos de trabajo, proporcionando retroalimentación sobre la calidad de los productos de trabajo, proponiendo y negociando asuntos y comunicando decisiones. La comunicación se dificulta por la diversidad de conocimientos de los participantes, por su distribución geográfica y por el volumen, complejidad y evolución de la información que se intercambia.

Los participantes en el proyecto disponen de muchas herramientas para manejar los asuntos de comunicación. Las más efectivas son las convenciones: cuando los participantes se ponen de acuerdo en la notación para representar la información, en las herramientas para el manejo de información y en los procedimientos para presentar y resolver asuntos, ya han eliminado una buena cantidad de fuentes de incomprensión. Ejemplos de notación incluyen diagramas UML, plantillas para la escritura de documentos y minutas de reuniones y esquemas de identificación para la denominación de componentes de software. Ejemplos de herramientas incluyen herramientas de ingeniería de software asistida por computadora (CASE, por sus siglas en inglés) para el mantenimiento de modelos, procesadores de palabras para la generación de documentos y formatos de intercambio para publicar información. Ejemplos de procedimientos incluyen procedimientos de reuniones para la organización, conducción y captura de una reunión, procedimientos de revisión para revisar documentos y proporcionar retroalimentación y procedimientos de inspección para la detección de defectos en los modelos o en el código fuente. Las convenciones que se seleccionen no necesitan ser las mejores de que se disponga, ya que sólo es necesario que todos las compartan y estén de acuerdo en ellas. En el capítulo 3, *Comunicación de proyectos*, describimos con detalle los asuntos de comunicación.

1.5.2 Administración de la fundamentación

La fundamentación es la justificación de las decisiones. Para una decisión dada, su fundamentación incluye el problema que resuelve, las alternativas que consideraron los desarrolladores, los criterios que usaron los desarrolladores para evaluar las alternativas, el debate que sostuvieron los desarrolladores para lograr el consenso y la decisión. La fundamentación es la información más importante que necesitan los desarrolladores cuando hacen cambios al sistema. Si cambia un criterio, los desarrolladores pueden volver a evaluar todas las decisiones que dependen de dicho criterio. Si llegan a estar disponibles nuevas alternativas, se les puede comparar con todas las demás que ya han sido evaluadas. Si se cuestiona una decisión, pueden recuperar su fundamentación para justificarla.

Por desgracia, la fundamentación también es la información más compleja que manejan los desarrolladores durante el desarrollo y, por lo tanto, la más difícil de actualizar y mantener. Para manejar este reto, los desarrolladores capturan la fundamentación durante las reuniones en línea, representan la fundamentación con modelos de problema y tienen acceso a la fundamentación durante los cambios. En el capítulo 8, *Administración de la fundamentación*, describimos con detalle estos asuntos.

1.5.3 Pruebas

Durante las pruebas, los desarrolladores encuentran diferencias entre el sistema y sus modelos ejecutando el sistema (o partes de él) con conjuntos de datos de prueba. Aunque las pruebas no se consideran, por lo general, como una actividad administrativa, las describimos aquí debido a que ayudan a la determinación de la calidad del sistema y sus modelos relacionados. Durante las pruebas unitarias, los desarrolladores comparan el modelo del diseño de objetos con cada objeto y sub-

sistema. Durante las pruebas de integración se forman combinaciones de subsistemas y se comparan con el modelo de diseño del sistema. Durante las pruebas del sistema se ejecutan casos típicos y excepcionales en el sistema y se comparan con el modelo de requerimientos. El objetivo de las pruebas es descubrir la mayor cantidad posible de fallas para que puedan repararse antes de la entrega del sistema. En el capítulo 9, *Pruebas*, describimos estos asuntos con mayor detalle.

1.5.4 Administración de la configuración del software

La administración de la configuración del software es el proceso que supervisa y controla los cambios en los productos de trabajo. El cambio se extiende por el desarrollo del software. Los requerimientos cambian conforme el cliente solicita nuevas características y conforme los desarrolladores mejoran su comprensión del dominio de aplicación. La plataforma de hardware y software sobre la que se está construyendo el sistema cambia conforme se dispone de nuevas tecnologías. El sistema cambia conforme se descubren fallas durante las pruebas y se reparan. La administración de la configuración del software acostumbra estar en el ámbito del mantenimiento, cuando se introducen mejoras graduales en el sistema. Sin embargo, en los procesos de desarrollo modernos los cambios suceden mucho más pronto que el mantenimiento. Conforme se hace difusa la distinción entre desarrollo y mantenimiento se pueden manejar los cambios usando la administración de la configuración en todas las etapas.

La administración de la configuración permite que los desarrolladores lleven cuenta de los cambios. El sistema está representado como varios conceptos de configuración que pueden revisarse en forma independiente. Para cada concepto de configuración se lleva cuenta de su evolución como una serie de versiones. El examen y la selección de versiones permite que los desarrolladores den marcha atrás hacia un estado bien definido del sistema cuando falla un cambio.

La administración de la configuración también permite que los desarrolladores controlen el cambio. Después de que se ha definido una línea básica, cualquier cambio necesita ser valorado y aprobado antes de que se le ponga en práctica. Esto permite que la administración se asegure que el sistema esté evolucionando de acuerdo con los objetivos y que se limite la cantidad de problemas introducidos en el sistema. En el capítulo 10, *Administración de la configuración del software*, describimos estos asuntos con más detalle.

1.5.5 Administración del proyecto

La administración del proyecto no produce ningún artefacto por sí misma. En vez de ello, incluye las actividades de vigilancia que aseguran la entrega de un sistema de alta calidad a tiempo y dentro del presupuesto. Esto incluye la planeación y presupuestación del proyecto durante las negociaciones con el cliente, la contratación de desarrolladores y su organización en equipos, la vigilancia del estado del proyecto y las intervenciones cuando suceden desviaciones. La mayoría de las actividades de administración del proyecto caen más allá del alcance de este libro. Sin embargo, describimos las actividades de administración del proyecto que son visibles ante los desarrolladores y las técnicas que hacen más efectiva la comunicación entre el desarrollo y la administración. En el capítulo 11, *Administración del proyecto*, describimos estos asuntos con detalle.

1.5.6 Ciclo de vida del software

En la sección 1.2 describimos la ingeniería de software como una actividad de modelado. Los desarrolladores construyen modelos de los dominios de la aplicación y la solución para

manejar su complejidad. Al ignorar detalles irrelevantes y enfocarse sólo en lo que es relevante para un problema específico, los desarrolladores pueden resolver en forma más efectiva los problemas y responder preguntas. El proceso de desarrollo de software también puede verse como un sistema complejo con entradas, salidas, actividades y recursos. Por lo tanto, no es sorprendente que las mismas técnicas de modelado que se aplican a los artefactos de software puedan usarse para los procesos de modelado de software. A un modelo general del proceso de desarrollo de software se le llama ciclo de vida del software. En el capítulo de cierre de este libro, capítulo 12, *Ciclo de vida del software*, describimos los ciclos de vida del software.

1.6 Ejercicios

1. ¿Cuál es el propósito del modelado?
2. Un lenguaje de programación es una notación para la representación de algoritmos y estructuras de datos. Liste dos ventajas y dos desventajas del uso de un lenguaje de programación como notación única a lo largo del proceso de desarrollo.
3. Considere una tarea con la que no esté familiarizado, como el diseño de un automóvil con cero emisiones de contaminantes. ¿Cómo podría atacar el problema?
4. ¿Qué significa "la adquisición de conocimiento no es lineal"? Proporcione un ejemplo concreto de adquisición de conocimiento que ilustre esto.
5. Plantee una fundamentación hipotética para las siguientes decisiones de diseño:
   - "El distribuidor de boletos será, a lo mucho, de un metro y medio de alto".
   - "El distribuidor de boletos incluirá dos sistemas de cómputo redundantes".
   - "El distribuidor de boletos incluirá una pantalla sensible al tacto para el desplegado de instrucciones y entrada de comandos. El único control adicional será un botón de cancelación para abortar la transacción".
6. Especifique cuáles de los siguientes enunciados son requerimientos funcionales y cuáles son requerimiento no funcionales:
   - "El distribuidor de boletos debe permitir que un viajero compre pases semanales".
   - "El distribuidor de boletos debe estar escrito en Java".
   - "El distribuidor de boletos debe ser fácil de usar".
7. Especifique cuáles de las siguientes decisiones se tomaron durante el levantamiento de requerimientos o durante el diseño del sistema:
   - "El distribuidor de boletos está compuesto por un subsistema de interfaz de usuario, un subsistema para calcular la tarifa y un subsistema de red para manejar la comunicación con la computadora central".
   - "El distribuidor de boletos usará chips de procesador PowerPC".
   - "El distribuidor de boletos proporciona ayuda en línea al viajero".
8. ¿Cuál es la diferencia entre una tarea y una actividad?
9. Un avión de pasajeros está compuesto por varios millones de partes individuales y requiere miles de personas para ensamblarlo. Un puente de autopista de cuatro carriles es otro ejemplo de complejidad. La primera versión de Word para Windows, un procesador de palabras lanzado por Microsoft en noviembre de 1989, requirió 55 años hombre, dando como resultado 249,000 líneas de código fuente y fue entregado con 4 años de retraso. Los

aviones y los puentes de autopista por lo general se entregan a tiempo y por debajo de su presupuesto, mientras que con el software a menudo no es así. Discuta cuáles son, en su opinión, las diferencias entre el desarrollo de un avión, un puente y un procesador de palabras que pueden causar esta situación.

Referencias

[Babich, 1986] W. A. Babich, *Software Configuration Management.* Addison-Wesley, Reading, MA, 1986.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2a. ed. Benjamin/ Cummings, Redwood City, CA, 1994.

[Bruegge, 1992] B. Bruegge, "Teaching an industry-oriented software engineering course", *Software Engineering Education*, SEI Conference, Lecture Notes in Computer Sciences, Springer Verlag, vol. 640, octubre de 1992, págs. 65-87.

[Bruegge y Coyne, 1993] B. Bruegge y R. Coyne, "Model-based software engineering in larger scale project courses", *IFIP Transactions on Computer Science and Technology*, Elsevier Science, Países Bajos, vol. A-40, 1993, págs. 273-287.

[Bruegge y Coyne, 1994] B. Bruegge y R. Coyne, "Teaching iterative object-oriented development: Lessons and directions", en Jorge L. Diaz-Herrera (ed.), *7th Conference on Software Engineering Education*, Lecture Notes in Computer Science, Springer Verlag, vol. 750, enero de 1994, págs. 413-427.

[Coyne *et al.*, 1995] R. Coyne, B. Bruegge, A. Dutoit y D. Rothenberger, "Teaching more comprehensive model-based software engineering: Experience with Objectory's use case approach", en Linda Ibraham (ed.), *8th Conference on Software Engineering Education*, Lecture Notes in Computer Science, Springer Verlag, abril de 1995, págs. 339-374.

[D'Souza y Wills, 1999] D. F. D'Souza y A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA, 1999.

[IEEE, 1997] *IEEE Standards Collection Software Engineering*. IEEE, Piscataway, NJ, 1997.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[Jacobson *et al.*, 1999] I Jacobson, G. Booch y J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[Moran y Carroll, 1996] T. P. Moran y J. M. Carroll (eds.), *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Neumann, 1995] P. G. Neumann, *Computer-Related Risks*. Addison-Wesley, Reading, MA, 1995.

[OMG, 1998] Object Management Group, *OMG Unified Modeling Language Specification.* Framingham, MA, 1998. http://www.omg.org.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach.* Clarendon, Oxford, 1992.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Simon, 1970] H. A. Simon, *The Sciences of the Artificial.* MIT Press, Cambridge, MA, 1970.

[Spivey, 1989] J. M. Spivey, *The Z Notation, A Reference Manual.* Prentice Hall Int., Hertfordshire, Inglaterra, 1989.

2

Modelado con UML

"Todos los mecánicos están familiarizados con el problema de la parte que no se puede comprar porque no se le puede encontrar, debido a que el fabricante la considera como parte de alguna otra cosa".

—Robert Pirsig, en Zen and the Art of Motorcycle Maintenance

Las notaciones nos permiten formular ideas complejas en forma resumida y precisa. En los proyectos que involucran a muchos participantes, a menudo con diferentes conocimientos técnicos y culturales, la precisión y claridad son críticas conforme se incrementa rápidamente el costo de la falta de comunicación.

Para que una notación permita la comunicación precisa debe tener una semántica *bien definida*, debe ser *muy adecuada* para la representación de un aspecto dado de un sistema y debe ser *bien comprendida* por los participantes del proyecto. En esto último se encuentra la fortaleza de los estándares y las convenciones: cuando una notación es utilizada por gran cantidad de participantes, hay poco espacio para las malas interpretaciones y la ambigüedad. Por el contrario, cuando existen muchos dialectos de una notación, o cuando se usa una notación muy especializada, los usuarios de la notación están propensos a malas interpretaciones cada vez que cada usuario impone su propia interpretación. Seleccionamos el UML (lenguaje de modelado unificado [OMG, 1998]) como notación principal para este libro debido a que tiene una semántica bien definida, proporciona un espectro de notaciones para la representación de diferentes aspectos de un sistema y ha sido aceptado como notación estándar en la industria.

En este capítulo, primero describimos los conceptos del modelado en general y del modelado orientado a objetos en particular. Luego describimos cinco notaciones fundamentales del UML que usamos a lo largo del libro: diagramas de caso de uso, diagramas de clase, diagramas de secuencia, diagramas de gráfica de estado y diagramas de actividad. Para cada una de estas notaciones describimos su semántica básica y proporcionamos ejemplos. Luego volvemos a ver estas notaciones con más detalle en capítulos posteriores conforme describimos las actividades que las usan. Las notaciones especializadas que usamos sólo en un capítulo se presentan después, como las gráficas PERT en el capítulo 11, *Administración del proyecto*, y los componentes UML y diagramas de entrega en el capítulo 6, Diseño del sistema.

2.1 Introducción

UML es una notación que se produjo como resultado de la unificación de la técnica de modelado de objetos (OMT, por sus siglas en inglés [Rumbaugh *et al.*, 1991]), Booch [Booch, 1994] e ingeniería de software orientada a objetos (OOSE, por sus siglas en inglés [Jacobson *et al.*, 1992]). El UML también ha sido influido por otras notaciones orientadas a objetos, como las presentadas por Shlaer/Mellor [Mellor y Shlaer, 1998], Coad/Yourdon [Coad *et al.*, 1995], Wirfs-Brock [Wirfs-Brock *et al.*, 1990] y Martin/Odell [Martin y Odell, 1992]. El UML ha sido diseñado para un amplio rango de aplicaciones. Por lo tanto, proporciona construcciones para un amplio rango de sistemas y actividades (por ejemplo, sistemas de tiempo real, sistemas distribuidos, análisis, diseño del sistema, entregas). El desarrollo de sistemas se enfoca en tres modelos diferentes del sistema:

- El **modelo funcional**, representado en UML con diagramas de caso de uso, describe la funcionalidad del sistema desde el punto de vista del usuario.
- El **modelo de objetos**, representado en UML con diagramas de clase, describe la estructura de un sistema desde el punto de vista de objetos, atributos, asociaciones y operaciones.
- El **modelo dinámico**, representado en UML con diagramas de secuencia, diagramas de gráfica de estado y diagramas de actividad, describe el comportamiento interno del sistema. Los diagramas de secuencia describen el comportamiento como una secuencia de mensajes intercambiados entre un *conjunto de objetos*, mientras que los diagramas de gráfica de estado describen el comportamiento desde el punto de vista de estados de un *objeto individual* y las transiciones posibles entre estados.

En este capítulo describimos diagramas UML para la representación de estos modelos. La presentación de estas notaciones plantea un reto interesante. Por un lado, la comprensión del propósito de una notación requiere alguna familiaridad con las actividades que la usan. Por otro lado, es necesario comprender la notación antes de describir las actividades. Para resolver este problema presentamos el UML en forma iterativa. En la siguiente sección primero proporcionamos una panorámica de las cinco notaciones básicas de UML. En la sección 2.3 presentamos las ideas fundamentales del modelado. En la sección 2.4 volvemos a ver las cinco notaciones básicas del UML a la luz de los conceptos de modelado. En capítulos subsiguientes examinamos estas notaciones con mayor detalle cuando presentamos las actividades que las utilizan.

2.2 Una panorámica del UML

En esta sección presentamos brevemente cinco notaciones UML:

- Diagramas de caso de uso (sección 2.2.1)
- Diagramas de clase (sección 2.2.2)
- Diagramas de secuencia (sección 2.2.3)
- Diagramas de gráfica de estado (sección 2.2.4)
- Diagramas de actividad (sección 2.2.5)

2.2.1 Diagramas de caso de uso

Los casos de uso se utilizan durante la obtención de requerimientos y el análisis para representar la funcionalidad del sistema. Los casos de uso se enfocan en el comportamiento del sistema desde un punto de vista externo. Un diagrama de caso de uso describe una función proporcionada por el sistema que produce un resultado visible para un actor. Un actor describe cualquier entidad que interactúa con el sistema (por ejemplo, un usuario, otro sistema, el ambiente físico del sistema). La identificación de los actores y los casos de uso da como resultado la definición de la frontera del sistema, esto es, diferencia entre las tareas realizadas por el sistema y las realizadas por su ambiente. Los actores están fuera de la frontera del sistema, mientras que los casos de uso están dentro de la frontera del sistema.

Por ejemplo, la figura 2-1 muestra un diagrama de caso de uso para un reloj simple. El actor `UsuarioReloj` puede consultar la hora en su reloj (con el caso de uso `LeerHora`) o ajustar la hora (con el caso de uso `AjustarHora`). Sin embargo, sólo el actor `PersonaReparadoraRelojes` puede cambiar la batería del reloj (con el caso de uso `CambiarBatería`).

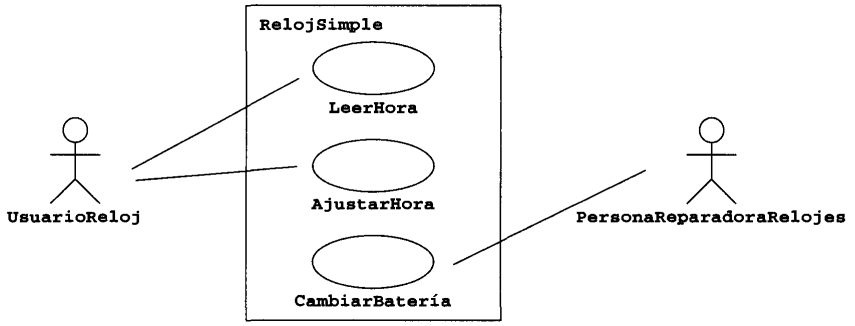


Figura 2-1 Un diagrama de caso de uso UML que describe la funcionalidad de un reloj simple. El actor `UsuarioReloj` puede consultar la hora en su reloj (con el caso de uso `LeerHora`) o ajustar la hora (con el caso de uso `AjustarHora`). Sin embargo, sólo el actor `PersonaReparadoraRelojes` puede cambiar la batería del reloj (con el caso de uso `CambiarBatería`). Los actores se representan con muñequitos ("hombres de paja"), los casos de uso con óvalos y la frontera del sistema con un cuadro que encierra los casos de uso.

2.2.2 Diagramas de clase

Usamos diagramas de clase para describir la estructura del sistema. Las clases son abstracciones que especifican la estructura y el comportamiento común de un conjunto de objetos. Los objetos son instancias de las clases que se crean, modifican y destruyen durante la ejecución del sistema. Los objetos tienen estados que incluyen los valores de sus atributos y sus relaciones con otros objetos.

Los diagramas de clase describen el sistema desde el punto de vista de objetos, clases, atributos, operaciones y sus asociaciones. Por ejemplo, la figura 2-2 es un diagrama de clase que describe los elementos de todos los relojes de la clase `RelojSimple`. Todos estos objetos de reloj tienen una asociación con un objeto de la clase `BotónOprimible`, un objeto de la clase `Pantalla`, un objeto de la clase `Hora` y un objeto de la clase `Batería`. Los números que están al final de las asociaciones

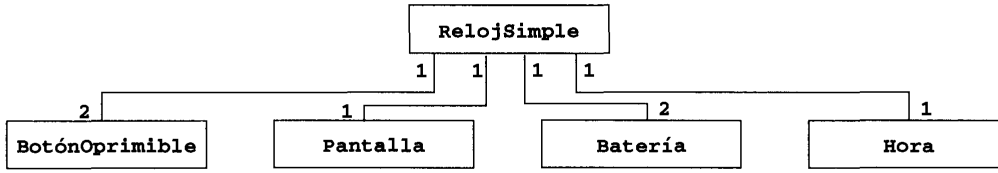


Figura 2-2 Un diagrama de clase UML que describe los elementos de un reloj simple.

indican la cantidad de vínculos que puede tener cada objeto `RelojSimple` con un objeto de una clase dada. Por ejemplo, un `RelojSimple` tiene exactamente dos `BotónOprimible`, una `Pantalla`, dos `Batería` y una `Hora`. En forma similar, todos los objetos `BotónOprimible`, `Pantalla`, `Hora` y `Batería` están asociados exactamente a un objeto `RelojSimple`.

2.2.3 Diagramas de secuencia

Los diagramas de secuencia se usan para formalizar el comportamiento del sistema y para visualizar la comunicación entre objetos. Son útiles para la identificación de objetos adicionales que participan en los casos de uso. A los objetos involucrados en un caso de uso les llamamos **objetos participantes**. Un diagrama de secuencia representa las interacciones que suceden entre esos objetos. Por ejemplo, la figura 2-3 es un diagrama de secuencia para el caso de uso `AjustarHora` de nuestro reloj simple. La columna de la extrema izquierda representa al actor `UsuarioReloj` que inicia el caso de uso. Las flechas etiquetadas representan los estímulos que un actor u objeto envía a otros objetos. En este caso, el `UsuarioReloj` oprime el botón 1 dos veces y el botón 2 una vez para adelantar el reloj un minuto. El caso de uso `AjustarHora` termina cuando el `UsuarioReloj` oprime ambos botones simultáneamente.

2.2.4 Diagramas de gráfica de estado

Los diagramas de gráfica de estado describen el comportamiento de un objeto individual como varios estados y transiciones entre esos estados. Un estado representa un conjunto particular de valores para un objeto. En un estado dado, una transición representa un estado futuro hacia el cual se puede mover el objeto y las condiciones asociadas con el cambio de estado. Por ejemplo, la figura 2-4 es un diagrama de gráfica de estado para el `RelojSimple`. Observe que este diagrama representa información diferente a la del diagrama de secuencia de la figura 2-3. El diagrama de secuencia se enfoca en los mensajes intercambiados entre los objetos como resultado de eventos externos creados por actores. El diagrama de gráfica de estado se enfoca en las transiciones entre estados como resultado de eventos externos de un objeto individual.

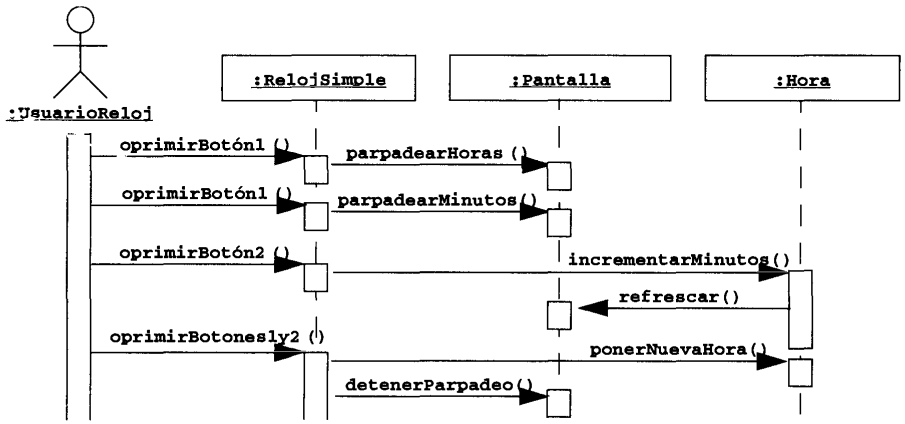


Figura 2-3 Un diagrama de secuencia UML para el `RelojSimple`. La columna de la extrema izquierda representa la línea de tiempo del actor `UsuarioReloj`, quien inicia el caso de uso. Las demás columnas representan la línea de tiempo de los objetos que participan en este caso de uso. Los nombres de objetos están subrayados para indicar que son instancias (en vez de clases). Las flechas etiquetadas son estímulos que envía un actor o un objeto a otros objetos.

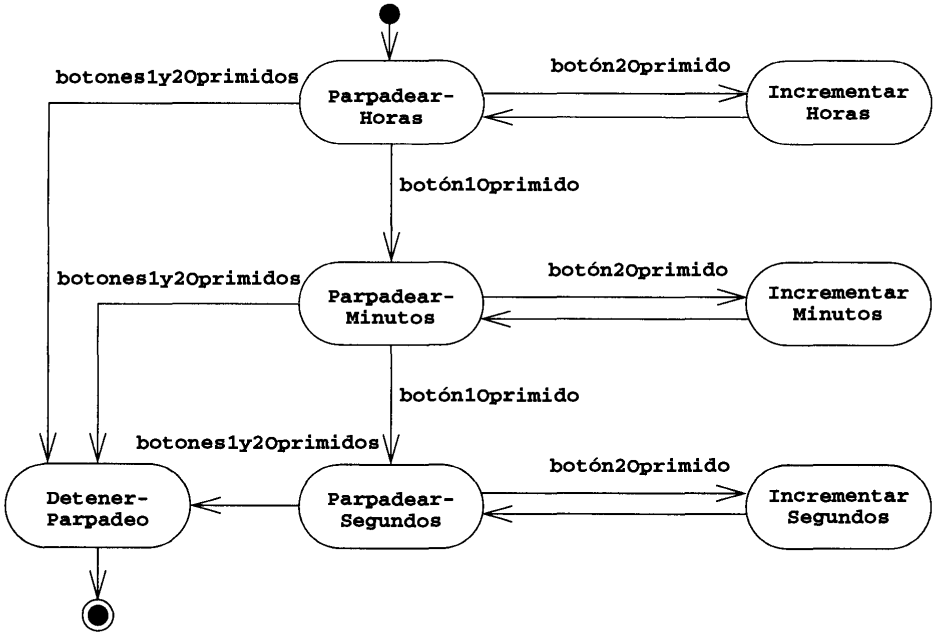


Figura 2-4 Un diagrama de gráfica de estado UML para el caso de uso `AjustarHora` de `RelojSimple`.

2.2.5 Diagramas de actividad

Un diagrama de actividad describe un sistema desde el punto de vista de las actividades. Las actividades son estados que representan la ejecución de un conjunto de operaciones. La terminación de estas operaciones dispara una transición hacia otra actividad. Los diagramas de actividad se parecen a los diagramas de flujo en que pueden usarse para representar el flujo de control (es decir, el orden en que suceden las operaciones) y el flujo de datos (es decir, los objetos que se intercambian entre operaciones). Por ejemplo, la figura 2-5 es un diagrama de actividad que representa las actividades relacionadas con el manejo de un `Incidente` en FRIEND. Los rectángulos redondeados representan actividades, las flechas representan transiciones entre actividades y las barras gruesas representan la sincronización del flujo de control. El diagrama de actividad de la figura 2-5 muestra que `AsignarRecursos`, `CoordinarRecursos` y `DocumentarIncidente` sólo pueden iniciarse después de que haya terminado la actividad `AbrirIncidente`. Del mismo modo, la actividad `ArchivarIncidente` sólo puede iniciarse después de la terminación de `AsignarRecursos`, `CoordinarRecursos` y `DocumentarIncidente`. Sin embargo, estas tres últimas actividades pueden suceder en forma concurrente.

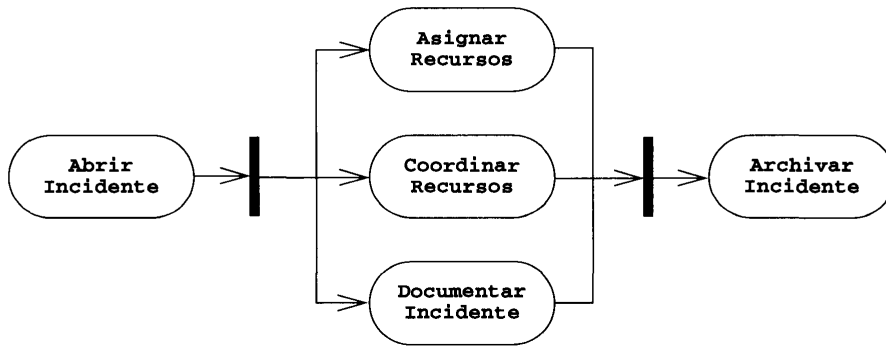


Figura 2-5 Un ejemplo de un diagrama de actividad UML. Los diagramas de actividad representan el comportamiento en términos de actividades y sus restricciones de precedencia. La terminación de una actividad dispara una transición saliente que, a su vez, puede iniciar otra actividad.

Esto concluye nuestro primer paseo por las cinco notaciones básicas del UML. Ahora entraremos a un mayor detalle: en la sección 2.3 presentamos los conceptos de modelado básicos, incluyendo la definición de sistemas, modelos, tipos e instancias, abstracción y falsificación. En las secciones 2.4.1 a 2.4.5 describimos de manera detallada los diagramas de caso de uso, diagramas de clase, diagramas de secuencia, diagramas de gráfica de estado y diagramas de actividad. Ilustramos su uso con un ejemplo simple. La sección 2.4.6 describe construcciones diversas, como los paquetes y las notas, que se usan en todos los tipos de diagramas. Usamos estas cinco notaciones a lo largo del libro para describir sistemas de software, productos de trabajo, actividades y organizaciones. Mediante el uso consistente y sistemático de un conjunto pequeño de notaciones, esperamos proporcionar al lector un conocimiento operacional del UML.

2.3 Conceptos de modelado

En esta sección describimos los conceptos básicos del modelado. Primero definimos los términos **sistema** y **modelo**, y tratamos el propósito del **modelado**. Luego definimos los términos **concepto** y **fenómeno**. Explicamos su relación con los lenguajes de programación y con términos como **tipos**, **clases**, **instancias** y **objetos**. Por último, describimos la manera en que se enfoca el modelado orientado a objetos en la construcción de una abstracción del ambiente del sistema como base para el modelo del sistema.

2.3.1 Sistemas, modelos y vistas

Un **sistema** es un conjunto organizado de partes que se comunican, diseñado para un propósito específico. Un automóvil, compuesto por cuatro ruedas, un chasis, una carrocería y un motor, está diseñado para transportar personas. Un reloj, compuesto por una batería, un circuito, engranajes y manecillas, está diseñado para medir el tiempo. Un sistema de nómina, compuesto por una gran computadora, impresoras, discos, software y el personal de nóminas, está diseñado para expedir cheques de salario para los empleados de una compañía. Partes de un sistema pueden, a su vez, considerarse como sistemas más simples llamados **subsistemas**. El motor de un automóvil, compuesto por cilindros, pistones, un módulo de inyección y muchas otras partes, es un subsistema del automóvil. En forma similar, el circuito integrado de un reloj y la gran computadora del sistema de nómina son subsistemas. Esta descomposición en subsistemas puede aplicarse en forma repetida a los subsistemas. Los objetos representan el final de esta repetición, cuando cada parte es lo suficientemente simple para que podamos comprenderla por completo sin mayor descomposición.

Muchos sistemas están compuestos por varios subsistemas interconectados en formas complicadas, a menudo tan complejas que ningún desarrollador solo puede manejarlas en forma completa. El **modelado** es un medio para manejar esta complejidad. Los sistemas complejos se describen, por lo general, mediante más de un modelo, enfocándose cada uno en un aspecto diferente o nivel de precisión. El modelado significa la construcción de una abstracción del sistema que se enfoca en aspectos interesantes e ignora los detalles irrelevantes. Lo que es interesante o irrelevante varía con la tarea que se está realizando. Por ejemplo, supongamos que queremos construir un aeroplano. Aun con la ayuda de expertos en el campo, no podemos construir un aeroplano a partir de cero y esperar que funcione en forma correcta en su primer vuelo. En vez de ello, primero construimos un modelo a escala del fuselaje para probar sus propiedades aerodinámicas. En este modelo a escala sólo necesitamos representar la superficie exterior del aeroplano. Podemos ignorar detalles como el tablero de instrumentos o el motor. Para entrenar pilotos para este nuevo aeroplano también construimos un simulador de vuelo. El simulador de vuelo necesita representar con precisión la disposición y el comportamiento de los instrumentos de vuelo. Sin embargo, en este caso se pueden ignorar los detalles acerca del exterior del aeroplano. Tanto el simulador de vuelo como el modelo a escala son mucho menos complejos que el aeroplano que representan. El modelado nos permite manejar la complejidad mediante un enfoque de dividir y conquistar o de dividir y vencer: para cada tipo de problema que queremos resolver (por ejemplo, la prueba de las propiedades aerodinámicas, el entrenamiento de pilotos) construimos un modelo que sólo se centra en las cuestiones relevantes del problema. Por lo general, el modelado se enfoca en la construcción de un modelo que sea lo suficientemente simple para que una persona lo comprenda en forma plena. Una regla práctica es que cada entidad debe contener, cuando mucho, 7 ± 2 partes [Miller, 1956].

Por desgracia, incluso un modelo puede llegar a ser tan complejo que no sea comprensible con facilidad. Al igual que sucede con los sistemas, aplicamos el mismo enfoque de dividir y con-

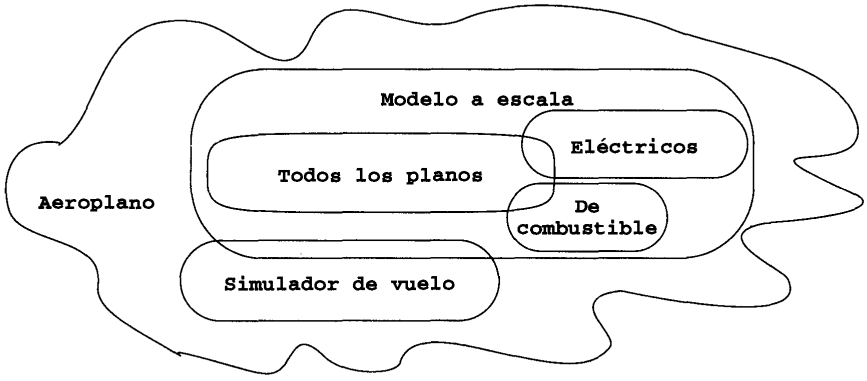


Figura 2-6 Un modelo es una abstracción que describe un subconjunto de un sistema. Una vista muestra aspectos seleccionados de un modelo. Pueden traslaparse las vistas y los modelos de un solo sistema.

quistar para manejar la complejidad de los modelos. Una **vista** se enfoca en un subconjunto de un modelo para hacerlo comprensible (figura 2-6). Por ejemplo, todos los planos necesarios para la construcción de un aeroplano constituyen un modelo. Los extractos necesarios para explicar el funcionamiento del sistema de combustible constituyen la vista del sistema de combustible. Las vistas pueden traslaparse: una vista del aeroplano que representa el alambrado eléctrico también incluye el alambrado para el sistema de combustible.

Las **notaciones** son reglas gráficas o textuales para la representación de vistas. Un diagrama de clase UML es una vista gráfica del modelo del objeto. En los diagramas de alambrado, cada línea de conexión representa un alambre o manojo de alambres diferentes. En los diagramas de clase UML, un rectángulo con un título representa una clase. Una línea entre dos rectángulos representa una relación entre las dos clases correspondientes. Observe que se pueden usar diferentes notaciones para representar la misma vista (figuras 2-7 y 2-8).

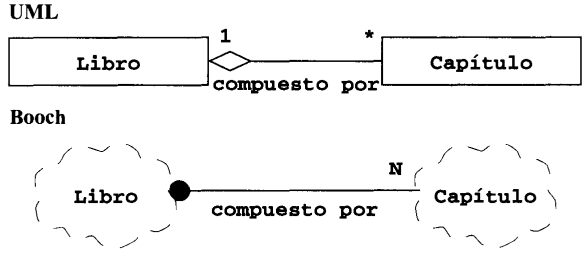


Figura 2-7 Un ejemplo de la descripción de un modelo con dos notaciones diferentes. El modelo incluye dos clases, `Libro` y `Capítulo`, con la relación `El libro está compuesto por capítulos`. En UML las clases se muestran con rectángulos y las asociaciones de agregación con una línea que termina con un rombo. En la notación Booch las clases se muestran con nubes y las asociaciones de agregación con una línea que termina con un círculo relleno.

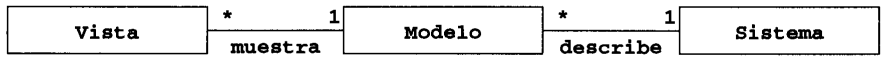


Figura 2-8 Un ejemplo de la descripción del mismo modelo con dos notaciones diferentes. Este diagrama UML representa la información de la figura 2-6: un `sistema` puede describirse mediante muchos `modelos` diferentes que pueden mostrarse mediante muchas `vistas` diferentes.

En el desarrollo de software también hay muchas otras notaciones para el modelado de sistemas. UML describe un sistema desde el punto de vista de clases, eventos, estados, interacciones y actividades. Los diagramas de flujo de datos [De Marco, 1978] muestran la manera en que se recuperan, procesan y almacenan los datos. Cada notación es adecuada para un problema diferente.

En las siguientes secciones nos enfocamos con un mayor detalle en el proceso de modelado. Examinamos las definiciones de **concepto** y **fenómeno**, y su relación con los conceptos de programación de **tipo**, **variable**, **clase** y **objeto**.

2.3.2 Conceptos y fenómenos

Un **fenómeno** es un objeto del mundo tal como es percibido. Los siguientes son fenómenos:

- Este libro
- La tasa actual de interés para los ahorros es 3%
- Mi reloj negro
- El Club de Pescadores del Valle.

Un **concepto** es una abstracción que describe un conjunto de fenómenos. Los siguientes son conceptos:

- Libros de texto sobre ingeniería de software orientado a objetos
- Las tasas de interés para los ahorros
- Los relojes negros
- Los clubes de pescadores

Un concepto describe las propiedades que son comunes para un conjunto de fenómenos. Por ejemplo, el concepto *relojes negros* sólo está interesado en el color de los relojes y no en su origen o su calidad. Un concepto está definido como una tercia: su **nombre** (para distinguirlo de otros conceptos), su **propósito** (las propiedades que determinan si un fenómeno es parte del concepto o no) y sus **miembros** (el conjunto de fenómenos que son parte del concepto).[1] La figura 2-9 ilustra el concepto de reloj. *Reloj* es el nombre del concepto. *Medición de tiempo* es el propósito del reloj. *Mi reloj de pulsera* y *el reloj de pared que está arriba de mi escritorio* son miembros del concepto reloj. Un club tiene un nombre (por ejemplo, "Club de Pescadores

1. A los tres componentes de un concepto también se les llama nombre, intensión y extensión.

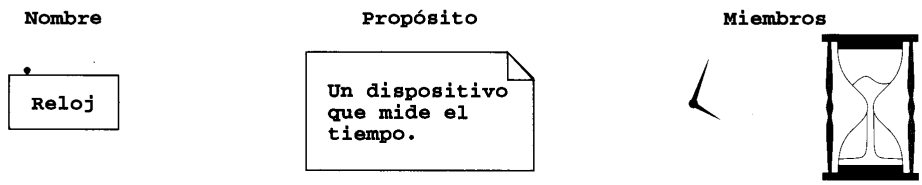


Figura 2-9 Los tres componentes del concepto `Reloj`: nombre, propósito y miembros.

del Valle"), atributos que deben satisfacer los miembros para ser parte del club (por ejemplo, "pescadores que vivan en el valle") y miembros reales (por ejemplo, "Juan Pérez", "Pedro López").

La **abstracción** es la clasificación de los fenómenos en conceptos. El **modelado** es el desarrollo de abstracciones que pueden usarse para resolver preguntas específicas acerca de un conjunto de fenómenos. Una abstracción es más simple de manejar y examinar que su conjunto de fenómenos correspondiente debido a que contiene menos información: los detalles irrelevantes se eliminan de la abstracción. En química, la tabla de elementos resume los diferentes tipos de átomos con base en su peso atómico y número de pares de electrones. No están representados detalles como la disponibilidad de cada sustancia o su participación en diferentes moléculas. En biología se clasifica a las especies en árboles genealógicos con base en características significativas (por ejemplo, especies que tienen sangre caliente, especies que tienen vértebras). Un árbol de especies ignora asuntos relacionados con el comportamiento o el hábitat. En astronomía, las estrellas se clasifican en tipos diferentes con base en su espectro y energía disipada. En esta clasificación se ignoran la ubicación de las estrellas, su composición detallada y sus dimensiones.

En ingeniería puede existir un modelo con anterioridad al fenómeno que representa. Por ejemplo, un modelo UML puede describir un sistema que todavía no ha sido implementado. En la ciencia, el modelo puede establecer la existencia de sistemas y conducir a experimentos que muestren dicha existencia. Por ejemplo, la teoría que hay tras el quark arriba fue desarrollada antes que los experimentos en el acelerador del CERN demostraran la existencia del quark arriba.

Resumiendo, el modelado es la actividad que realizan los ingenieros de software cuando diseñan un sistema. El propósito del modelado es construir una abstracción del sistema que deje a un lado los detalles irrelevantes. Los ingenieros de software abstraen conceptos del dominio de aplicación (es decir, el ambiente en el que está operando el sistema) y del dominio de solución (es decir, las tecnologías para construir un sistema). El modelo resultante es más simple que el ambiente o el sistema y, por lo tanto, es más fácil de manejar. Durante el desarrollo del modelo o su validación, los ingenieros de software necesitan comunicarse acerca del sistema con otros ingenieros, clientes o usuarios. Pueden representar el modelo en su imaginación, en una servilleta, con una herramienta CASE o usando diferentes notaciones. Al hacerlo construyen vistas del modelo para apoyar sus necesidades específicas de comunicación.

2.3.3 Tipos de datos, tipos de datos abstractos e instancias

Un **tipo de dato** es una abstracción en el contexto de un lenguaje de programación. Un tipo de dato tiene un nombre único que lo distingue con respecto a otros tipos de datos, tiene un propósito (es decir, la estructura y las operaciones válidas sobre todos los miembros del tipo de dato) y tiene miembros (es decir, los miembros del tipo de dato). Los tipos de datos se usan en lenguajes con tipo para asegurarse que sólo se apliquen las operaciones válidas a los datos miembro específicos.

Por ejemplo, el nombre `int` en Java corresponde a todos los enteros con signo entre -2^{32} y $2^{32} - 1$. Las operaciones válidas sobre los miembros de este tipo son todas las operaciones de la aritmética entera (por ejemplo, suma, resta, multiplicación, división) y todas las funciones y métodos que tienen parámetros de tipo `int` (por ejemplo, `mod`). El ambiente del tiempo de ejecución de Java lanza una excepción si se aplica una operación de punto flotante a un miembro del tipo de dato `int` (por ejemplo, `trunc` o `floor`).

Un **tipo de dato abstracto** es un tipo de dato especial cuya estructura está oculta con relación al resto del sistema. Esto permite que el desarrollador pueda hacer cambios a la estructura interna y a la implementación del tipo de dato abstracto sin que haya impactos en el resto del sistema.

Por ejemplo, el tipo de dato abstracto Persona puede definir las operaciones `obtenerNombre()`,[2] `obtenerNúmeroSeguroSocial()` y `obtenerDirección()`. El hecho de que el número de seguro social de la persona esté almacenado como número o como cadena no es visible ante el resto del sistema. A tales decisiones se les llama **decisiones de implementación**.

Una **instancia** es cualquier miembro de un tipo de dato específico. Por ejemplo, `1291` es una instancia del tipo `int` y `3.14` es una instancia del tipo `float`. Una instancia de un tipo de dato puede ser manejada con las operaciones definidas por el tipo de dato.

La relación entre el tipo de dato y la instancia es similar a la relación entre concepto y fenómeno: un tipo de dato es una abstracción que describe un conjunto de instancias que comparten características comunes. Por ejemplo, la operación de renombrar una instancia de `Persona` sólo necesita definirse una vez en el tipo de dato `Persona`, pero será aplicable a todas las instancias de `Persona`.

2.3.4 Clases, clases abstractas y objetos

Una **clase** es una abstracción en los lenguajes de programación orientados a objetos. Al igual que los tipos de datos abstractos, una clase encapsula estructura y comportamiento. A diferencia de los tipos de datos abstractos, las clases pueden definirse desde el punto de vista de otras clases usando la generalización. Supongamos que tenemos un reloj que también puede funcionar como calculadora. Entonces la clase `RelojCalculadora` puede ser vista como un refinamiento de la clase `Reloj`. A este tipo de relación entre una clase base y una clase refinada se le llama **generalización**. A la clase base (por ejemplo, `Reloj`) se le llama **superclase** y a la clase refinada se le llama **subclase** (por ejemplo, `RelojCalculadora`). En una relación de generalización, la subclase refina a la superclase definiendo nuevos atributos y operaciones. En la figura 2-10, `RelojCalculadora` define la funcionalidad para la realización de la aritmética simple que no tienen los `Reloj` normales.

2. Nos referimos a una operación mediante su nombre seguido por sus argumentos entre paréntesis. Si no se especifican los argumentos ponemos un par de paréntesis al final del nombre. En la siguiente sección describimos las operaciones con mayor detalle.

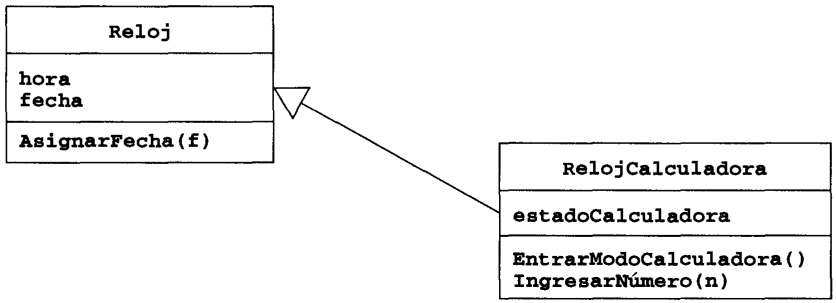


Figura 2-10 Un diagrama de clase UML que muestra dos clases: `Reloj` y `RelojCalculadora`. `RelojCalculadora` es un refinamiento de `Reloj`, proporcionando la funcionalidad de una calculadora que por lo general no se encuentra en los relojes normales. En un diagrama de clase UML las clases y objetos están representados como cuadros con tres compartimientos: el primer compartimiento muestra el nombre de la clase, el segundo sus atributos y el tercero sus operaciones. Los compartimientos segundo y tercero pueden omitirse para abreviar. Una relación de herencia se muestra con una línea que termina con un triángulo. El triángulo apunta hacia la superclase y el otro extremo está en la subclase.

Cuando una generalización sirve sólo para el propósito del modelado de atributos y operaciones compartidos, esto es, si nunca es instancia, a la generalización se le llama **clase abstracta**. Las clases abstractas con frecuencia representan conceptos generalizados en el dominio de aplicación. Cada vez que clasificamos fenómenos en conceptos, con frecuencia se crean generalizaciones para manejar la complejidad de la clasificación. Por ejemplo, en química, al `Benceno` puede considerársele como una clase de moléculas que pertenece a la clase abstracta *`CompuestoOrgánico`* (figura 2-11). Observe que *`CompuestoOrgánico`* es una generalización y no corresponde a ninguna molécula, esto es, no tiene ninguna instancia. Cuando se modelan sistemas de software, con frecuencia las clases abstractas no corresponden a un concepto del dominio de aplicación existente sino que se introducen para reducir la complejidad en el modelo o para promover la reutilización.

Una clase define las **operaciones** que pueden aplicarse a sus instancias. Las operaciones de una superclase pueden ser heredadas y aplicadas también a los objetos de la subclase. Por ejemplo, en la figura 2-10 la operación `AsignarFecha(f)`, que asigna la fecha actual de un `Reloj`, también es aplicable para los `RelojCalculadora`. Sin embargo, la operación `EntrarModoCalculadora()`, definida en la clase `RelojCalculadora`, no es aplicable a la clase `Reloj`.

Una clase define los **atributos** que se aplican a todas sus instancias. Un atributo es una ranura con nombre en la instancia en donde se almacena un valor. Los atributos tienen un nombre único dentro de la clase y un tipo. Los `Reloj` tienen un atributo `hora` y otro `fecha`. Los `RelojCalculadora` tienen un atributo `estadoCalculadora`.

Un **objeto** es una instancia de una clase. Un objeto tiene una identidad y almacena valores de atributo. Cada objeto pertenece exactamente a una clase. En UML, una instancia se muestra con un rectángulo con su nombre subrayado. Esta convención se usa en UML para distinguir entre instancias y tipos.[3] En la figura 2-12, `RelojSimple1291` es una instancia de `Reloj`, y `RelojCalculadora1515` es una instancia de `RelojCalculadora`. Observe que, aunque las ope-

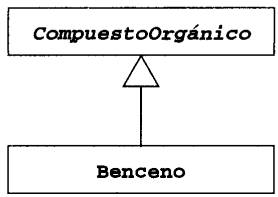


Figura 2-11 Un ejemplo de una clase abstracta (diagrama de clase UML). `CompuestoOrgánico` nunca tiene instancias y sólo sirve como una generalización.

raciones de `Reloj` son aplicables a `RelojCalculadora1515`, `RelojCalculadora1515` no es una instancia de la clase `Reloj`. A diferencia de los tipos de datos abstractos, en algunos lenguajes de programación los atributos de un objeto pueden estar visibles ante otras partes del sistema. Por ejemplo, Java permite que el implementador especifique con gran detalle cuáles atributos son visibles y cuáles no.

2.3.5 Clases de evento, eventos y mensajes

Las **clases de evento** son abstracciones que representan un tipo de evento para el cual el sistema tiene una respuesta común. Un **evento**, una instancia de la clase de evento, es una ocurrencia relevante en el sistema. Por ejemplo, un evento puede ser un estímulo dado por un actor (por ejemplo, "el `UsuarioReloj` oprime el botón izquierdo"), un transcurso de tiempo (por ejemplo, "después de 2 minutos") o el envío de un mensaje entre dos objetos. El envío de un **mensaje** es el mecanismo por el cual el objeto que lo envía solicita la ejecución de una operación en el objeto que lo recibe. El mensaje está compuesto por un nombre y varios argumentos.

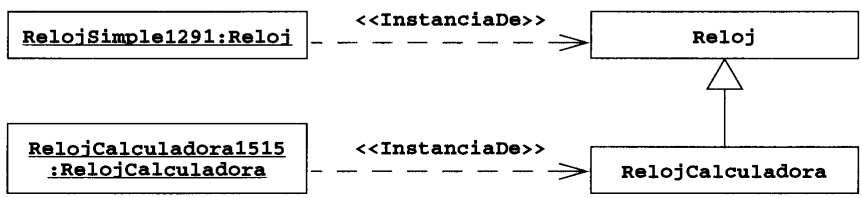


Figura 2-12 Un diagrama de clase UML que muestra instancias de dos clases. `RelojSimple1291` es una instancia de `Reloj`. `RelojCalculadora1515` es una instancia de `RelojCalculadora`. Aunque las operaciones de `Reloj` también son aplicables a `RelojCalculadora1515`, esta última no es una instancia de la anterior.

3. Las cadenas subrayadas también se usan para representar a los localizadores de recursos uniformes (URL). Para mejorar la legibilidad no usamos tipos subrayados en el texto, sino que usamos el mismo tipo para indicar las instancias y las clases. En general, esta ambigüedad puede resolverse examinando el contexto. Sin embargo, en los diagramas UML siempre usamos tipos subrayados para distinguir las instancias de las clases.

El objeto receptor hace corresponder el nombre del mensaje con alguna de sus operaciones y le pasa los argumentos a la operación. Cualquier resultado se regresa al objeto que lo envió.

Por ejemplo, en la figura 2-13 el objeto `Reloj` calcula la hora actual obteniendo la hora de Greenwich a partir del objeto `Hora`, y la diferencia de horas a partir del objeto `ZonaHoraria`, enviando los mensajes `ObtenerHora()` y `ObtenerIncrementoHora()`, respectivamente.

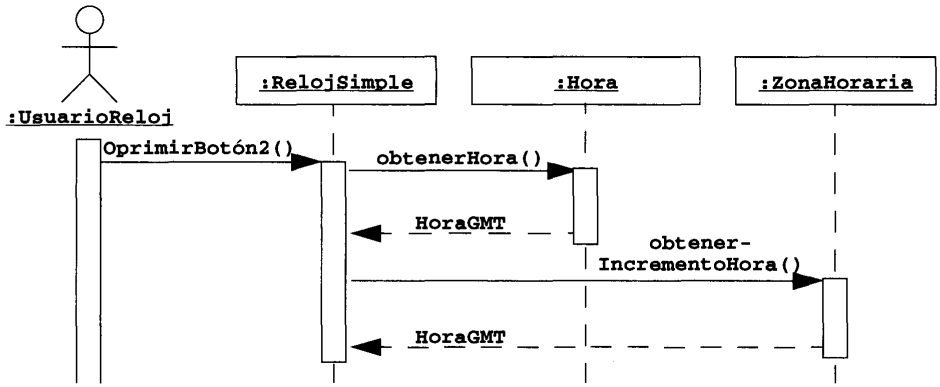


Figura 2-13 Ejemplo de envío de mensajes (diagrama de secuencia UML). El objeto `Reloj` envía el mensaje `ObtenerHora()` al objeto `Hora` para consultar la hora actual de Greenwich. Luego envía el mensaje `ObtenerIncrementoHora()` al objeto `ZonaHoraria` para consultar la diferencia que hay que añadir a la hora de Greenwich. Las flechas de guiones representan los resultados que se envían de regreso a quien envió el mensaje.

Los eventos y los mensajes son instancias: representan ocurrencias concretas en el sistema. Las clases de evento son abstracciones que describen grupos de eventos para los cuales tiene una respuesta común el sistema. En la práctica, el término "evento" puede referirse a instancias o clases. Esta ambigüedad se resuelve examinando el contexto en el que se usa el término.

2.3.6 Modelado orientado a objetos

El **dominio de aplicación** representa todos los aspectos del problema del usuario. Esto incluye el ambiente físico, los usuarios y otras personas, sus procesos de trabajo, etc. Es crítico que los analistas y desarrolladores comprendan el dominio de aplicación de un sistema para realizar con efectividad la tarea que pretenden. Tome en cuenta que el dominio de aplicación cambia a lo largo del tiempo conforme cambian los procesos de trabajo y las personas.[4]

4. El dominio de aplicación a veces se divide adicionalmente en dominio de usuario y dominio de cliente. El dominio de cliente incluye los asuntos relevantes para el cliente, como el costo de operación del sistema, el impacto del sistema en el resto de la organización. El dominio de usuario incluye los asuntos relevantes para el usuario final, como funcionalidad, facilidad de aprendizaje y de uso.

El **dominio de solución** es el espacio de todos los sistemas posibles. El dominio de solución es mucho más rico y volátil que el dominio de aplicación. Esto se debe a las tecnologías emergentes (también llamadas facilitadores de tecnología), a cambios conforme madura la tecnología de implementación o a una mejor comprensión de la tecnología de implementación por parte de los desarrolladores cuando construyen el sistema. El modelado del dominio de solución representa las actividades de diseño del sistema y diseño de objetos del proceso de desarrollo. Observe que la entrega del sistema puede cambiar el dominio de aplicación conforme los usuarios desarrollan nuevos procesos de trabajo para acomodar el sistema.

El **análisis orientado a objetos** está interesado en el modelado del dominio de aplicación. El **diseño orientado a objetos** está interesado en el modelado del dominio de solución. Ambas actividades de modelado usan las mismas representaciones (es decir, clases y objetos). En el análisis y diseño orientados a objetos, el modelo del dominio de aplicación también es parte del modelo del sistema. Por ejemplo, un sistema de control de tráfico aéreo tiene una clase `ControladorTráfico` para representar a los usuarios individuales, sus preferencias e información de bitácora. El sistema también tiene una clase `Aeronave` para representar la información asociada con la aeronave a la que se le está dando seguimiento. `ControladorTráfico` y `Aeronave` son conceptos del dominio de aplicación que están codificados dentro del sistema (figura 2-14).

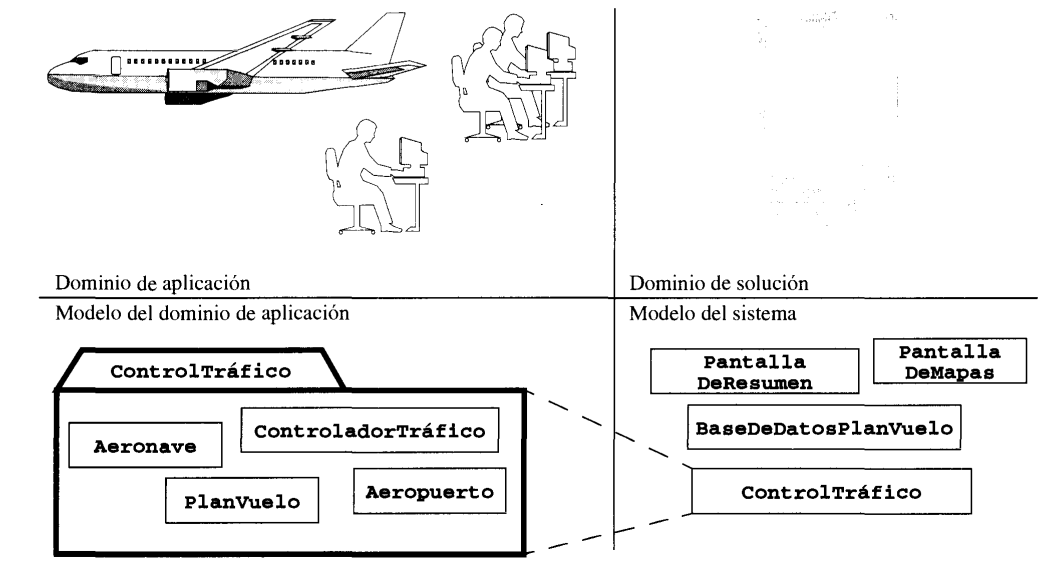


Figura 2-14 El modelo del dominio de aplicación representa entidades del ambiente que son relevantes para un sistema de control de tráfico aéreo (es decir, aeronaves, controladores de tráfico). El modelo del sistema representa a las entidades que son parte del sistema (por ejemplo, desplegado de mapas, base de datos de planes de vuelo). En el análisis y diseño orientados a objetos el modelo de dominio de aplicación también es parte del modelo del sistema. En esta figura, un ejemplo es el paquete `ControlTráfico` que aparece en ambos modelos. (Para mayores detalles vea el capítulo 5, *Análisis*.)

El modelado del dominio de aplicación y del dominio de solución con una sola notación tiene ventajas y desventajas. Por un lado puede ser poderoso: las clases del dominio de solución que representan conceptos de la aplicación pueden ser seguidas hacia el dominio de aplicación. Además, estas clases pueden ser encapsuladas en subsistemas independientes de otros conceptos de instrumentación (por ejemplo, la interfaz de usuario y la tecnología de base de datos) y empacados en una herramienta reutilizable de clases de dominio. Por otro lado, el uso de una sola notación puede provocar confusión, debido a que elimina la distinción entre el mundo real y su modelo. El dominio de sistema debe ser más simple y estar orientado hacia la solución. Para resolver este asunto usamos una sola notación y, cuando hay ambigüedad, hacemos distinciones entre los dos dominios. En la mayoría de los casos hacemos referencia al modelo (por ejemplo, "una `Aeronave` está compuesta de `Manifiesto` y `PlanVuelo`" es una declaración acerca del modelo).

2.3.7 Falsificación y elaboración de prototipos

Un modelo es una simplificación de la realidad en el sentido de que se ignoran los detalles irrelevantes. Sin embargo, es necesario representar los detalles relevantes. La **falsificación** [Popper, 1992] es el proceso de demostrar que se han representado los detalles relevantes en forma incorrecta o no se han representado, esto es, que el modelo no corresponde a la realidad que se supone que representa.

El proceso de falsificación es bien conocido en otras ciencias: los investigadores proponen diferentes modelos de una realidad, los cuales son aceptados de manera gradual conforme una cantidad cada vez mayor de datos apoya al modelo, pero que son rechazados cuando se encuentra un ejemplo en contra. El modelo geocéntrico del universo de Ptolomeo fue falsificado (a la larga) a favor del modelo heliocéntrico copernicano una vez que fueron aceptados los datos de Galileo. Luego el modelo heliocéntrico copernicano fue falsificado cuando se descubrieron otras galaxias y se añadió el concepto de galaxia al modelo.

También podemos aplicar la falsificación al desarrollo de sistemas de software. Por ejemplo, una técnica para el desarrollo de un sistema es la elaboración de un **prototipo**: cuando se diseña la interfaz de usuario, los desarrolladores construyen un prototipo que sólo simula la interfaz de usuario de un sistema. Luego se presenta el prototipo a los usuarios potenciales para que lo evalúen, esto es, que lo falsifiquen, y se modifica con posterioridad. En las primeras iteraciones de este proceso es probable que los desarrolladores desechen el prototipo inicial a consecuencia de la retroalimentación de los usuarios. En otros términos, los usuarios falsifican el prototipo inicial, un modelo del sistema futuro, debido a que no representa con precisión los detalles relevantes.

Observe que sólo es posible demostrar que un modelo es incorrecto. Aunque en algunos casos es posible demostrar matemáticamente que dos modelos son equivalentes, no es posible mostrar que alguno de ellos represente correctamente a la realidad. Por ejemplo, las técnicas de verificación formales pueden permitir que los desarrolladores muestren que una implementación de software específica sea consistente con una especificación formal. Sin embargo, sólo las pruebas de campo y el uso amplio pueden indicar que un sistema satisface las necesidades del cliente. En cualquier momento los modelos de sistema pueden ser falsificados debido a cambios en los requerimientos, en la tecnología de implementación o en el ambiente.

2.4 Una vista más profunda al UML

Ahora describimos con más detalle los cinco diagramas UML principales que usamos en este libro.

- Los **diagramas de caso de uso** representan la funcionalidad del sistema desde el punto de vista del usuario. Definen las fronteras del sistema (sección 2.4.1).
- Los **diagramas de clase** se usan para representar la estructura de un sistema en términos de objetos con sus atributos y relaciones (sección 2.4.2).
- Los **diagramas de secuencia** representan el comportamiento del sistema en términos de interacciones entre un conjunto de objetos. Se usan para identificar objetos en los dominios de aplicación e implementación (sección 2.4.3).
- Los **diagramas de gráfica de estado** se usan para representar el comportamiento de objetos no triviales (sección 2.4.4).
- Los **diagramas de actividad** son diagramas de flujo que se usan para representar el flujo de datos o el flujo de control a través de un sistema (sección 2.4.5).

2.4.1 Diagramas de caso de uso

Casos de uso y actores

Los **actores** son entidades externas que interactúan con el sistema. Ejemplos de actores incluyen un papel de usuario (por ejemplo, un administrador de sistema, un cliente de banco, un cajero de banco) u otro sistema (por ejemplo, una base de datos central, una línea de fabricación). Los actores tienen nombres y descripciones únicos.

Los **casos de uso** describen el comportamiento de un sistema tal como es visto desde el punto de vista de un actor. Al comportamiento descrito por el modelo de caso de uso también se le llama **comportamiento externo**. Un caso de uso describe una función proporcionada por el sistema como un conjunto de eventos que producen un resultado visible para los actores. Los actores inician un caso de uso para acceder a la funcionalidad del sistema. Luego el caso de uso puede iniciar otros casos de uso y recopilar más información de los actores. Cuando los actores y casos de uso intercambian información, se dice que se **comunican**. Posteriormente veremos que representamos esos intercambios con relaciones de comunicación.

Por ejemplo, en un sistema para el manejo de accidentes [FRIEND, 1994], los oficiales de campo, como el policía o el bombero, tienen acceso a una computadora inalámbrica que les permite interactuar con un despachador. El despachador, a su vez, puede visualizar el estado actual de todos sus recursos, como los automóviles o camiones de la policía, en una pantalla de computadora y despachar un recurso dando comandos desde una estación de trabajo. En este ejemplo, `OficialCampo` y `Despachador` son actores.

La figura 2-15 muestra al actor `OficialCampo` que llama al caso de uso `ReporteEmergencia` para notificar al actor `Despachador` que hay una nueva emergencia. En respuesta, el `Despachador` llama al caso de uso `AbrirIncidente` para crear un reporte de incidente e iniciar el manejo del incidente. El `Despachador` captura información preliminar dada por el `OficialCampo` en la base de datos de incidentes y con el caso de uso `AsignarRecursos` ordena que vayan a la escena unidades adicionales.

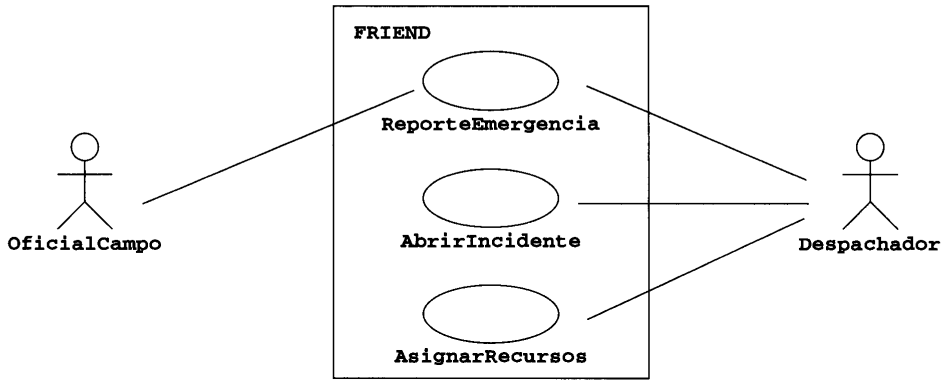


Figura 2-15 Un ejemplo de un diagrama de caso de uso UML: Inicio de incidente en un sistema de manejo de accidentes. Las asociaciones entre actores y casos de uso representan flujo de información. En UML estas asociaciones son bidireccionales: pueden representar al actor iniciando un caso de uso (por ejemplo, `OficialCampo` inicia `ReporteEmergencia`) o un caso de uso que proporcione información a un actor (por ejemplo, `ReporteEmergencia` notifica a `Despachador`).

Para describir un caso de uso usamos una plantilla compuesta de seis campos (vea también la figura 2-16):

- El **nombre** del caso de uso es único en todo el sistema para que los desarrolladores (y participantes en el proyecto) puedan hacer referencia al caso de uso sin ambigüedad.
- Los **actores participantes** son los actores que interactúan con el caso de uso.
- Las **condiciones iniciales** describen las condiciones que necesitan satisfacerse antes de que se inicie el caso de uso.
- El **flujo de eventos** describe la secuencia de acciones del caso de uso y estarán numeradas para su referencia. El caso común (es decir, los casos que suceden con frecuencia) y los casos excepcionales (es decir, casos que ocurren rara vez, como los errores y condiciones inusuales) se describen por separado en diferentes casos de uso para efectos de claridad.
- Las **condiciones de salida** describen las condiciones que se satisfacen después de que termina el caso de uso.
- Los **requerimientos especiales** son aquellos que no están relacionados con la funcionalidad del sistema. Incluyen restricciones sobre el desempeño del sistema, su implementación, las plataformas de hardware en las que se ejecuta, etc. Los requerimientos especiales se describen con más detalle en el capítulo 4, *Obtención de requerimientos*.

Los casos de uso se escriben en lenguaje natural. Esto permite que los desarrolladores los usen para comunicarse con los clientes y los usuarios, quienes, por lo general, no tienen un conocimiento amplio de las notaciones de ingeniería de software. El uso del lenguaje natural también permite que los participantes de otras disciplinas comprendan los requerimientos del sistema.

Nombre del caso de uso	`ReporteEmergencia`
Actor participante	Llamado por `OficialCampo` Se comunica con `Despachador`
Condición inicial	1. El `OficialCampo` activa la función "ReporteEmergencia" de su terminal. FRIEND responde presentando un formulario al oficial.
Flujo de eventos	2. El `OficialCampo` llena el formulario, seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia. Una vez que ha llenado el formulario, el `OficialCampo` lo envía y en ese momento se le notifica al `Despachador`. 3. El `Despachador` revisa la información enviada y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. El `Despachador` selecciona una respuesta y da un acuse de recibo del reporte de emergencia.
Condición de salida	4. El `OficialCampo` recibe el acuse de recibo y la respuesta seleccionada.
Requerimientos especiales	Se da acuse de recibo del reporte del `OficialCampo` en menos de 30 segundos. La respuesta seleccionada llega antes de que transcurran 30 segundos a partir de que la envía el `Despachador`.

Figura 2-16 Un ejemplo de un caso de uso: el caso de uso `ReporteEmergencia`.

Escenarios

Un caso de uso es una abstracción que describe todos los escenarios posibles que involucran la funcionalidad que se describe. Un **escenario** es una instancia de un caso de uso que describe un conjunto de acciones concretas. Los escenarios se usan como ejemplos para ilustrar casos comunes. Su enfoque es la comprensión. Los casos de uso se utilizan para describir todos los casos posibles. Su enfoque es la totalidad. Describimos un escenario usando una plantilla con tres campos:

- El **nombre** del escenario permite que nos refiramos a él sin ambigüedad. El nombre de un escenario está subrayado para indicar que es una instancia.
- El campo **instancias de actor participante** indica cuáles instancias de actor están involucradas en este escenario. Las instancias de actor también tienen nombres subrayados.
- El **flujo de eventos** de un escenario describe la secuencia de eventos paso a paso.

Observe que no hay necesidad de condiciones iniciales o de salida en los escenarios. Las condiciones iniciales y de salida son abstracciones que permiten que los desarrolladores describan un rango de condiciones bajo las cuales se llama a un caso de uso. Dado que un escenario sólo describe un flujo único de eventos, tales condiciones no son necesarias (figura 2-17).

Los diagramas de caso de uso pueden incluir cuatro tipos de relaciones: comunicación, inclusión, extensión y generalización. Luego describiremos estas relaciones con mayor detalle.

Nombre del escenario	bodegaEnLlamas
Instancias de actores participantes	roberto, alicia: OficialCampo juan: Despachador
Flujo de eventos	1. Roberto, manejando por la calle principal en su patrulla, observa que sale humo de una bodega. Su compañera, Alicia, activa la función "Reporte de emergencia" en su laptop FRIEND. 2. Alicia captura la dirección del edificio, una breve descripción de su ubicación (es decir, esquina noroeste) y un nivel de emergencia. Además de un carro de bomberos, solicita varias ambulancias, ya que el área parece estar algo atareada. Confirma lo capturado y espera el acuse de recibo. 3. Juan, el Despachador, es alertado de que hay una emergencia mediante un sonido de su estación de trabajo. Revisa la información enviada por Alicia y da el acuse de recibo del reporte. Asigna un carro de bomberos y dos ambulancias al lugar del Incidente y envía la hora estimada de llegada (ETA, por sus siglas en inglés) a Alicia. 4. Alicia recibe el acuse de recibo y la ETA.

Figura 2-17 El escenario de bodegaEnLlamas para el caso de uso ReporteEmergencia.

Relaciones de comunicación

Los actores y los casos de uso se **comunican** cuando intercambian información entre ellos. Las relaciones de comunicación se muestran con una línea continua entre los símbolos de actor y caso de uso. En la figura 2-15, los actores OficialCampo y Despachador se comunican con el caso de uso ReporteEmergencia. Sólo el actor Despachador se comunica con los casos de uso AbrirIncidente y AsignarRecursos. Las relaciones de comunicación entre actores y casos de uso pueden emplearse para indicar acceso a funcionalidad. En el caso de nuestro ejemplo, al OficialCampo y al Despachador se les proporcionan diferentes interfaces ante el sistema y tienen acceso a funcionalidades diferentes.

Relaciones de inclusión

Cuando se describe un sistema complejo, su modelo de casos de uso puede llegar a ser muy complicado y contener redundancias. La complejidad del modelo se reduce identificando las cosas comunes que hay en diferentes casos de uso. Por ejemplo, supongamos que el Despachador puede oprimir una tecla en cualquier momento para tener acceso a la ayuda. Esto puede modelarse mediante un caso de uso AyudaDespachador que está incluido en los casos de uso AbrirIncidente y AsignarRecursos (y cualesquiera otros casos de uso a los que tenga acceso el Despachador). El modelo resultante describe solamente una vez la funcionalidad de AyudaDespachador, reduciendo de esta forma la complejidad. Dos casos de uso están relacionados por una relación de inclusión si alguno de ellos incluye al segundo en su flujo de eventos. En UML las **relaciones de inclusión** se muestran mediante una flecha de guiones que se inicia en el caso de uso que incluye al otro (vea la figura 2-18). Las relaciones de inclusión están etiquetadas con el texto <<incluye>>.

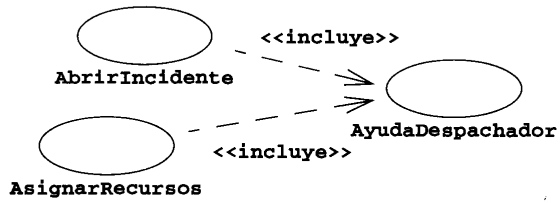


Figura 2-18 Un ejemplo de una relación `<<incluye>>` (diagrama de caso de uso UML).

Representamos las relaciones de inclusión en el caso de uso mismo de dos formas. Si el caso de uso incluido puede incluirse en cualquier momento del flujo de eventos (por ejemplo, el caso de uso `AyudaDespachador`), indicamos la inclusión en la sección *Requerimientos especiales* del caso de uso. Si el caso de uso se llama explícitamente durante un evento, indicamos la inclusión en el flujo de eventos.

Relaciones extendidas

Las **relaciones extendidas** son un medio alterno para reducir la complejidad en el modelo del caso de uso. Un caso de uso puede extender a otro caso de uso mediante la adición de eventos. Una relación extendida indica que una instancia del caso de uso extendido puede incluir (bajo determinadas condiciones) el comportamiento especificado por el caso de uso que extiende. Una aplicación típica de las relaciones extendidas es la especificación de un comportamiento excepcional. Por ejemplo (figura 2-19), supongamos que la conexión de red entre el `Despachador` y el `OficialCampo` puede interrumpirse en cualquier momento (por ejemplo, si el `OficialCampo` entra a un túnel). El caso de uso `ConexiónPerdida` describe el conjunto de eventos realizados por el sistema y los actores mientras no se tiene conexión. `ConexiónPerdida` extiende los casos de uso `AbrirIncidente` y `AsignarRecursos`. La separación del comportamiento excepcional con respecto al comportamiento común nos permite escribir casos de uso más cortos y más enfocados.

En la representación textual de un caso de uso, representamos las relaciones extendidas como condiciones iniciales del caso de uso que se extiende. Por ejemplo, las relaciones extendidas que se muestran en la figura 2-19 están representadas como condición inicial en el caso de uso `ConexiónPerdida` (figura 2-20).

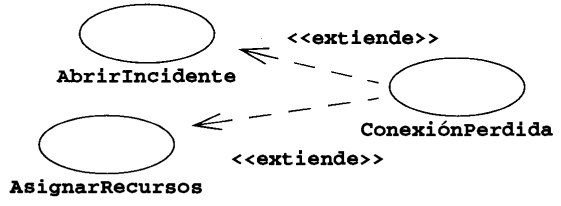


Figura 2-19 Un ejemplo de una relación `<<extiende>>` (diagrama de caso de uso UML).

Nombre del caso de uso	`ConexiónPerdida`
Actor participante	Se comunica con el `OficialCampo` y `Despachador`
Condición inicial	Este caso de uso extiende los casos de uso `AbrirIncidente` y `AsignarRecursos`. Lo inicia el sistema cada vez que se pierde la conexión de red entre el `OficialCampo` y el `Despachador`
Flujo de eventos	1. ...

Figura 2-20 Representación textual de las relaciones extendidas de la figura 2-19.

La diferencia entre las relaciones de inclusión y extendidas es la ubicación de la dependencia. Supongamos que añadimos varios casos de uso nuevos para el actor `Despachador`. Si modelamos la función `AyudaDespachador` con relaciones de inclusión, cada nuevo caso de uso necesitará incluir el caso de uso `AyudaDespachador`. Si en vez de ello usamos relaciones extendidas, sólo necesita modificarse el caso de uso `AyudaDespachador` para extender los casos de uso adicionales. En general, los casos de excepción, como la ayuda, errores y otras situaciones inesperadas, se modelan con relaciones extendidas. Los casos de uso que describen comportamientos compartidos por lo común por un conjunto fijo de casos de uso se modelan con relaciones de inclusión.

Relaciones de generalización

Las relaciones de generalización y especialización son un tercer mecanismo para reducir la complejidad de un modelo. Un caso de uso puede especializar a otro más general añadiendo más detalle. Por ejemplo, se requiere que los `OficialCampo` se autentifiquen antes de que puedan usar FRIEND. Durante las primeras etapas de la obtención de requerimientos, la autentificación se modela como un caso de uso `Autentificar` de alto nivel. Después, los desarrolladores describen el caso de uso `Autentificar` con mayor detalle y permiten varias plataformas de hardware diferentes. Esta actividad de refinamiento da como resultado dos casos de uso adicionales, `AutentificarConContraseña`, que permite que los `OficialCampo` se registren sin ningún hardware específico, y `AutentificarConTarjeta`, que permite que los `OficialCampo` se registren usando una tarjeta inteligente. Los dos nuevos casos de uso están representados como especializaciones del caso de uso `Autentificar` (figura 2-21).

Aplicación de los diagramas de casos de uso

Los casos de uso y los actores definen las fronteras del sistema. Se desarrollan durante la obtención de requerimientos, con frecuencia con el cliente y los usuarios. Durante el análisis, los casos de uso se refinan y corrigen cuando son revisados por una audiencia más amplia que incluye a los desarrolladores, y se validan contra situaciones reales.

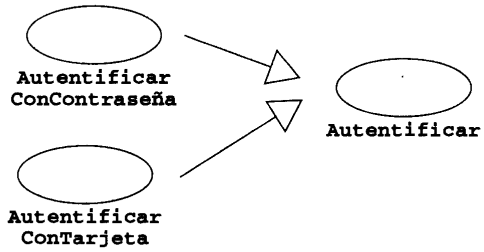


Figura 2-21 Un ejemplo de una relación de generalización (diagrama de caso de uso UML). El caso de uso `Autentificar` es un caso de uso de alto nivel que describe, en términos generales, el proceso de autentificación. `AutentificarConContraseña` y `AutentificarConTarjeta` son dos especializaciones de `Autentificar`.

2.4.2 Diagramas de clase

Clases y objetos

Los diagramas de clase describen la estructura del sistema desde el punto de vista de clases y objetos. Las **clases** son abstracciones que especifican los atributos y comportamientos de un conjunto de objetos. Los **objetos** son entidades que encapsulan estado y comportamiento. Cada objeto tiene una identidad: se puede hacer referencia a él de manera individual y es distinguible con respecto a otros objetos.

En UML las clases y objetos se muestran mediante cuadros que incluyen tres compartimientos. El compartimiento superior muestra el nombre de la clase u objeto. El compartimiento central muestra sus atributos y el compartimiento inferior muestra sus operaciones. Los compartimientos de atributos, y operaciones se pueden omitir por claridad. Los nombres de objetos están subrayados para indicar que son instancias. Por convención, los nombres de clase comienzan con una letra mayúscula. En los diagramas de objetos se les puede dar nombre a los objetos (seguido del nombre de la clase) para facilitar su referencia. En ese caso, los nombres comienzan con minúscula. En el ejemplo de FRIEND (figuras 2-22 y 2-23), Roberto y Alicia son oficiales de campo, y están representados en el sistema como objetos `OficialCampo` llamados `roberto:OficialCampo` y `alicia:OficialCampo`. `OficialCampo` es una clase que describe a todos los objetos `OficialCampo` y, en cambio, Roberto y Alicia están representados por dos objetos `OficialCampo` individuales.

En la figura 2-22 la clase `OficialCampo` tiene dos atributos: un `nombre` y un `númeroIdentificación`. Esto indica que todos los objetos `OficialCampo` tienen esos dos atributos. En la figura 2-23 los objetos `roberto:OficialCampo` y `alicia:OficialCampo` tienen valores específicos para esos atributos: "Roberto D." y "Alicia W.", respectivamente. En la figura 2-22 el atributo `OficialCampo.nombre` es de tipo `String`, lo que indica que sólo se pueden asignar instancias de `String` al atributo `OficialCampo.nombre`. El tipo de un atributo se usa para especificar el rango válido de valores que puede tener un atributo. Observe que cuando los tipos de atributo no son esenciales para la definición del sistema, las decisiones sobre el tipo de atributo se pueden dejar para después hasta el diseño del objeto. Esto permite que los desarrolladores se concentren en la funcionalidad del sistema y se minimice la cantidad de cambios triviales cuando se revisa la funcionalidad del sistema.

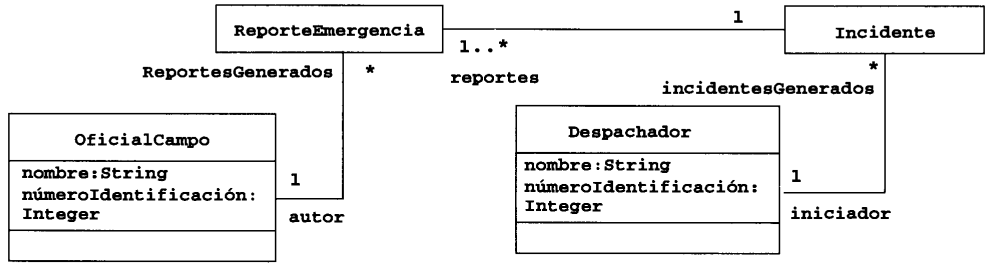


Figura 2-22 Un ejemplo de un diagrama de clase UML: las clases que participan en el caso de uso `ReporteEmergencia`. Por lo general, la información de tipo detallado se omite hasta el diseño del objeto (vea el capítulo 7, *Diseño de objetos*).

Asociaciones y vínculos

Un **vínculo** representa una conexión entre dos objetos. Las **asociaciones** son relaciones entre clases y representan grupos de vínculos. Cada objeto `OficialCampo` también tiene una lista de `ReporteEmergencia` que ha sido escrita por el `OficialCampo`. En la figura 2-22 la línea que hay entre la clase `OficialCampo` y la clase `ReporteEmergencia` es una asociación. En la figura 2-23 la línea que hay entre el objeto `alicia:OficialCampo` y el objeto `reporte_1291:ReporteEmergencia` es un vínculo. Este vínculo representa un estado que se conserva en el sistema para indicar que `alicia:OficialCampo` generó el `reporte_1291:ReporteEmergencia`.

Papeles

Cada extremo de una asociación puede etiquetarse con un texto llamado **papel**. En la figura 2-22 los papeles de la asociación entre las clases `ReporteEmergencia` y `OficialCampo` son `autor` y `reportesGenerados`. El etiquetado de los extremos de la asociación con papeles nos permite distinguir entre varias asociaciones que se originan en una clase. Además, los papeles aclaran el propósito de la asociación.

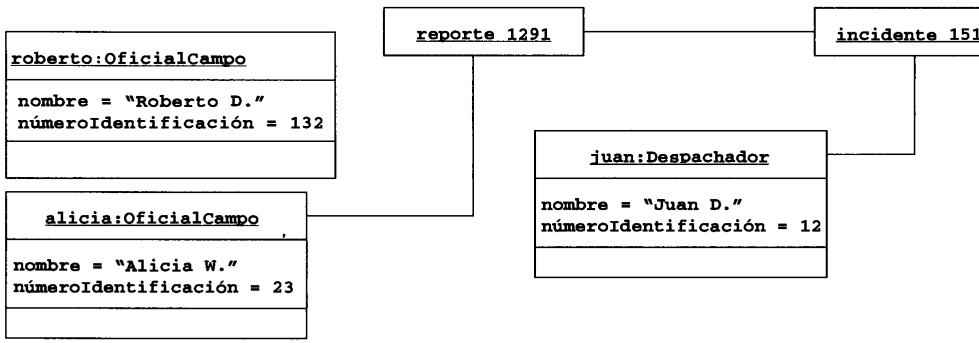


Figura 2-23 Un ejemplo de un diagrama de objetos UML: los objetos que participan en el escenario `bodegaEnLlamas`.

Multiplicidad

Cada extremo de una asociación puede ser etiquetado con un conjunto de enteros que indica la cantidad de vínculos que se pueden originar legítimamente a partir de una instancia de la clase conectada al extremo de la asociación. El extremo de la asociación `autor` tiene una multiplicidad de `1`. Esto significa que todos los `ReporteEmergencia` están escritos por exactamente un `OficialCampo`. En otras palabras, cada objeto `ReporteEmergencia` tiene exactamente un vínculo hacia un objeto de la clase `OficialCampo`. La multiplicidad del extremo de la asociación del papel `ReportesGenerados` es "muchos", indicado por un asterisco. La multiplicidad "muchos" es una abreviatura de `0..n`. Esto significa que cualquier `OficialCampo` puede ser el autor de cero o más `ReporteEmergencia`.

Clase de asociación

Las asociaciones son similares a las clases en que pueden tener atributos y operaciones asociados a ellas. A una asociación de éstas se le llama **clase de asociación**, se muestra con un símbolo de clase que contiene los atributos y operaciones y está conectada con el símbolo de asociación mediante una línea de guiones. Por ejemplo, en la figura 2-24 la asignación de un `OficialCampo` con un `Incidente` se modela con una clase de asociación con los atributos `papel` y `tiempoNotificación`.

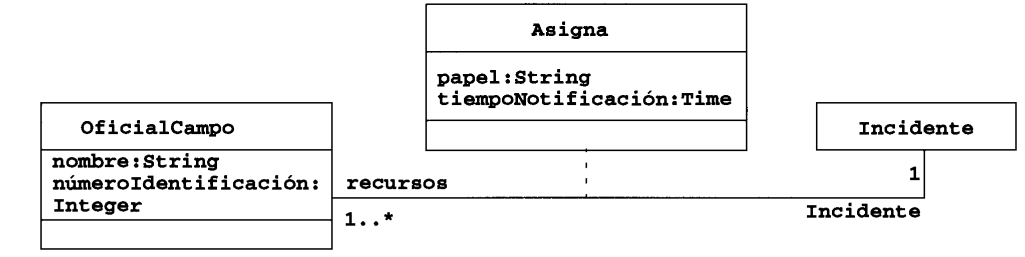


Figura 2-24 Un ejemplo de una clase de asociación (diagrama de clase UML).

Cualquier clase de asociación puede ser transformada a una clase y asociaciones simples, como se muestra en la figura 2-25. Aunque ambas representaciones son similares, la representación de la clase de asociación es más clara: no puede existir una asociación sin las clases que vincula. En forma similar, el objeto `Asignación` no puede existir sin los objetos `OficialCampo` e `Incidente`. Aunque la figura 2-25 tiene la misma información, ese diagrama requiere un examen cuidadoso de la multiplicidad de la asociación. En el capítulo 5, *Análisis*, examinamos estos compromisos del modelado.

Agregación

Las asociaciones se usan para representar un amplio rango de conexiones entre un conjunto de objetos. En la práctica sucede a menudo un caso especial de asociación: la composición. Por ejemplo, un `Estado` contiene muchas `Regiones`, las cuales, a su vez, contienen muchos `Pueblos`. Una `EstaciónPolicía` está compuesta por muchos `OficialPolicía`. Otro ejemplo es un `Directorio` que contiene varios `Archivo`. Tales relaciones podrían modelarse usando una

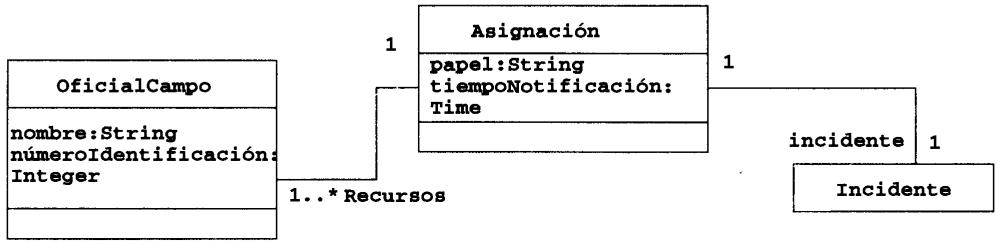


Figura 2-25 Un modelo alterno para la `Asignación` (diagrama de clase UML).

asociación de uno a muchos. En vez de ello, UML proporciona el concepto de una **agregación** para indicar la composición. Una agregación se indica mediante una línea simple con un rombo en el extremo del contenedor de la asociación (vea la figura 2-26). Aunque las asociaciones de uno a muchos y las agregaciones pueden usarse en forma alterna, se prefieren las agregaciones debido a que enfatizan los aspectos jerárquicos de la relación. Por ejemplo, en la figura 2-26 los `OficialPolicía` son parte de la `EstaciónPolicía`.

Generalización

La **generalización** es la relación entre una clase general y una o más clases más especializadas. La generalización nos permite describir todos los atributos y operaciones que son comunes para un conjunto de clases. Por ejemplo, `OficialCampo` y `Despachador` tienen los atributos `nombre` y `númeroIdentificación`. Sin embargo, `OficialCampo` tiene una asociación con `ReporteEmergencia`, mientras que `Despachador` tiene una asociación con `Incidente`. Los atributos comunes de `OficialCampo` y `Despachador` pueden modelarse introduciendo una clase *`OficialPolicía`*, que es especializada por las clases `OficialCampo` y `Despachador` (vea la figura 2-27). A *`OficialPolicía`*, la generalización, se le llama **superclase**. A `OficialCampo` y `Despachador`, las especializaciones, se les llama **subclases**. Las subclases **heredan** los atributos y operaciones de su clase de origen. Las clases abstractas (definidas en la sección 2.3.4) se distinguen con respecto a las clases concretas poniendo en *cursivas* el nombre de las clases abstractas. En la figura 2-27 *`OficialPolicía`* es una clase abstracta. Las clases abstractas se usan en el modelado orientado a objetos para clasificar conceptos relacionados, reduciendo, por lo tanto, la complejidad general del modelo.

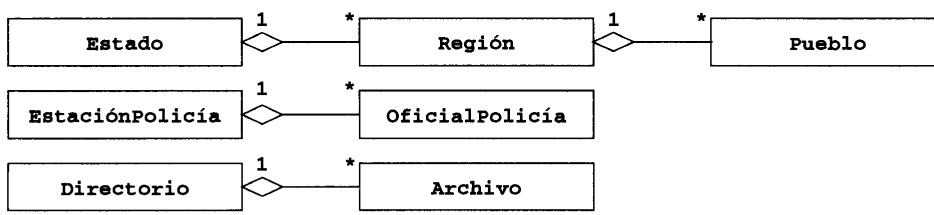


Figura 2-26 Ejemplos de agregaciones (diagramas de clase UML). Un `Estado` contiene muchas `Región`, las cuales, a su vez, contienen muchos `Pueblo`. Una `EstaciónPolicía` tiene muchos `OficialPolicía`. Un `Directorio` de un sistema de archivos contiene muchos `Archivo`.

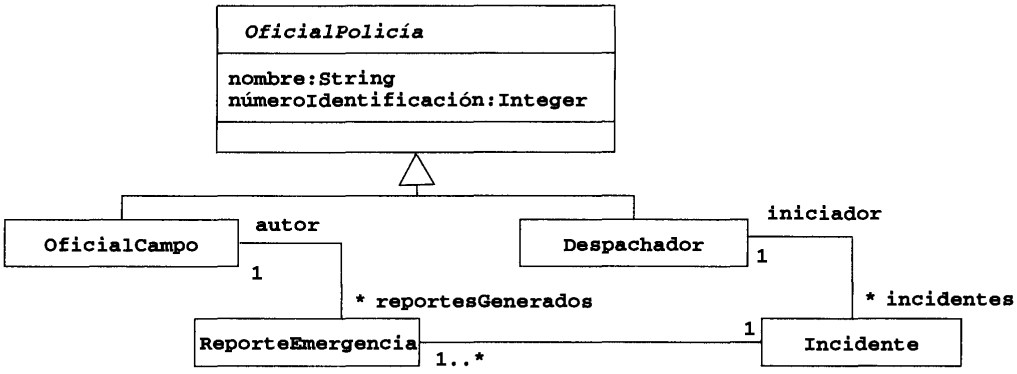


Figura 2-27 Un ejemplo de generalización (diagrama de clase UML). *OficialPolicía* es una clase abstracta que define los atributos y operaciones comunes de las clases `OficialCampo` y `Despachador`.

El comportamiento de los objetos se especifica mediante **operaciones**. Un conjunto de operaciones representa un **servicio** proporcionado por una clase particular. Un objeto solicita la ejecución de una operación de otro objeto enviándole un **mensaje**. El mensaje concuerda con un **método** definido por la clase a la que pertenece el objeto receptor o por cualquiera de sus superclases. Las operaciones de una clase son los servicios públicos que proporciona la clase. Los métodos de su clase son las implementaciones de estas operaciones.

La distinción entre operaciones y métodos permite una separación más clara entre el mecanismo para solicitar un servicio y la ubicación en donde se le proporciona. Por ejemplo, la clase `Incidente` de la figura 2-28 define una operación llamada `asignarRecurso()`, la cual teniendo a un `OficialCampo` crea una asociación entre el `Incidente` recibido y el `Recurso` especificado. La operación `asignarRecurso()` también puede tener un efecto lateral, como el envío de una notificación al `Recurso` recién asignado. La operación `Cerrar()` de `Incidente` es responsable del cierre del `Incidente`. Esto incluye ir a todos los recursos que hayan sido asignados al incidente a lo largo del tiempo y recopilar sus reportes.

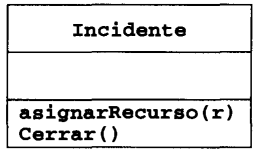


Figura 2-28 Ejemplos de operaciones proporcionadas por la clase `Incidente` (diagrama de clase UML).

Aplicación de los diagramas de clase

Los diagramas de clase se usan para describir la estructura de un sistema. Durante el análisis, los ingenieros de software construyen diagramas de clase para formalizar el conocimiento del dominio de aplicación. Las clases representan objetos participantes que se encuentran en los casos de uso y diagramas de secuencia, y describen sus atributos y operaciones. El propósito de los modelos de análisis es describir el alcance del sistema y descubrir sus fronteras. Por ejemplo, usando el diagrama de clase que se muestra en la figura 2-22, un analista puede examinar la multiplicidad de la asociación entre `OficialCampo` y `ReporteEmergencia` (es decir, un `OficialCampo` puede escribir cero o más `ReporteEmergencia`, pero cada `ReporteEmergencia` es escrito por exactamente un `OficialCampo`) y preguntar al usuario si esto es correcto. ¿Puede haber más de un autor en un `ReporteEmergencia`? ¿Puede haber reportes anónimos? Dependiendo de la respuesta del usuario, el analista podría cambiar el modelo para reflejar el dominio de aplicación. El desarrollo del modelo de análisis se describe en el capítulo 5, *Análisis*.

Los modelos de análisis no se enfocan en la implementación. No están representados conceptos como los detalles de interfaz, la comunicación en red y el almacenamiento en base de datos. Los diagramas de clase se refinan durante el diseño del sistema y el diseño de objetos para incluir clases que representen el dominio de solución. Por ejemplo, el desarrollador añade clases que representan bases de datos, ventanas de interfaz de usuario, adaptadores alrededor de código heredado, optimizaciones, etc. Las clases también se agrupan en subsistemas con interfaces bien definidas. El desarrollo del modelo de diseño se describe en los capítulos 6, *Diseño del sistema*, y 7, *Diseño de objetos*.

2.4.3 Diagramas de secuencia

Los **diagramas de secuencia** describen patrones de comunicación entre un conjunto de objetos interactuantes. Un objeto interactúa con otro objeto enviando **mensajes**. La recepción de un mensaje por parte de un objeto activa la ejecución de una operación, la cual, a su vez, puede enviar mensajes a otros objetos. Se pueden pasar **argumentos** junto con un mensaje y se asocian a los parámetros de la operación que se va a ejecutar en el objeto que los recibe.

Por ejemplo, considere un reloj con dos botones (llamado a partir de aquí Reloj2B). El ajuste del tiempo en Reloj2B requiere que el actor `PropietarioReloj2B` oprima primero ambos botones en forma simultánea, y después de eso Reloj2B entra al modo de ajustar el tiempo. En el modo de ajustar el tiempo, Reloj2B hace parpadear el número que se está cambiando (por ejemplo, horas, minutos, segundos, día, mes y año). En un principio, cuando el `PropietarioReloj2B` inicia el modo de ajustar el tiempo, parpadean las horas. Si el actor oprime el primer botón, parpadea el siguiente número (por ejemplo, si están parpadeando las horas y el actor oprime el primer botón, las horas dejan de parpadear y comienzan a parpadear los minutos). Si el actor oprime el segundo botón, el número que está parpadeando se incrementa en una unidad. Si el número que está parpadeando llega al final de su rango, se le pone al inicio de su rango (por ejemplo, suponiendo que los minutos estén parpadeando y su valor actual sea `59`, el nuevo valor es puesto a `0` si el actor oprime el segundo botón). El actor sale del modo de ajustar tiempo oprimiendo ambos botones en forma simultánea. La figura 2-29 muestra un diagrama de secuencia para un actor que ajusta su Reloj2B un minuto hacia delante.

Cada columna representa un objeto que participa en la interacción. El eje vertical representa el tiempo de arriba hacia abajo. Los mensajes se muestran con flechas. Los rótulos de las

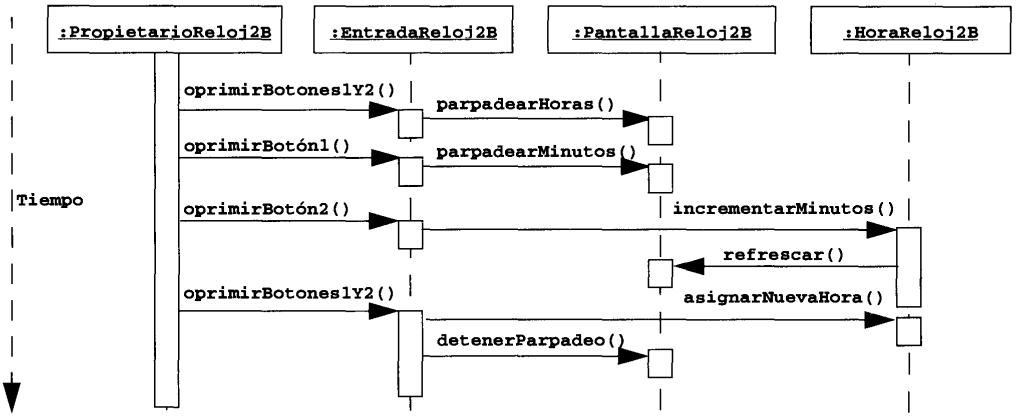


Figura 2-29 Ejemplo de un diagrama de secuencia: ajuste del tiempo en Reloj2B.

flechas representan nombres de mensajes y pueden contener argumentos. Las activaciones (es decir, la ejecución de métodos) se muestran con rectángulos verticales. Los actores se muestran en la columna de la extrema izquierda.

Los diagramas de secuencia pueden usarse para describir una secuencia abstracta (es decir, todas las interacciones posibles) o secuencias concretas (es decir, una interacción posible, como en la figura 2-29). Cuando se describen todas las interacciones posibles, los diagramas de secuencia también proporcionan notaciones para condiciones e iteradores. Una condición en un mensaje se indica por una expresión entre corchetes antes del nombre del mensaje (vea `[i>0] op1()` y `[i<=0] op2()` en la figura 2-30). Si la expresión es cierta se envía el mensaje. La invocación repetitiva de un mensaje se indica con un "*" antes del nombre del mensaje (vea `*op3` en la figura 2-30).

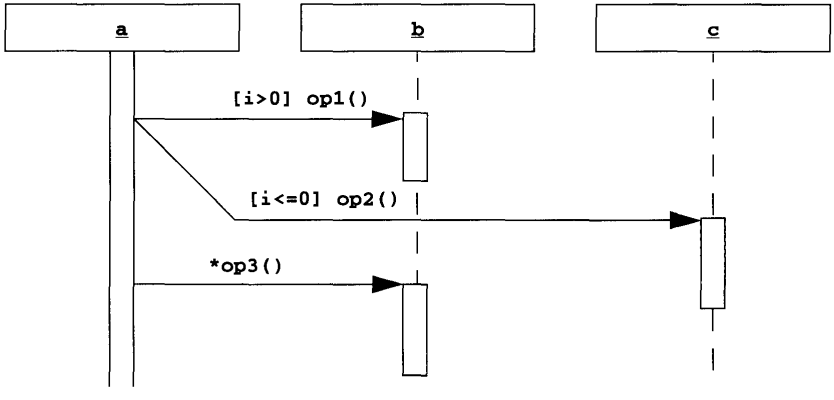


Figura 2-30 Ejemplos de condiciones e iteradores en diagramas de secuencia.

Aplicación de los diagramas de secuencia

Los diagramas de secuencia describen interacciones entre varios objetos. Por lo general, usamos un diagrama de secuencia para describir el flujo de eventos de un caso de uso, identificando los objetos que participan en el caso de uso y asignando partes del comportamiento del caso de uso a los objetos en forma de servicios. Este proceso conduce con frecuencia a refinamientos en el caso de uso (por ejemplo, corrección de descripciones ambiguas, adición de comportamientos faltantes) y, en consecuencia, al descubrimiento de más objetos y más servicios. En el capítulo 5, *Análisis*, describimos en forma detallada el uso de los diagramas de secuencia.

2.4.4 Diagramas de gráfica de estado

Una **gráfica de estado UML** es una notación para la descripción de la secuencia de estados por los que pasa un objeto en respuesta a eventos externos. Las gráficas de estado son extensiones del modelo de máquina de estado finito. Por un lado, las gráficas de estado proporcionan una notación para anidar estados y máquinas de estado (es decir, un estado puede ser descrito por una máquina de estado). Por otro lado, las gráficas de estado proporcionan una notación para el agrupamiento de transiciones con mensajes enviados y condiciones en los objetos. Las gráficas de estado UML están basadas en las gráficas de estado de Harel [Harel, 1987]. Una gráfica de estado UML es equivalente a una máquina de estado de Mealy o Moore.

Un **estado** es una condición que satisface un objeto. Se puede pensar que un estado es una abstracción de los valores de atributo de una clase. Por ejemplo, un objeto `Incidente` en FRIEND puede estar en cuatro estados: `Activo`, `Inactivo`, `Cerrado` y `Archivado` (vea la figura 2-31). Un `Incidente` activo indica una situación que requiere una respuesta (por ejemplo, un incendio en curso, un accidente de tránsito). Un `Incidente` inactivo indica una situación que ya fue atendida pero de la que todavía falta que se escriban los reportes (por ejemplo, ya se apagó el fuego pero todavía no se realizan las estimaciones de daños). Un `Incidente` cerrado indica una situación que ya ha sido atendida y documentada. Un `Incidente` archivado es un `Incidente` cerrado cuya documentación ha sido movida a almacenamiento fuera del sitio.

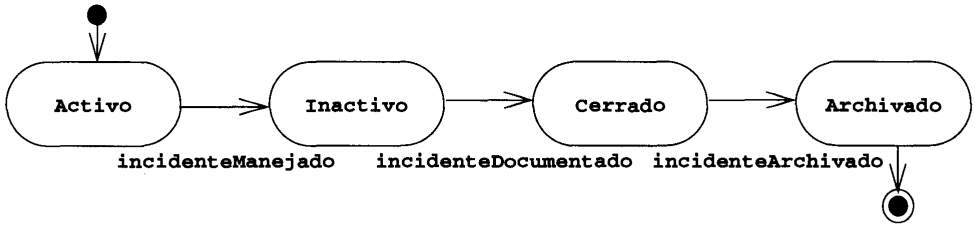


Figura 2-31 Un diagrama de gráfica de estado UML para la clase `Incidente`.

Una **transición** representa cambios de estado activados por eventos, condiciones o tiempo. Por ejemplo, en la figura 2-31 hay tres transiciones: del estado `Activo` hacia el estado `Inactivo`, del estado `Inactivo` hacia el estado `Cerrado` y del estado `Cerrado` hacia el estado `Archivado`.

Un estado se muestra mediante un rectángulo redondeado. Una transición se muestra mediante flechas que conectan dos estados. Los estados se rotulan con su nombre. Un pequeño círculo negro relleno indica el estado inicial. Un círculo que rodea a un pequeño círculo negro relleno indica un estado final.

La figura 2-32 muestra otro ejemplo, una gráfica de estado para el Reloj2B (para el que se construyó un diagrama de secuencia en la figura 2-29). En el nivel de abstracción más alto Reloj2B tiene dos estados, `MedirTiempo` y `AjustarHora`. Reloj2B cambia estados cuando el usuario oprime y suelta ambos botones en forma simultánea. Cuando al Reloj2B se le aplica corriente por primera vez, está en el estado `AjustarHora`. Esto está indicado por el pequeño círculo negro relleno, el cual representa el estado inicial. Cuando al Reloj2B se le acaba la batería, queda fuera de servicio de manera permanente. Esto se indica con un estado final. En este ejemplo las transiciones pueden ser activadas por un evento (por ejemplo, `oprimirBotonesLYR`) o por el paso del tiempo (por ejemplo, `después de 2 min.`). Las acciones pueden estar asociadas con una transición (por ejemplo, emitir un sonido cuando se activa la transición entre `AjustarHora` y `MedirTiempo` en el evento `oprimirBotonesLYR`).

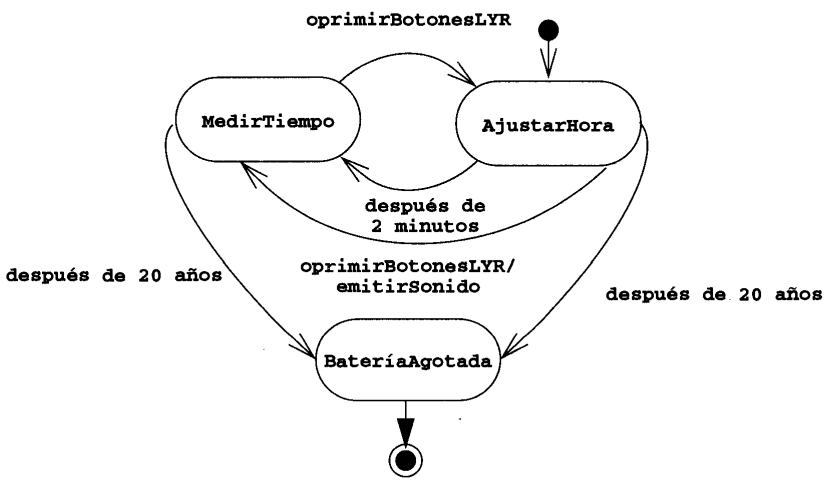


Figura 2-32 Diagrama de gráfica de estado para la función `AjustarHora` de Reloj2B.

El diagrama de gráfica de estado de la figura 2-32 no representa los detalles de la medición o el ajuste del tiempo. Estos detalles han sido abstraídos de la gráfica de estado, y se les puede modelar por separado usando transiciones internas o gráficas de estado anidadas. Las transiciones internas (figura 2-33) son transiciones que permanecen dentro de un solo estado. También pueden tener acciones asociadas con ellas. El inicio y la salida se muestran como transiciones internas, ya que sus acciones no dependen de los estados de origen y destino.

Las gráficas de estado anidadas reducen la complejidad. Pueden usarse en vez de las transiciones internas. En la figura 2-34 el número actual está modelado como un estado anidado, mientras que las acciones que corresponden a la modificación del número actual están modeladas usando transiciones internas. Observe que cada estado podría modelarse como una gráfica de

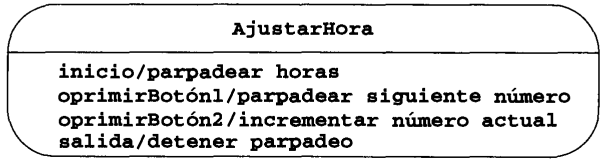


Figura 2-33 Transiciones internas asociadas con el estado `AjustarHora` (diagrama de gráfica de estado UML).

estado anidada (por ejemplo, la gráfica de estado de `ParpadearHoras` podría tener 24 subestados que correspondieran a las horas del día, y las transiciones entre estos estados corresponderían a la opresión del segundo botón).

Aplicación de los diagramas de gráfica de estado

Los diagramas de gráfica de estado se usan para representar el comportamiento no trivial de un subsistema o un objeto. A diferencia de los diagramas de secuencia, que se enfocan en los eventos que tienen un impacto en el comportamiento de un conjunto de objetos, los diagramas de gráfica de estado hacen explícito cuál atributo o conjunto de atributos tienen un impacto en el comportamiento de un objeto individual. Las gráficas de estado se usan para identificar atributos de objetos y para refinar la descripción del comportamiento de un objeto, y los diagramas de secuencia se usan para identificar los objetos participantes y los servicios que proporcionan. Los diagramas de gráfica de estado también pueden usarse durante el diseño del sistema, y los objetos para describir los objetos del dominio de solución que tienen un comportamiento interesante. En los capítulos 5, *Análisis*, y 6, *Diseño del sistema*, describimos con más detalle el uso de los diagramas de gráfica de estado.

2.4.5 Diagramas de actividad

Las transiciones salientes son activadas por la terminación de una acción que está asociada con el estado. A esto se le llama **estado de acción**. Por convención, el nombre de un estado indica

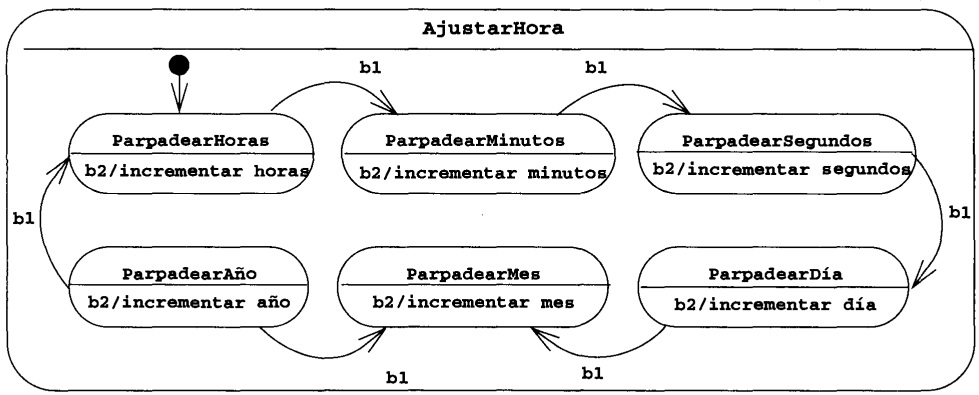


Figura 2-34 Gráfica de estado refinada asociada con el estado `AjustarHora` (diagrama de gráfica de estado UML).

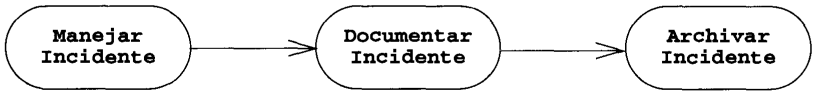


Figura 2-35 Un diagrama de actividad UML para `Incidente`. Durante el estado de acción `ManejarIncidente`, el `Despachador` recibe reportes y asigna recursos. Una vez que se cierra el `Incidente`, éste pasa a la actividad `DocumentarIncidente`, durante la cual todos los `OficialCampo` y `Despachador` participantes documentan el `Incidente`. Por último, la actividad `ArchivarIncidente` representa el archivado de la información relacionada con el `Incidente` en algún medio de acceso lento.

una condición, mientras que el nombre de un estado de acción indica una acción. Los **diagramas de actividad** son diagramas de gráfica de estado cuyos estados son estados de acción. La figura 2-35 es un diagrama de actividad que corresponde al diagrama de estado de la figura 2-31. Una visión alternativa y equivalente de los diagramas de actividad es interpretar los estados de acción como flujo de control entre actividades y transiciones, esto es, las flechas se interpretan como restricciones secuenciales entre actividades.

Las **decisiones** son ramas en el flujo de control. Indican transiciones alternas basadas en una condición del estado de un objeto o de un conjunto de objetos. Las decisiones se muestran con un rombo con una o más flechas entrantes y dos o más flechas salientes. Las flechas salientes están rotuladas con las condiciones que seleccionan una rama en el flujo de control. El conjunto de todas las transacciones salientes de una decisión representa el conjunto de todas las salidas posibles. En la figura 2-36, una decisión después de la acción `AbrirIncidente` selecciona entre tres ramas: si el incidente es de alta prioridad y es un incendio se le notifica al `JefeBomberos`. Si el incidente es de alta prioridad y no es un incendio se le notifica al `JefePolicía`. Por último, si no se satisface ninguna de estas condiciones, esto es, si el `Incidente` es de baja prioridad, no se notifica a ningún superior y se continúa con la asignación de recursos.

Las **transiciones complejas** son transiciones con varios estados de origen o varios estados de destino. Las transiciones complejas indican la sincronización de varias actividades (en el caso de varias fuentes) o la división del flujo de control en varios hilos (en el caso de destinos múltiples).

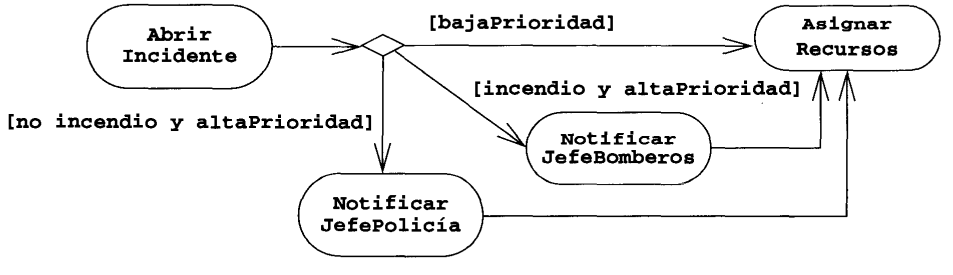


Figura 2-36 Ejemplo de decisiones en el proceso `AbrirIncidente`. Si el `Incidente` es un incendio y es de alta prioridad, el `Despachador` le notifica al `JefeBomberos`. Si el incidente es de alta prioridad y no es un incendio, se notifica al `JefePolicía`. En todos los casos, el `Despachador` asigna recursos para manejar el `Incidente`.

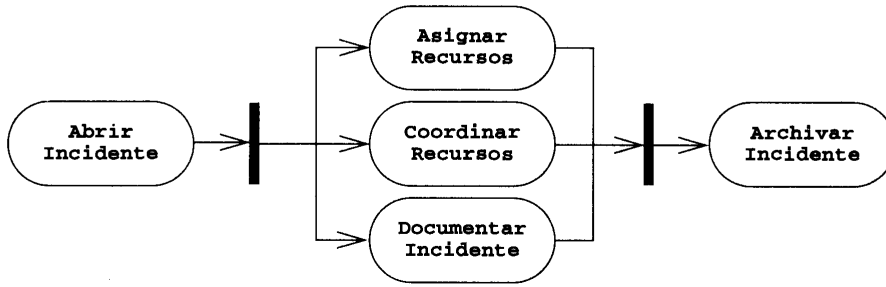


Figura 2-37 Un ejemplo de transiciones complejas en un diagrama de actividad UML.

Por ejemplo, en la figura 2-37 los estados de acción `AsignarRecursos`, `CoordinarRecursos` y `DocumentarIncidente` pueden suceder en paralelo. Sin embargo, sólo pueden iniciarse después de `AbrirIncidente`, y la acción `ArchivarIncidente` sólo puede iniciarse después de que hayan terminado todas las demás actividades.

Las acciones pueden agruparse en **carriles** para indicar el objeto o subsistema que instrumenta las acciones. Los carriles se representan como rectángulos que encierran a un grupo de acciones. Las transiciones pueden cruzar carriles. En la figura 2-38 el carril `Despachador` agrupa todas las actividades que realiza el objeto `Despachador`. El carril `OficialCampo` indica que el objeto `OficialCampo` es responsable de la acción `DocumentarIncidente`.

Aplicación de los diagramas de actividad

Los diagramas de actividad proporcionan una vista del comportamiento de un objeto centrada en la tarea. Pueden usarse, por ejemplo, para describir restricciones de secuencia entre casos de uso, actividades secuenciales entre un grupo de objetos o las tareas de un proyecto. En este libro usamos diagramas de actividad para describir las actividades del desarrollo de software en el capítulo 11, *Administración del proyecto*, y en el capítulo 12, *Ciclo de vida del software*.

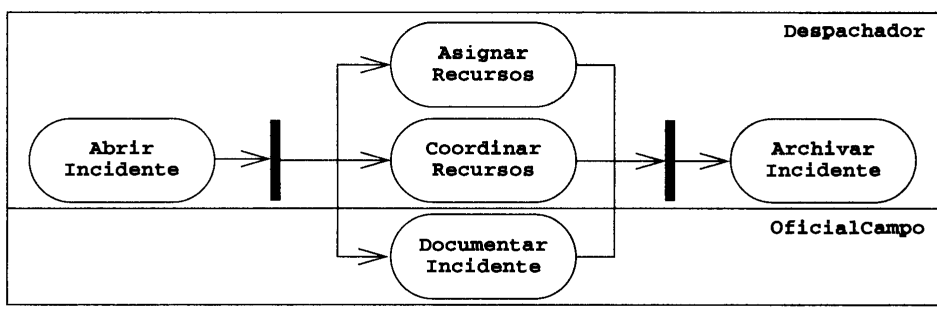


Figura 2-38 Un ejemplo de carriles en un diagrama de actividad UML.

2.4.6 Organización del diagrama

Los modelos de sistemas complejos se hacen complejos con rapidez conforme los refinan los desarrolladores. La complejidad de los modelos puede manejarse agrupando en **paquetes** los elementos relacionados. Un paquete es un agrupamiento de elementos del modelo, como casos de uso, clases o actividades, que definen alcances de la comprensión.

Por ejemplo, la figura 2-39 muestra casos de uso del sistema FRIEND agrupados por actor. Los paquetes se muestran como rectángulos con una ceja en su esquina superior izquierda. Los casos de uso que manejan la administración del incidente (por ejemplo, creación, asignación de recursos, documentación) están agrupados en el paquete `AdministraciónIncidente`. Los casos de uso que manejan el archivado del incidente (por ejemplo, archivado de un incidente, generación de reportes de incidentes archivados) están agrupados en el paquete `ArchivoIncidente`. Los casos de uso que manejan la administración del sistema (por ejemplo, adición de usuarios, registro de estaciones finales) están agrupados en el paquete `AdministraciónSistema`. Esto permite que el cliente y los desarrolladores organicen los casos de uso en grupos relacionados y que se enfoquen solamente en un conjunto limitado de casos de uso a la vez.

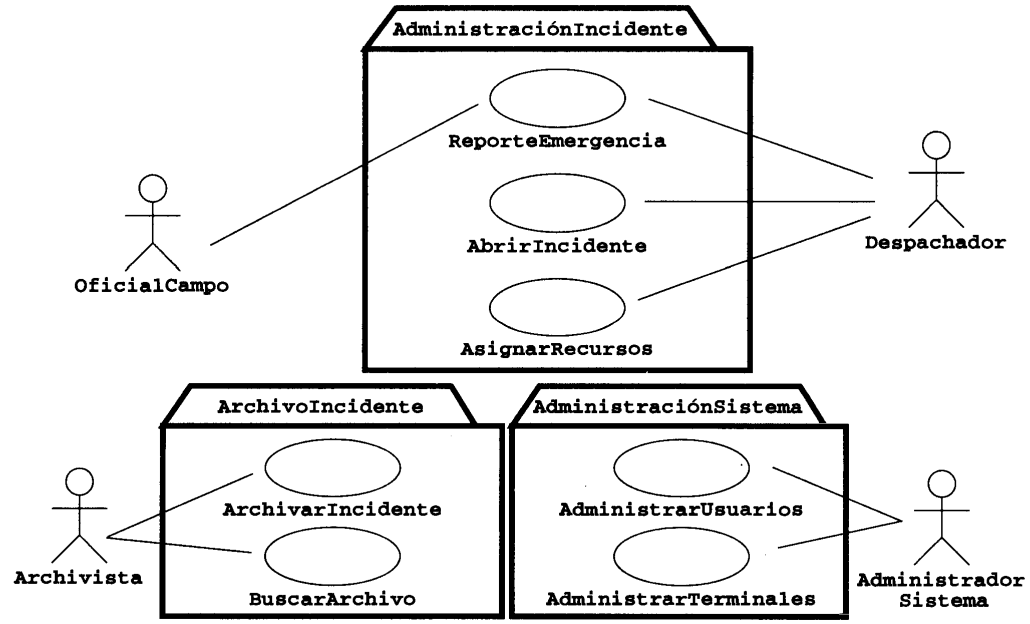


Figura 2-39 Ejemplo de paquetes: los casos de uso de FRIEND organizados por actor (diagrama de caso de uso UML).

Las figuras 2-39 y 2-40 son ejemplos de diagramas de clase que usan paquetes. Las clases del caso de uso `ReporteEmergencia` están organizadas de acuerdo con el sitio en donde se crean los objetos. `OficialCampo` y `ReporteEmergencia` son parte del paquete `EstaciónCampo`, y `Despachador` e `Incidente` son parte de `EstaciónDespachadora`. La figura 2-39 muestra los

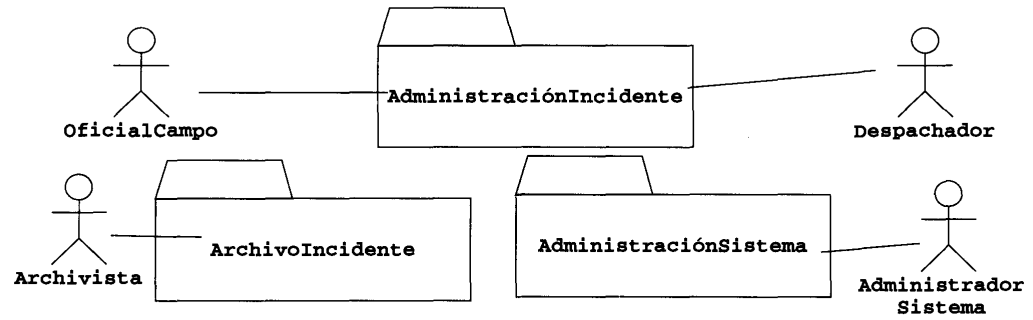


Figura 2-40 Ejemplo de paquetes: esta figura muestra los mismos paquetes que la figura 2-39, a excepción de que se suprimen los detalles de cada paquete (diagrama de caso de uso UML).

paquetes con los elementos del modelo que contienen, y la figura 2-40 muestra la misma información sin el contenido de cada paquete. La figura 2-40 es una imagen de nivel más alto del sistema y puede usarse para discutir asuntos en un nivel de sistema, mientras que la figura 2-39 es una vista más detallada que puede usarse para discutir el contenido de paquetes específicos.

Los paquetes (figura 2-41) se usan para manejar la complejidad, en la misma forma en que un usuario organiza los archivos y subdirectorios en directorios. Sin embargo, los paquetes no son necesariamente jerárquicos: la misma clase puede aparecer en más de un paquete. Para reducir las inconsistencias, las clases (y en términos más generales, los elementos del modelo) son poseídos exactamente por un paquete, mientras que se dice que los demás paquetes se refieren al elemento modelado. Observe que los paquetes son construcciones de organización y no objetos. No tienen comportamientos asociados y no pueden enviar o recibir mensajes.

Una **nota** es un comentario asociado a un diagrama. Los desarrolladores usan las notas para agregar información a los modelos y a los elementos de los modelos. Éste es un mecanismo ideal para el registro de asuntos pendientes relevantes para el modelo, la aclaración de un punto complejo, el registro de lo que hay que hacer o recordatorios. Aunque las notas no tienen semántica propia, se usan, a veces, para expresar restricciones que no pueden expresarse de otra manera en UML. La figura 2-42 proporciona un ejemplo de una nota.

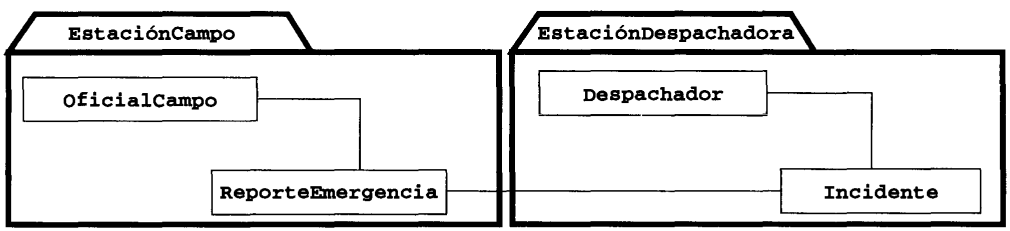


Figura 2-41 Ejemplo de paquetes. Las clases `OficialCampo` y `ReporteEmergencia` se encuentran en el paquete `EstaciónCampo`, y las clases `Despachador` e `Incidente` se encuentran en el paquete `Estación-Despachadora`.

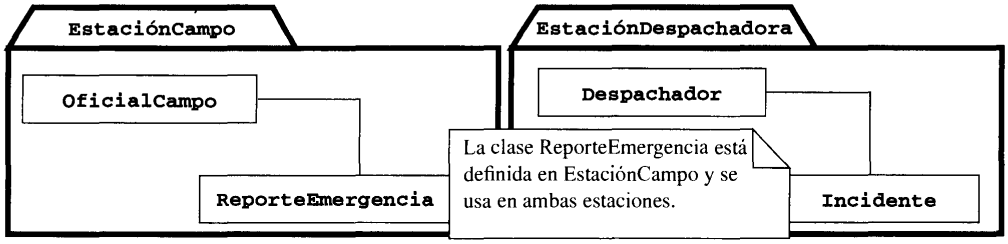


Figura 2-42 Un ejemplo de una nota. Se pueden añadir notas a un elemento específico de un diagrama.

2.4.7 Extensiones al diagrama

El objetivo de los diseñadores del UML fue proporcionar un conjunto de notaciones para modelar una amplia clase de sistemas de software. También reconocieron que un conjunto fijo de notaciones no podría lograr este objetivo, debido a que es imposible anticipar las necesidades que se encuentran en todos los dominios de aplicación y de solución. Por esta razón, UML proporciona varios mecanismos de extensión que permiten que el modelador extienda el lenguaje. En esta sección describimos dos de estos mecanismos, los **estereotipos** y las **restricciones**.

Un **estereotipo** es un texto encerrado entre paréntesis angulares (por ejemplo, `<<subsistema>>`) que se añade a un elemento UML, como una clase o una asociación. Esto permite que los modeladores creen nuevos tipos de bloques de construcción que necesitan en su dominio. Por ejemplo, durante el análisis clasificamos los objetos en tres tipos: entidad, frontera y control. El lenguaje UML básico sólo sabe acerca de objetos. Para introducir estos tres tipos adicionales usamos tres estereotipos, `<<entidad>>`, `<<frontera>>` y `<<control>>`, para representar el tipo de objeto (figura 2-43). Otro ejemplo es la relación entre casos de uso. Como vimos en la sección 2.4.1, las relaciones de inclusión en los diagramas de caso de uso están indicadas con una flecha de guiones y el estereotipo `<<incluye>>`. En este libro definiremos el significado de cada estereotipo conforme lo presentemos. Los estereotipos `<<entidad>>`, `<<frontera>>` y `<<control>>` se describen en el capítulo 5, *Análisis*.

Una **restricción** es una regla que se añade a un bloque de construcción UML. Esto nos permite representar fenómenos que no se pueden expresar de otra manera con UML. Por ejemplo, en la figura 2-44 un `Incidente` puede estar asociado con uno o más `ReporteEmergencia` del campo. Sin embargo, desde la perspectiva del `Despachador` es importante que se puedan ver los reportes en orden cronológico. Representamos el ordenamiento cronológico de `ReporteEmergencia`

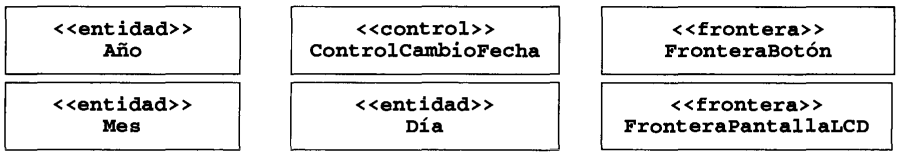


Figura 2-43 Ejemplos de estereotipos (diagrama de clase UML).

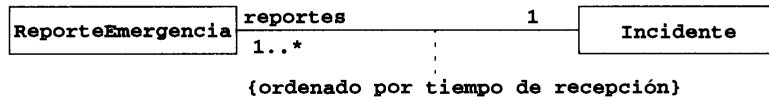


Figura 2-44 Un ejemplo de una restricción (diagrama de clase UML).

con los `Incidente` asociados con la restricción `{ordenado por tiempo de recepción}`. Las restricciones pueden expresarse como un texto informal o usando un lenguaje formal como el lenguaje de restricciones de objetos (OCL, por sus siglas en inglés [OMG, 1998]). En el capítulo 7, *Diseño de objetos*, describimos el OCL y el uso de restricciones.

2.5 Ejercicios

1. Trace un diagrama de caso de uso para un distribuidor de boletos de un sistema de trenes. El sistema incluye dos actores: un viajero que compra diferentes tipos de boletos y un sistema de computadora central que mantiene una base de datos de referencia para las tarifas. Los casos de uso deben incluir: `ComprarBoletoSencillo`, `ComprarTarjetaSemanal`, `ComprarTarjetaMensual` y `ActualizarTarifa`. También hay que incluir los siguientes casos excepcionales: `Retraso` (es decir, el viajero tarda demasiado para insertar el importe correcto), `TransacciónAbortada` (es decir, el viajero selecciona el botón de cancelar sin terminar la transacción), `DistribuidorSinCambio` y `DistribuidorSinPapel`.
2. Trace un diagrama de clase que represente un libro definido por la siguiente declaración: "un libro está compuesto por varias partes que a su vez están compuestas de varios capítulos. Los capítulos están compuestos de secciones". Enfóquese sólo en las clases y sus relaciones.
3. Trace un diagrama de objeto que represente la primera parte de este libro (es decir, parte I, *Comenzando*). Asegúrese que el diagrama de objeto que trace sea consistente con el diagrama de clase del ejercicio 2.
4. Amplíe el diagrama de clase del ejercicio 2 para que incluya los siguientes atributos:
   - Un libro incluye a un editor, fecha de publicación y un ISBN.
   - Una parte incluye un título y un número.
   - Un capítulo incluye un título, un número y un resumen.
   - Una sección incluye un título y un número.
5. Considere el diagrama de clase del ejercicio 4. Observe que las clases `Parte`, `Capítulo` y `Sección` incluyen un atributo de título y otro de número. Añada una clase abstracta y una relación de generalización para factorizar estos dos atributos en la clase abstracta.
6. Trace un diagrama de secuencia para el escenario `bodegaEnLlamas` de la figura 2-17. Incluya los objetos `roberto`, `alicia`, `juan`, `FRIEND` e instancias de otras clases que pueda necesitar. Trace sólo los cinco primeros mensajes enviados.
7. Trace un diagrama de secuencia para el caso de uso `ReporteEmergencia` de la figura 2-16. Trace sólo los cinco primeros mensajes enviados. Asegúrese de que sea consistente con el diagrama de secuencia del ejercicio 6.

8. Considere las actividades de desarrollo de software que describimos en la sección 1.4 del capítulo 1, *Introducción a la ingeniería de software*. Trace un diagrama de actividad que muestre estas actividades suponiendo que se ejecutan estrictamente en secuencia. Trace un segundo diagrama de actividad que muestre las mismas actividades sucediendo en forma incremental (es decir, una parte del sistema se analiza, diseña, implementa y prueba por completo antes de que se desarrolle la siguiente parte del sistema). Trace un tercer diagrama de actividad que muestre las mismas actividades sucediendo en forma concurrente.

Referencias

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* 2a. ed. Benjamin/Cummings, Redwood City, CA, 1994.

[Coad *et al.*, 1995] P. Coad, D. North y M. Mayfield, *Object Models: Strategies, Patterns, & Applications.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification.* Yourdon, Nueva York, 1978.

[FRIEND, 1994] *FRIEND Project Documentation.* School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1994.

[Harel, 1987] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 1987, págs. 231-274.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[Martin y Odell, 1992] J. Martin y J. J. Odell, *Object-Oriented Analysis and Design.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Mellor y Shlaer, 1998] S. Mellor y S. Shlaer, *Recursive Design Approach.* Prentice Hall, Upper Saddle River, NJ, 1998.

[Miller, 1956] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information", *Psychological Review*, vol. 63, 1956, págs. 81-97.

[OMG, 1998] Object Management Group, *OMG Unified Modeling Language Specification.* Framingham, MA, 1998. http://www.omg.org.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach.* Clarendon, Oxford, 1992.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Wirfs-Brock *et al.*, 1990] R. Wirfs-Brock, B. Wilkerson y L. Wiener, *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ, 1990.

3

3

Comunicación de proyectos

Dos tableros eléctricos para un cohete, fabricados por contratistas diferentes, se conectaban mediante un par de alambres. Gracias a una revisión exhaustiva anterior al vuelo se descubrió que los alambres estaban invertidos. Después de que se estrelló el cohete, la investigación reveló que los contratistas habían corregido los alambres invertidos como se les había instruido.

De hecho, ambos lo hicieron.

La ingeniería de software es una actividad de colaboración. El desarrollo de software reúne a participantes con diferentes conocimientos, como expertos en dominios, analistas, diseñadores, programadores, administradores, escritores técnicos, diseñadores gráficos y usuarios. Ningún participante puede comprender o controlar todos los aspectos del sistema que se está desarrollando y, por lo tanto, todos los participantes dependen de los demás para realizar su trabajo. Además, cualquier cambio al sistema o al dominio de aplicación requiere que los participantes actualicen su comprensión del sistema. Estas dependencias hacen que sea crítico compartir información en forma precisa y a tiempo.

La comunicación puede tomar muchas formas, dependiendo del tipo de actividad que está apoyando. Los participantes comunican su estado durante reuniones semanales o bimestrales y la registran en minutas de la reunión. Los participantes comunican el estado del proyecto al cliente durante las revisiones. La comunicación de los requerimientos y las alternativas de diseño es apoyada por los modelos y su documentación correspondiente. Las crisis y las incomprensiones se manejan mediante intercambios de información espontáneos, como llamadas telefónicas, mensajes de correo electrónico, conversaciones en los pasillos o en reuniones *ad hoc*.

En este capítulo, primero hacemos patente la necesidad e importancia de la comunicación en la ingeniería de software. Luego describimos diferentes aspectos de la comunicación de proyectos y su relación con diferentes tareas. Luego investigamos varias herramientas para apoyar la comunicación de proyectos. Por último, describimos un ejemplo de la infraestructura de comunicaciones.

3.1 Introducción: el ejemplo de un cohete

Cuando se desarrolla un sistema, los desarrolladores se enfocan en la construcción de un sistema que se comporte de acuerdo con las especificaciones. Cuando los desarrolladores interactúan con otros participantes en el proyecto, se enfocan en comunicar información con precisión y en forma eficiente. Aunque no parezca que la comunicación sea una actividad creativa o retadora, contribuye tanto al éxito del proyecto como un buen diseño o una implementación eficiente, como lo ilustra el siguiente ejemplo [Lions, 1996].

Ariane 501

El 4 de junio de 1996, a los 30 segundos del despegue, explotó el Ariane 501, el primer prototipo de la serie Ariane 5. La computadora de navegación principal tuvo un desbordamiento aritmético, se apagó y pasó el control a su gemela de reemplazo, como había sido diseñada. La computadora de respaldo, al haber tenido la misma excepción unas cuantas centésimas de segundos antes, ya se había apagado. El cohete, sin un sistema de navegación, dio una vuelta cerrada fatal para corregir una desviación que no había sucedido.

A un comité de investigación independiente le llevó menos de dos meses documentar la manera en que un error de software dio como resultado una falla total. El diseño del sistema de navegación del Ariane 5 fue uno de los pocos componentes del Ariane 4 que se reutilizó. Había sido probado en vuelo y no había fallado en el Ariane 4.

El sistema de navegación es responsable del cálculo de las correcciones de curso de una trayectoria especificada con base en la entrada recibida desde el sistema de referencia inercial. Un sistema de referencia inercial permite que un vehículo en movimiento (por ejemplo, un cohete) calcule su posición basado únicamente en datos de sensores de acelerómetros y giróscopos, esto es, sin referencia con el mundo externo. El sistema inercial debe ser inicializado con las coordenadas iniciales y alinear su eje con la orientación inicial del cohete. Los cálculos de alineación los realiza el sistema de navegación antes del lanzamiento, y es necesario actualizarlos en forma continua para tomar en cuenta la rotación de la Tierra. Los cálculos de alineación son complicados y se necesitan 45 minutos, aproximadamente, para realizarlos. Una vez que se lanza el cohete, los datos de alineación se transfieren al sistema de navegación de vuelo. Por diseño, los cálculos de alineación continúan durante otros 50 segundos después de la transferencia de datos al sistema de navegación. La decisión permite que se detenga la cuenta regresiva después que se realiza la transferencia de datos de alineación pero antes que se enciendan los motores sin tener que volver a iniciar los cálculos de alineación (esto es, sin tener que volver a iniciar un ciclo de cálculos de 45 minutos). Si el lanzamiento tiene éxito, el modo de alineación simplemente genera, durante 40 segundos, datos que no se usan después del despegue.

El sistema de computadora del Ariane 5 es diferente al del Ariane 4. Se duplicó la electrónica: dos sistemas de referencia inercial para calcular la posición del cohete, dos computadoras para comparar la trayectoria planeada contra la trayectoria actual y dos conjuntos de electrónica de control para hacer girar al cohete. Si fallara cualquier componente, se encargarían los sistemas de respaldo.

El sistema de alineación, diseñado sólo para cálculos en tierra, usa palabras de 16 bits para almacenar la velocidad horizontal (más que suficiente para los desplazamientos a causa del viento y la rotación de la Tierra). Después de 32 segundos de vuelo, la velocidad horizontal del Ariane 5 causó un desbordamiento, envió una excepción que fue manejada apagando la computadora a bordo y pasando el control al sistema de respaldo.

Discusión

No hubo una cobertura de pruebas adecuada para el software de alineación. Había estado sujeto a miles de pruebas, pero ninguna de ellas incluyó una trayectoria real. El sistema de navegación fue probado de manera individual. El equipo del sistema especificó pruebas y las ejecutaron los constructores del sistema de navegación. El equipo del sistema no se dio cuenta que el módulo de alineación podría causar que el procesador principal se apagara, en especial cuando no estaba en vuelo. Había sido una falla de comunicación entre el equipo del componente y el equipo del sistema.

En este capítulo tratamos la comunicación de proyectos dentro de un proyecto de desarrollo de software. Este tema no es específico de la ingeniería de software. Sin embargo, la comunicación es omnipresente a lo largo de un proyecto de desarrollo de software. El costo de una falla de comunicación puede tener un impacto alto y a veces fatal sobre el proyecto y la calidad del sistema entregado.

3.2 Una panorámica de la comunicación de proyectos

Las necesidades y herramientas de comunicación que la apoyan son varias y diversas. Para facilitar nuestra discusión, presentamos la siguiente clasificación y definiciones (figura 3-1):

- Un **modo de comunicación** se refiere a un tipo de intercambio de información que tiene objetivos y alcances definidos. Una revisión del cliente es un ejemplo de un modo de comunicación durante el cual el cliente revisa el contenido y la calidad de un proyecto disponible. El reporte de problemas es otro ejemplo del modo de comunicación durante el cual un usuario reporta un error a los desarrolladores. Un modo de comunicación puede ser **calendarizado**, si es un evento planeado, o **manejado por eventos**, si sucede en forma no determinística. Las revisiones del cliente son, por lo general, calendarizadas, mientras que los reportes de problemas son manejados por eventos.
- Un **mecanismo de comunicación** se refiere a una herramienta o procedimiento que puede usarse para transmitir y recibir información y apoyar un modo de comunicación. Las señales de humo y las máquinas de fax son mecanismos de comunicación. Los mecanismos de comunicación son **síncronos** si requieren que el emisor y los receptores estén disponibles al mismo tiempo. Si no es así, se dice que son **asíncronos**. Las señales de humo son síncronas, mientras que las máquinas de fax son asíncronas.

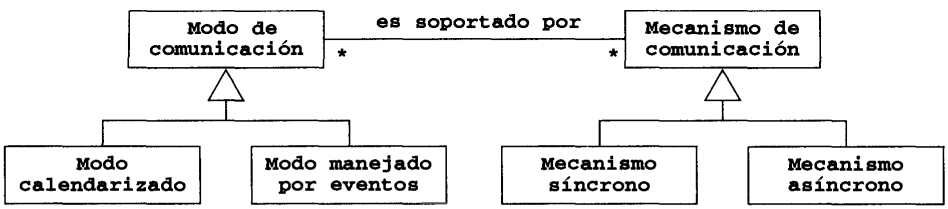


Figura 3-1 Clasificación de los modos y mecanismos de comunicación (diagrama de clase UML).

Los mecanismos de comunicación síncronos y asíncronos pueden usarse para apoyar un modo de comunicación calendarizado. Por ejemplo, en la figura 3-2 se pueden usar las señales de humo o una máquina de fax para una revisión del cliente. Por otro lado, sólo se pueden usar los mecanismos de comunicación asíncronos para apoyar los modos manejados por eventos (el reporte de un problema con señales de humo puede conducir a una pérdida de información si nadie estaba a cargo de observar el humo). Observe que un modo de comunicación puede estar apoyado por muchos mecanismos de comunicación diferentes: el documento de análisis de requerimientos puede ser enviado por fax al cliente, y el cliente puede enviar sus comentarios de regreso usando señales de humo. En

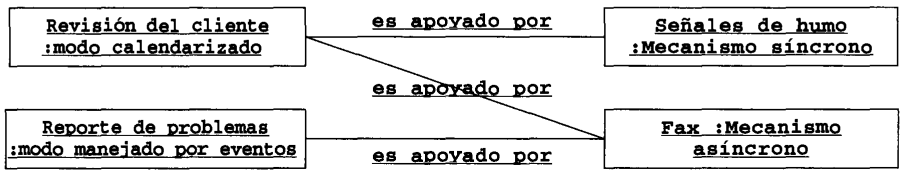


Figura 3-2 Ejemplos de modos y mecanismos (diagrama de clase UML). Tanto los modos calendarizados como los manejados por evento pueden estar apoyados por mecanismos asíncronos. Sin embargo, los modos manejados por eventos sólo pueden ser apoyados por mecanismos asíncronos.

forma similar, el mismo mecanismo puede apoyar muchos modos de comunicación: la máquina de fax puede recibir reportes de problemas o comentarios de una revisión del cliente.

En la sección 3.3 revisamos los diferentes modos de comunicación que pueden usarse en un proyecto. Los modos calendarizados que ahí se revisan incluyen revisiones del cliente, reportes de estado y versiones del sistema. Los modos manejados por eventos incluyen peticiones de cambios, aclaraciones y resolución de problemas. En la sección 3.4 investigamos varios mecanismos de comunicación que están disponibles para los participantes en el proyecto. En particular, nos enfocamos en reuniones, revisiones, formularios de reporte, cuestionarios, correo electrónico, grupos de noticias y groupware. En la sección 3.5 examinamos una combinación específica de modos y mecanismos de comunicación en un proyecto de ejemplo.

3.3 Modos de comunicación

La **comunicación calendarizada** sucede de acuerdo con un proceso periódico (tabla 3-1). Por ejemplo, un equipo revisa su estado cada semana, un cliente revisa el estado del proyecto cada mes. Estos son puntos de comunicación enfocados, bien organizados y, por lo general, frente a frente, dirigidos hacia el descubrimiento y resolución de un amplio rango de asuntos.

La **comunicación manejada por eventos** es causada por un evento específico (tabla 3-2), como el descubrimiento de una incompatibilidad entre dos subsistemas, la necesidad de una nueva característica o la necesidad de resolver diferentes interpretaciones antes de una revisión formal. Estos son puntos de comunicación espontáneos, *ad hoc* y posiblemente asíncronos dirigidos hacia la resolución rápida de un solo asunto.

Los proyectos de software requieren el apoyo para comunicación periódica y manejada por eventos. No todas las necesidades de comunicación pueden manejarse mediante eventos calendarizados y procesos formales, mientras que el intercambio de información que se realiza sólo cuando se necesita o se requiere da como resultado crisis adicionales debidas a fallas de comunicación y omisiones. En esta sección describimos ambos tipos de modos.

Tabla 3-1 Modos de comunicación calendarizados.

Modo	Objetivos	Alcance
Definición del problema (sección 3.3.1)	• Extraer del cliente y del usuario el conocimiento de los requerimientos • Extraer del cliente y del usuario el conocimiento del dominio	• Dominio de aplicación • Requerimientos funcionales y no funcionales • Calendario de entregas
Revisiones del cliente (sección 3.3.2)	• Asegurarse que el sistema que se está construyendo sea el que quiere el cliente • Revisar la factibilidad de los requerimientos no funcionales (por ejemplo, plataforma, desempeño) • Revisar el calendario de entregas	• Requerimientos funcionales y no funcionales • Calendario de entrega
Revisiones del proyecto (sección 3.3.3)	• Asegurarse que puedan comunicarse los subsistemas dependientes • Anticipar la terminación tardía de subsistemas críticos • Diseminar el conocimiento operacional entre equipos	• Diseño del sistema • Diseño de objetos • Pruebas
Inspecciones/ pruebas de recorrido (sección 3.3.4)	• Asegurar que la implementación de los subsistemas sea correcta • Mejorar la calidad de la implementación de los subsistemas • Diseminar el conocimiento operacional entre los participantes	• Implementación
Revisiones del estado (sección 3.3.5)	• Asegurar el apego al plan de tareas • Asegurar la terminación a tiempo de los productos a entregar • Diseminación de problemas potenciales	• Tareas • Conceptos de acción • Asuntos
Lluvia de ideas (sección 3.3.6)	• Generar y evaluar soluciones	• Un solo problema
Versiones (sección 3.3.7)	• Diseminación de gran cantidad de información (por ejemplo, documentos, código de subsistemas)	• Organización del proyecto • Plan de tareas • Sistema, documentos y modelos
Revisión post mortem (sección 3.3.8)	• Captura y organización de las lecciones aprendidas	• Organización del proyecto • Plan de tareas • Sistema y modelos

Tabla 3-2 Modos de comunicación manejados por eventos.

Modo	Objetivos	Alcance
Peticiones de aclaraciones (sección 3.3.9)	• Aclaraciones • Reporte de una ambigüedad en la documentación o en el código	• Cualquiera
Peticiones de cambios (sección 3.3.10)	• Reporte de un problema o una mejora posible	• Cualquiera
Resolución de problemas (sección 3.3.11)	• Alcanzar el consenso • Formalizar una solución seleccionada	• Un solo problema

3.3.1 Definición del problema

El objetivo de la **definición del problema** es que el gerente de proyecto y el cliente se pongan de acuerdo sobre el alcance del sistema que se va a construir. La actividad de definición del proyecto produce un documento de definición del problema que indica el dominio y la funcionalidad del sistema. También contiene requerimientos no funcionales, como la especificación de la plataforma o restricciones de velocidad. Considere el siguiente ejemplo [FRIEND, 1994].

Enunciado del problema FRIEND

Debilidades del manejo de emergencias actual. La mayoría de los sistemas para el manejo de emergencias de los gobiernos locales están establecidos para manejar cargas de llamadas normales, y se sobrecargan de inmediato cuando sucede un evento no usual o varios eventos. Durante una emergencia, quienes la atienden pueden adquirir información útil para quienes toman decisiones. Esta información no se transmite por radio, a menos que se solicite, debido al tiempo aire limitado.

Los servicios gubernamentales poseen gran cantidad de información que es inestimable durante las emergencias. Los departamentos de bomberos, la policía, los servicios médicos de emergencia, los trabajadores públicos y los inspectores de edificios recopilan gran cantidad de información basada en papel. Durante una emergencia, esta información basada en papel es en extremo difícil de usar, y rara vez se usa.

La información crucial no está siempre disponible de inmediato durante las emergencias. La información sobre el lugar del incidente, como materiales peligrosos, ubicación de tuberías de gas, etc., no se tiene disponible o no existe. Hay una necesidad de un sistema que agilice el tiempo de respuesta en las emergencias y proporcione información para tomar mejores decisiones.

Requerimientos funcionales. El objetivo del proyecto FRIEND es construir un sistema que maneje varios incidentes de emergencia a la vez bajo cargas de llamadas no usuales. El sistema debe estar preparado para un escenario de peor caso o una emergencia. El sistema debe permitir una respuesta rápida y remota entre oficiales de policía, bomberos, trabajadores de mantenimiento y el despachador del cuartel general. Se usará un ancho de banda amplio para que las necesidades de acuse de recibo antes de la transmisión no sean una limitación.

El sistema proporcionará uso instantáneo de información como:

- Información geográfica, como mapas de la ciudad que incluyan los servicios que hay sobre el terreno y subterráneos.
- Información sobre materiales peligrosos (Matpel).
- Información específica sobre edificios, como tuberías de gas y agua, ubicación de hidrantes para bomberos, planos de los pisos, etcétera.
- Recursos gubernamentales disponibles, como el plan de operación para emergencias (EOP, por sus siglas en inglés).

Siguiendo los lineamientos y el plan de operación para emergencias, el sistema notificará de manera automática al personal de los departamentos adecuados, creará listas de tareas a realizar y asignará recursos, así como cualquier otra tarea que podría ahorrar tiempo durante una emergencia.

Cada vehículo de puesto de comando que interactúe con el sistema FRIEND tendrá una computadora móvil que use comunicación inalámbrica para enviar reportes al despachador del cuartel general. El objetivo es reemplazar los mecanismos de entrada de los reportes de quienes primero atienden el caso con una interfaz de usuario de respuesta rápida y fácil captura de datos, ya sea mediante reconocimiento de voz, pantalla sensible al tacto o un sistema basado en pluma. Todas las transacciones del sistema deberán archivarse para un análisis futuro.

Requerimientos no funcionales. El sistema FRIEND será usado por primera vez por el departamento de policía de Bellevue como producto aislado. También se usará después en un grupo de municipios y tendrá la capacidad para proporcionar información local a los niveles superiores de gobierno. El hardware usado por el personal de servicio en campo estará expuesto a las condiciones climáticas y al trabajo rudo. FRIEND deberá ser transportable hacia el hardware existente que se tiene disponible en los gobiernos locales. Se espera que el prototipo sea escalable. Por lo tanto, el diseño del sistema deberá tomar en cuenta que pueda expandirse para manejar accidentes a niveles estatal y federal.

Criterios de aceptación. El cliente considera que esta definición del problema es muy amplia, y no espera que se proporcione toda la funcionalidad en la primera versión. Durante la fase de ingeniería de requerimientos del proyecto, el cliente negociará con los ingenieros de software la entrega de un prototipo aceptable. Después de la fase de negociación se congelarán los requerimientos específicos para la prueba de aceptación del cliente. El cliente espera recibir el producto negociado después de cuatro a seis semanas de la presentación al cliente. Como mínimo, el cliente espera que el prototipo que se entregue sea expansible en el futuro. Como prueba de aceptación mínima, el cliente espera que el prototipo negociado tenga una demostración exitosa en el sistema Andrew con un componente inalámbrico. Como prueba aceptable deseada, el cliente espera que el prototipo se comporte en forma satisfactoria en una prueba de campo en el departamento de policía de Bellevue.

La definición del problema no produce una especificación completa del enunciado. Tan sólo es una actividad de requerimientos preliminar que establece el terreno común entre el cliente y el proyecto. Trataremos las actividades de requerimientos en los capítulos 4, *Obtención de requerimientos*, y 5, *Análisis*.

3.3.2 Revisiones del cliente

El objetivo de las **revisiones del cliente** es que el cliente valore el avance del desarrollo y que los desarrolladores confirmen o cambien los requerimientos del sistema. La revisión del cliente puede usarse para manejar las expectativas de ambos lados y para incrementar la comprensión compartida entre los participantes. El enfoque de la revisión es sobre lo que hace el sistema y cuáles restricciones son relevantes para el cliente (por ejemplo, desempeño, plataforma). En la mayoría de los casos, la revisión no debe enfocarse en el diseño o la implementación del sistema, a menos que tenga un impacto en el cliente o el usuario. Las excepciones incluyen los contratos de software que imponen restricciones sobre el proceso de desarrollo, como los sistemas de software con requerimientos de seguridad.

Una revisión del cliente se realiza, por lo general, como una presentación formal durante la cual los desarrolladores se enfocan en alguna funcionalidad específica con el cliente. La revisión está precedida por la entrega de un producto de trabajo, como un documento de especificaciones, una maqueta de interfaz o un prototipo para evaluación. Al término de la revisión el cliente proporciona retroalimentación a los desarrolladores. Esta retroalimentación puede

Agenda para la prueba de aceptación del cliente OWL

Fecha: 12/05/1997.

Hora: 3- 4:30 p.m.

Ubicación: Forest Hall.

Objetivo: Revisión del sistema por parte del cliente e identificación de asuntos pendientes.

Panorama:

- Definición del problema.
- Objetivos del diseño.
- Arquitectura del sistema.
- Demostración 1: Interfaz y control del usuario remoto.
- Demostración 2: Editor en el sitio.
- Demostración 3: Visualización 3D e interfaz de usuario de voz.
- Preguntas y respuestas.
- Conclusiones de la revisión.

Figura 3-3 Un ejemplo de una agenda para una revisión del cliente.

consistir en una aprobación general o la solicitud de cambios detallados en la funcionalidad o la calendarización.
La figura 3-3 muestra un ejemplo de agenda para una revisión del cliente.

3.3.3 Revisiones del proyecto

El objetivo de una **revisión del proyecto** es que el gerente del proyecto valore el estado y que los equipos revisen las interfaces de los subsistemas. Las revisiones del proyecto también pueden favorecer el intercambio de conocimiento operacional entre los equipos, como problemas comunes encontrados en las herramientas o en el sistema. El enfoque de la revisión depende del producto que se está revisando. Para el diseño del sistema se revisan la descomposición y las interfaces de subsistemas de alto nivel. Para el diseño de objetos se revisan las interfaces de objetos. Para la integración y pruebas se revisan las pruebas y sus resultados. El enfoque de las revisiones del proyecto no debe ser sobre la funcionalidad, a menos que tenga un impacto en la factibilidad o el estado del sistema.

Una revisión del proyecto se realiza, por lo general, como una presentación formal durante la cual cada equipo presenta su subsistema ante el gerente o ante los equipos que dependen del subsistema. La revisión está precedida, por lo general, por la entrega de un documento (por ejemplo, el documento de diseño del sistema) que describe los aspectos del sistema que se van a revisar (por ejemplo, las interfaces de los subsistemas). Al término de la revisión, los desarrolladores pueden negociar cambios en las interfaces y en la calendarización.

3.3.4 Inspecciones y pruebas de recorrido

El objetivo de las **inspecciones** y **pruebas de recorrido** del código es incrementar la calidad de un subsistema mediante la revisión entre iguales. En el caso de una prueba de recorrido, un desarrollador presenta ante los demás miembros del equipo el código, línea por línea, que ha escrito. Los demás miembros del equipo cuestionan cualquier código sospechoso y tratan de descubrir la mayor cantidad posible de errores. El papel del desarrollador es facilitar la presentación y responder las preguntas del equipo. En el caso de una inspección, los miembros del equipo se enfocan en que el código se

apegue a una lista de criterios predefinidos (por ejemplo, ¿implementa el algoritmo especificado? ¿Usa en forma correcta las interfaces de los subsistemas dependientes?). En el caso de una inspección, el equipo conduce la discusión y el desarrollador responde preguntas. El punto central de la inspección o las pruebas de recorrido será el código, y no el programador o el diseño.

La comunicación entre participantes se basa en el código. El código actual se usa como marco de referencia común. Las inspecciones son similares a las revisiones del proyecto en su objetivo de incrementar la calidad y diseminar información operacional. Difieren con respecto a las revisiones en su formalidad, su audiencia limitada y su duración extendida. Las inspecciones y las pruebas de recorrido se usan en forma amplia y han sido efectivas para la detección temprana de defectos [Fagan, 1976], [Seaman y Basili, 1997]. Describimos las pruebas de recorrido en el capítulo 9, *Pruebas*.

3.3.5 Revisiones de estado

A diferencia de las revisiones del cliente y del proyecto que se enfocan en el sistema, las **revisiones de estado** se centran en las tareas. Las revisiones de estado se realizan principalmente en un equipo (por ejemplo, cada semana) y en ocasiones en un proyecto (por ejemplo, cada mes). El objetivo de las revisiones de estado es detectar desviaciones con respecto al plan de tareas y corregirlas. Las revisiones de estado también motivan a los desarrolladores para la terminación de tareas pendientes. La revisión del estado de las tareas motiva la discusión de asuntos pendientes y problemas no anticipados y, por lo tanto, motiva la comunicación informal entre los miembros del equipo. Con frecuencia se pueden compartir con más efectividad las soluciones a asuntos comunes, y el conocimiento operacional se disemina cuando se tratan dentro del alcance de un equipo (a diferencia de cuando son dentro del alcance del proyecto).

Las revisiones de estado representan una inversión en el poder del personal. El incremento de la efectividad de las revisiones tiene un impacto global en el desempeño del equipo. Las reuniones de estado deben tener una agenda disponible con anterioridad a la reunión en la que se describan las tareas y asuntos a revisar. Esto permite que se preparen los participantes en la reunión y cambien la agenda si es necesario que se trate un asunto urgente. Un participante designado para capturar la mayor cantidad de información posible (sobre todo de estado y decisiones) deberá levantar las minutas de cada reunión. Las minutas se deben poner a disposición de los participantes para que las revisen tan pronto como sea posible después de la reunión. Esto motiva a quien levanta la minuta para que la termine, y sirve para que los miembros del equipo que no pudieron asistir a la reunión se enteren del estado del equipo. Posteriormente se hace referencia a las minutas de la reunión cuando se tratan tareas relacionadas o se necesitan aclaraciones. Además, las minutas de las reuniones representan una parte de la historia del proyecto que puede analizarse después de que se termina el proyecto.

3.3.6 Lluvia de ideas

El objetivo de la **lluvia de ideas** es generar y evaluar una gran cantidad de soluciones para un problema. La lluvia de ideas se realiza, por lo general, en reuniones frente a frente, pero también puede realizarse por medio de correo electrónico o groupware. La idea fundamental de la lluvia de ideas es que las soluciones propuestas por un participante, aunque no sean válidas, pueden generar ideas y propuestas adicionales de los demás participantes. En forma similar, la evaluación de propuestas dentro de un grupo conduce a un criterio de evaluación más explícito y a una evaluación

más consistente. En situaciones en donde no existe una solución clara, el proceso de lluvia de ideas puede facilitarse mediante la separación de la generación y la evaluación. La lluvia de ideas también tiene el efecto lateral de lograr el consenso sobre las soluciones escogidas.

3.3.7 Lanzamientos

El objetivo de un **lanzamiento** es poner un documento o un subsistema a disposición de otros participantes del proyecto, reemplazando con frecuencia una versión anterior del artefacto. Un lanzamiento puede ser tan simple como un mensaje electrónico de dos renglones (vea la figura 3-4) o puede consistir en varios fragmentos de información: la nueva versión del artefacto, una lista de los cambios realizados desde el último lanzamiento del artefacto, una lista de problemas o asuntos que falta por resolver y un autor.

```
De: Al
Grupo de noticias: cs413.f96.arquitectura.discusión
Tema: SDD
Fecha: Jueves 25 de noviembre 03:39:12 - 0500
Líneas: 6
ID del mensaje: <3299B30.3507@andres.cmu.edu>
VersiónMime: 1.0
Tipo de contenido: Texto/llano; caracteres=us-ascii
Codificación para la transferencia de contenido: 7 bit

Aquí pueden encontrar una versión actualizada del documento API para el
subsistema de notificación: http://decaf/~al/FRIEND/notifapi.html.

--Al
Notificación al líder de grupo
```

Figura 3-4 Un ejemplo de un anuncio de un lanzamiento.

Los lanzamientos se usan para poner a disposición una gran cantidad de información en forma controlada, agrupando en lotes, documentando y revisando muchos cambios al mismo tiempo. Las revisiones de proyecto del cliente son precedidas, por lo general, por un lanzamiento de uno o más productos disponibles.

La administración de versiones de documentos, modelos y subsistemas se describe en el capítulo 10, *Administración de la configuración del software.*

3.3.8 Revisión post mortem

Las **revisiones post mortem** se enfocan en la extracción de lecciones del desarrollo una vez que se ha entregado el software. Las revisiones post mortem necesitan realizarse poco después de la conclusión del proyecto para que se pierda o distorsione muy poca información por experiencias subsiguientes. El fin del proyecto es, por lo general, un buen momento para valorar cuáles técnicas, métodos y herramientas han funcionado y han sido críticas para el éxito (o fracaso) del sistema.

Una revisión post mortem puede realizarse como una sesión de lluvia de ideas, un cuestionario estructurado seguido de entrevistas o reportes individuales escritos por los equipos (o los parti-

Preguntas acerca de problemas que sucedieron	¿Qué tipo de problemas de comunicación y negociación surgieron en el desarrollo del sistema?
Preguntas que ayudan a descubrir soluciones posibles para esos problemas	Especule sobre el tipo de estructura de información que se necesita para un diseño basado en equipo junto con una metodología de ingeniería de software orientada a objetos basada en modelos. ¿Cree que los foros que se proporcionaron (de discusión, de problemas, de documentos, de anuncios, etc.) resolvieron este reto? Identifique problemas dentro de la estructura de la información y proponga soluciones.
Preguntas que descubren otros aspectos del proyecto que se percibieron como positivos o podrían ser mejorados	¿Qué observaciones y comentarios tiene acerca del proyecto con relación a lo siguiente?: • Sus expectativas al inicio del proyecto y la manera en que evolucionaron. • Los objetivos de este proyecto. • La utilización de los casos de uso. • El ciclo de vida usado en el proyecto. • La administración del proyecto (reuniones, comunicación, etcétera). • El proceso de documentación.
Preguntas abiertas para atrapar todo	Además de las preguntas anteriores, siéntase libre para tratar cualquier otro asunto y proponga soluciones que crea que son relevantes.

Figura 3-5 Un ejemplo de preguntas para una revisión post mortem.

cipantes). En todos los casos, las áreas cubiertas deben incluir las herramientas, los métodos, la organización y los procedimientos usados en el proyecto. La figura 3-5 es un ejemplo de las preguntas que se pueden formular durante una revisión post mortem.

Aun si los resultados de las revisiones post mortem no se diseminan por toda la organización mediante canales formales (por ejemplo, reportes técnicos), todavía pueden diseminarse en forma indirecta mediante los participantes en el proyecto. Los participantes en el proyecto con frecuencia son reasignados a funciones o proyectos diferentes, y a menudo diseminan las lecciones aprendidas en el proyecto anterior a otras partes de la organización.

3.3.9 Peticiones de aclaraciones

Las **peticiones de aclaraciones** representan la mayor parte de la comunicación entre desarrolladores, clientes y usuarios. Las peticiones de aclaraciones son manejadas por eventos. Un participante puede solicitar aclaración acerca de cualquier aspecto del sistema que puede ser ambiguo. Las peticiones de aclaraciones pueden ocurrir durante reuniones informales, o por medio de llamadas telefónicas, correo electrónico, formularios o en la mayoría de los demás mecanismos de comunicación disponibles para el proyecto. Las situaciones en las que la mayoría de las necesidades de información se manejan de este modo, son síntomas de una infraestructura de comunicación defectuosa. Tales proyectos a menudo enfrentan serias fallas en su desarrollo a causa de malos entendidos e información faltante o colocada en lugares erróneos. La figura 3-6 muestra un ejemplo de una petición de aclaración.

```
De: Alicia
Grupo de noticias: CS413.arquitectura.discusión
Tema: SDD
Fecha: jueves 10 de octubre 23:12:48 - 0400
ID del mensaje: <325DBB30.4380@andres.cmu.edu>
VersiónMime: 1.0
Tipo de contenido: texto/llano; caracteres=us-ascii

¿Exactamente cuándo quieres el documento del diseño del sistema? Hay algunas
confusiones sobre la fecha de entrega real: el calendario dice que es el 22 de
octubre y la plantilla dice que será hasta el 7 de noviembre.
Gracias,
     Alicia
```

Figura 3-6 Un ejemplo de una petición de aclaración.

3.3.10 Petición de cambio

Una **petición de cambio** es un modo de comunicación manejado por evento. Un participante reporta un problema y, en algunos casos, propone soluciones. El participante reporta un problema con el sistema mismo, su documentación, el proceso de desarrollo o la organización del proyecto. Las peticiones de cambio con frecuencia se formalizan cuando la cantidad de participantes y el tamaño del sistema son considerables. Las peticiones de cambio contienen una clasificación (por ejemplo, defecto severo, petición de características, comentario), una descripción del problema, una descripción del contexto en donde sucede y cualquier cantidad de material de apoyo. Los formularios de petición de cambio han sido popularizados por el software de seguimiento de defectos. Pueden aplicarse a otros aspectos del proyecto (por ejemplo, plan de tareas, proceso de desarrollo, procedimientos de pruebas). La figura 3-7 muestra un ejemplo de un formulario de petición de cambio.

3.3.11 Resolución de problemas

Una vez que se han reportado los problemas y se han propuesto y evaluado soluciones, es necesario que se seleccione, comunique y ponga en práctica una solución única. Una organización plana puede seleccionar una solución como parte de un proceso de lluvia de ideas. Una organización jerárquica, o una situación de crisis, puede requerir que un solo individuo seleccione o imponga una solución. En todos los casos, la solución necesita documentarse y comunicarse a los participantes relevantes. La documentación de la resolución permite que los participantes hagan referencia a la decisión más adelante en el proyecto, en caso de malos entendidos. La comunicación efectiva de la decisión permite que los participantes permanezcan sincronizados.

Una base de problemas puede servir como mecanismo de comunicación para apoyar el seguimiento del problema, la lluvia de ideas y la **solución del problema.** La base de problemas que se muestra en la figura 3-8 despliega una lista de mensajes intercambiados a consecuencia de soluciones de problemas. Los títulos de los mensajes precedidos por I: indican problemas, los que están precedidos por P: (de propuestas) sugieren soluciones, A+ y A– indican argumentos en apoyo y en contra de una solución. Por último, una vez que se resuelve el problema, se envía un solo mensaje, llamado solución, para documentar la decisión que se tomó sobre ese asunto. En el capítulo 8, *Administración de la fundamentación*, se describen bases de problemas y modelado de problemas.

Información de encabezado para identificar el cambio	Número de reporte: 1291 Fecha: 5/3 Autor: David Resumen: el cliente FRIEND falla cuando se envían formularios vacíos.
Información del contexto para la localización del problema	Subsistema: interfaz de usuario Versión: 3.4.1 Clasificación: • Funcionalidad faltante o incorrecta • Violación de convenciones • **Error** • Error de documentación Severidad: • **Severo** • Moderado • Molesto
Descripción del problema y razones para el cambio	Descripción: Razones
Descripción del cambio deseado	Solución propuesta:

Figura 3-7 Un ejemplo de un formulario para petición de cambios.

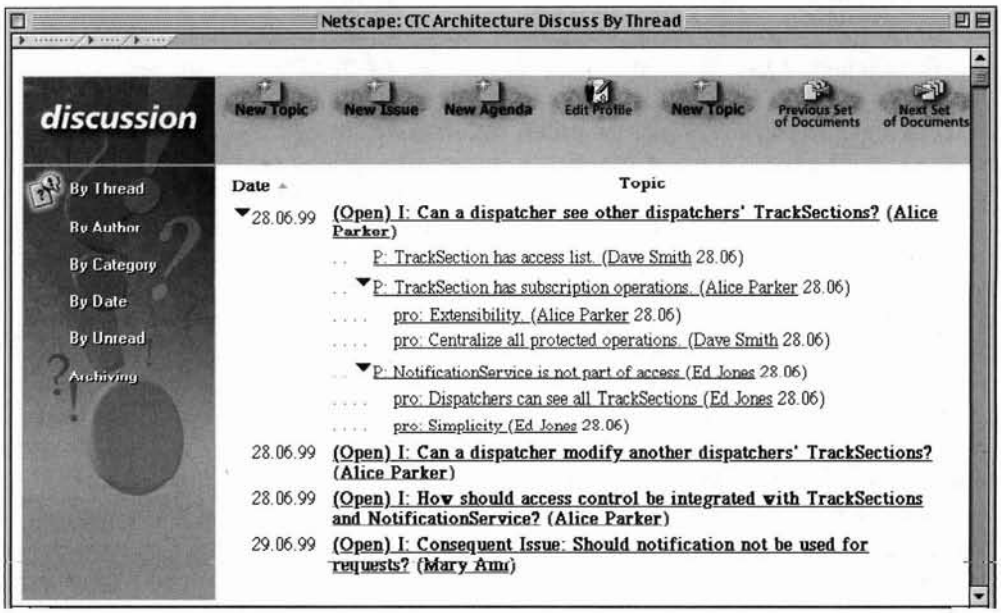


Figura 3-8 Un ejemplo de una base de problemas (base de datos Domino Lotus Notes).

3.3.12 Discusión

En esta sección presentamos un amplio rango de modos de comunicación. No todos los proyectos ponen el mismo énfasis en todos los modos de comunicación. Puede ser que los proyectos que producen software para entregarlo tal como es no tengan revisiones del cliente, pero pueden tener procedimientos de reporte de cambios sofisticados. Otros proyectos ponen más énfasis en las pruebas que en las inspecciones de código y pruebas de recorrido.

Los modos de comunicación que se usan en un proyecto están dictados, por lo general, por el dominio de aplicación del sistema y las actividades del proyecto. Con frecuencia estos modos se hacen explícitos y se formalizan en procedimientos para atacar rutas de comunicación críticas. Por ejemplo, las revisiones del proyecto facilitan que todos los participantes sincronicen su estado y aborden los problemas que se refieren a más de uno o dos subsistemas. Las revisiones del proyecto llegan a ser críticas en especial cuando el tamaño del proyecto se incrementa a tal punto que pocos participantes tienen una vista global del sistema.

3.4 Mecanismos de comunicación

En esta sección describimos varios mecanismos de comunicación que apoyan los intercambios de información. Clasificamos a los mecanismos de comunicación en dos categorías amplias. Los **mecanismos síncronos**, como el teléfono o la videoconferencia, requieren que el emisor y el receptor estén disponibles al mismo tiempo, mientras que no sucede así con los **mecanismos asíncronos**, como el correo electrónico o el fax. Clasificamos además a los mecanismos síncronos como mecanismos en el mismo lugar y en diferentes lugares. Los mecanismos en el mismo lugar (por ejemplo, conversaciones en los pasillos) requieren que el emisor y el receptor se encuentren en el mismo lugar (por ejemplo, el pasillo), mientras que no sucede así con los mecanismos en lugares diferentes.

Las reuniones personales informales, las entrevistas y las reuniones son mecanismos síncronos en el mismo lugar. Estos mecanismos se caracterizan por un bajo contenido de información técnica. Sin embargo, los mecanismos en un solo lugar permiten la transferencia de información no verbal que es crítica durante la generación de consenso y la negociación. Además, los mecanismos en un solo lugar permiten las aclaraciones y, en términos más generales, tienen un mayor grado de flexibilidad. El costo de los mecanismos síncronos es alto, en especial cuando están involucrados muchos participantes. El groupware síncrono en diferentes lugares trata de reducir los costos de la comunicación síncrona al no requerir que todos los participantes estén en una misma ubicación. Además, el groupware promete poner a disposición una variedad de información (por ejemplo, código ejecutable, documentos editables, estructuras de hipervínculo) incrementando, por lo tanto, el ancho de banda entre los participantes. Los mecanismos de comunicación síncronos se listan en la tabla 3-3 y se describen en las siguientes secciones.

Los cuestionarios, el fax, el correo electrónico, los grupos de noticias y el groupware de momentos diferentes son mecanismos de comunicación asíncronos. Permiten que se comuniquen con rapidez gran cantidad de detalles. Además, permiten que haya una mayor cantidad de receptores, y esto hace que estos mecanismos sean ideales para el lanzamiento de documentos y código. La comunicación asíncrona tiene la desventaja de que es inaccesible la información no verbal y, por lo tanto, incrementa el potencial para la falta de comprensión. Los mecanismos de comunicación asíncronos se describen en la tabla 3-4 y en las secciones 3.4.6 a 3.4.9.

Tabla 3-3 Mecanismos de comunicación síncronos.

Mecanismo	Modos soportados
Conversaciones en pasillos (sección 3.4.1)	Petición de aclaraciones Petición de cambios
Cuestionarios y entrevistas estructurados (sección 3.4.2)	Requerimientos de conocimientos sobre el dominio de aplicación Revisión post mortem
Reuniones (frente a frente, telefónicas, en vídeo) (sección 3.4.3)	Revisiones de cliente Revisión del proyecto Inspección Revisión de estado Revisión post mortem Lluvia de ideas Resolución de problemas
Groupware en diferente lugar y al mismo tiempo (Sección 3.4.5)	Revisión del cliente Revisión del proyecto Inspección Lluvia de ideas Resolución de problemas

Tabla 3-4 Mecanismos de comunicación asíncronos.

Mecanismo	Modos soportados
Correo electrónico (sección 3.4.6)	Lanzamiento Petición de cambio Lluvia de ideas
Grupos de noticias (sección 3.4.7)	Lanzamiento Petición de cambio Lluvia de ideas
World Wide Web (sección 3.4.8)	Lanzamiento Inspecciones de código asíncronas Petición de cambio Lluvia de ideas
Lotus Notes (sección 3.4.9)	Lanzamiento Petición de cambio Lluvia de ideas

3.4.1 Conversaciones en los pasillos

Las **conversaciones en los pasillos** son intercambios de información informales manejados por eventos basados en la oportunidad. Dos participantes se reúnen por accidente y aprovechan la situación para intercambiar información.

Ejemplo. Dos participantes en el proyecto, Sally y Bob, se reúnen junto a la cafetera. Sally, que es miembro del equipo de interfaz de usuario, se acuerda que Bob es miembro del equipo de notificación, el cual es responsable de la comunicación entre los subsistemas cliente y el servidor. Toda la mañana Sally ha estado experimentando fallas aleatorias cuando recibe paquetes del servidor. Ella no está segura si el problema se debe al servidor, al subsistema de comunicación o a su código. Bob le responde que no se había dado cuenta de que se estaba usando el servidor en ese momento y que ha estado probando una nueva revisión del sistema de comunicaciones, y eso explica el comportamiento que ha observado Sally. Bob ha dejado a un lado la política de administración de la configuración para ahorrar tiempo

Las conversaciones en los pasillos representan una parte sustancial de la comunicación del proyecto. Son baratas y efectivas para la resolución de problemas simples causados por falta de coordinación entre miembros del proyecto. Además, también son efectivas para apoyar el intercambio de conocimiento operacional, como las frecuentes acerca de herramientas, procedimientos o la ubicación de la información del proyecto. Las desventajas de las conversaciones en los pasillos incluyen su pequeña audiencia y la falta de historia: se puede perder información importante y pueden producirse malentendidos cuando el contexto de la conversación se apoya en otros participantes. Además, no se puede tener acceso a ningún documento, base de datos o mensaje electrónico cuando se hace referencia a una decisión que se tomó y comunicó durante una conversación en el pasillo.

3.4.2 Cuestionarios y entrevistas estructuradas

El objetivo de un **cuestionario** es obtener información de una o más personas en forma estructurada. Los cuestionarios se usan, por lo general, para obtener conocimiento del dominio de los usuarios y expertos, comprender los requerimientos del usuario y las prioridades. También pueden usarse para extraer lecciones aprendidas durante una revisión post mortem. Los cuestionarios pueden incluir preguntas de opción múltiple y preguntas abiertas.

Los cuestionarios tienen la ventaja de obtener información confiable con un costo mínimo por parte del usuario. Los cuestionarios pueden ser respondidos por los usuarios en forma independiente y luego revisados y analizados por el analista o desarrollador. La aclaración de respuestas ambiguas o incompletas puede obtenerse después durante una **entrevista estructurada**. La desventaja de los cuestionarios es que son difíciles de diseñar. Sin embargo, el costo de los errores de los requerimientos y la falta de comprensión entre el cliente y el desarrollador con frecuencia justifican su costo. Más adelante se recopila información suficiente acerca del dominio y se escribe un documento de análisis de requerimientos para que la mayoría de las revisiones del sistema y temas adicionales se traten en las revisiones de los clientes.

3.4.3 Reuniones

Las **reuniones**[1] frente a frente permiten que varios participantes revisen y negocien asuntos y soluciones. En la actualidad, las reuniones son el único mecanismo que permite una resolución efectiva de problemas y la generación de consenso. La desventaja de las reuniones es su costo en recursos y la dificultad para administrarlas. Para incrementar la transferencia de información y la cantidad de decisiones tomadas durante una reunión, se asignan papeles a participantes seleccionados:

- El **moderador principal** es responsable de organizar la reunión y guiar su desarrollo. El moderador lleva una agenda que describe el objetivo y el alcance de la reunión. La agenda se entrega, por lo general, antes de la reunión para que la revisen los participantes. Esto permite que los participantes decidan si la reunión es importante para ellos y permite la preparación de material de apoyo para las reuniones.
- El **secretario de actas** es responsable de registrar la reunión, y puede tomar notas por escrito o en una computadora laptop, organizarlas al terminar la reunión y repartirlas poco tiempo después de la reunión para que las revisen los participantes en ella. Esto permite que los participantes aprueben el resultado de la reunión. El registro escrito de la reunión facilita que los participantes compartan información con los miembros que no estuvieron presentes.
- El **tomador de tiempo** es responsable de llevar cuenta del tiempo y notificar al moderador si una discusión consume más tiempo que el que le fue asignado en la agenda.

Una **agenda de reunión** consta, al menos, de tres secciones: un encabezado que identifica la ubicación, el tiempo y los participantes de la reunión planeada; una lista de asuntos que reportarán los participantes y una lista de temas que necesitan ser tratados y resueltos en la reunión. A cada intercambio de información y concepto en discusión también se le asigna un tiempo que permite que el tomador de tiempo se asegure que la reunión termine a tiempo. La figura 3-9 es un ejemplo de una agenda de reunión.

Un conjunto de **minutas de reuniones** consta de tres secciones que corresponden con las secciones de la agenda. Además, las minutas de reunión incluyen una sección que describe los asuntos de acción que son resultado de la reunión (es decir, asuntos que describen las acciones a tomar por parte de los participantes en la reunión a consecuencia de ésta). La sección de encabezado contiene la ubicación, el tiempo y los participantes de la reunión actual. La sección de intercambio de información de asuntos contiene la información que se compartió durante la reunión. La sección de decisiones de asuntos contiene un registro de las decisiones que se tomaron (y las que no se tomaron). La figura 3-10 es un ejemplo de una minuta de reunión.

Aunque las reuniones que se realizan en un solo lugar son las más eficientes, es posible realizar reuniones cuando los participantes están distribuidos geográficamente usando teleconferencias o videoconferencias. Esto reduce el costo a expensas de un ancho de banda menor y una menor confiabilidad. Una agenda bien estructurada que se tenga disponible antes de la reunión llega a ser crucial, ya que con una calidad visual y de audio menor el control de la reunión llega

1. Los procedimientos de reunión descritos en esta sección se derivan de [Kayser, 1990].

Información de encabezado que identifica la reunión y la audiencia	**Cuándo y dónde** **Fecha:** 30/01 **Inicio:** 4:30 p.m. **Fin:** 5:30 p.m. **Edificio:** Wean Hall **Salón:** 3420	**Papel** **Moderador principal:** Pedro **Tomador de tiempo:** David **Secretario de actas:** Eduardo
Resultado deseado de la reunión	**1. Objetivo** Resolver cualquier requerimiento que nos impida iniciar la elaboración del prototipo	
Acciones a reportar	**2. Estado [Tiempo asignado: 15 minutos]** David: Estado del código de análisis sintáctico de comandos	
Asuntos programados para su discusión (y resolución) durante la reunión	**3. Asuntos a discusión [Tiempo asignado: 35 minutos]** 3.1 ¿Cómo manejar los datos de entrada formateados de manera arbitraria? 3.2 ¿Cómo manejar los datos de salida? 3.3 Código para el análisis sintáctico de comandos (modificabilidad, compatibilidad retrospectiva)	
El periodo de cierre es el mismo para todas las reuniones	**4. Cierre [Tiempo asignado: 5 minutos]** 4.1 Revisión y asignación de nuevas acciones 4.2 Crítica de la reunión	

Figura 3-9 Un ejemplo de una agenda de reunión.

a ser difícil. Además, el conocimiento de las voces individuales y sus particularidades mejora la comunicación entre los participantes.

Cuando se escribe una agenda de reunión, el moderador debe ser lo más concreto posible sin aumentar la longitud de la agenda. Por lo general, es tentador elaborar una plantilla de agenda genérica y reutilizarla de manera sistemática sin modificaciones. Esto tiene la desventaja de quitar la sustancia del proceso de reunión convirtiéndolo en un proceso burocrático. La figura 3.11 es un ejemplo de una agenda sin contenido. Con la sola modificación del encabezado, esta agenda se podría aplicar a la mayoría de las reuniones de subsistemas y, por lo tanto, no tiene ninguna información nueva que no sepan ya los participantes.

3.4.4 Revisiones

Las **revisiones** son reuniones formales durante las cuales una parte externa evalúa la calidad de un artefacto (por ejemplo, documento o código). Por lo general, se realiza como una presentación formal precedida por un lanzamiento y seguida por varias peticiones de cambio. Las revisiones son más costosas que las reuniones normales, ya que a menudo requieren viajes del cliente o los desarrolladores. Como sucede con las reuniones normales, se elabora una agenda y se envía a la parte externa antes de la revisión. Las minutas de la revisión se registran en forma meticulosa e incluyen peticiones de cambio y comentarios de otros asuntos que pueda tener la parte exterior.

Información de encabezado que identifica la reunión y la audiencia	**Cuándo y dónde** **Fecha:** 30/01 **Inicio:** 4:30 p.m. **Fin:** 6:00 p.m. **Edificio:** Wean Hall **Salón:** 3420	**Papel** **Moderador principal:** Pedro **Tomador de tiempo:** David **Secretario de actas:** Eduardo **Participantes:** Eduardo, David, María, Pedro, Alicia
Texto de la agenda	**1. Objetivo** ...	
Resumen de la información intercambiada	**2. Estado** ...	
Relación de los temas tratados y resoluciones	**3. Discusión** 3.1 El código para la revisión sintáctica es una instrucción if de 1200-1300 líneas. Esto hace que sea muy difícil añadir nuevos comandos o modificar los existentes sin romper la compatibilidad retrospectiva con los clientes existentes. Propuestas: 1) Reestructurar el código de revisión sintáctica asignando un objeto para cada tipo de comando. 2) Pasar todos los argumentos del comando por nombre. Esto último facilitaría que se mantuviera la compatibilidad retrospectiva. Por otro lado, esto incrementaría el tamaño de los comandos, incrementando por lo tanto el tamaño del archivo de comandos. Resolución: Reestructurar el código por ahora. Volver a ver este asunto si la compatibilidad retrospectiva realmente resulta un problema (de cualquier manera, el código de llamadas necesitaría volverse a escribir). Vea AI[1]. ... *Por brevedad se omite la discusión de los demás temas.*	
Adiciones y modificaciones al plan de tareas	**4. Cierre** AI[1] Para: David. Volver a ver el código de revisión sintáctica de comandos. Poner énfasis en la modularidad. Coordinarse con Guillermo del grupo de bases de datos (quien podría asumir la compatibilidad retrospectiva). ... *Se omiten otras acciones y la crítica de la reunión por brevedad.* ...	

Figura 3-10 Un ejemplo de minuta de reunión.

El alcance de una revisión (por ejemplo, una revisión de cliente) puede requerir que la revisión se organice mucho antes de su ejecución. Sin embargo, debe haber flexibilidad en la organización, ya que el objeto de la revisión (por ejemplo, el documento de requerimientos) puede evolucionar hasta el último momento (por ejemplo, unos cuantos días u horas antes de la revisión). Esta flexibilidad puede permitir que los desarrolladores descubran más asuntos, puntos de aclaración o soluciones alternas, y minimicen el tiempo durante el que necesite estar congelado el estado de lo que se va a entregar.

Información de encabezado que identifica la reunión y la audiencia	**Cuándo y dónde** **Fecha:** 30/01 **Inicio:** 4:30 p.m. **Fin:** 5:30 p.m. **Edificio:** Wean Hall **Salón:** 3420	**Papel** **Moderador principal:** Pedro **Tomador de tiempo:** David **Secretario de actas:** Eduardo
Resultado deseado de la reunión	**1. Objetivo** Resolver los asuntos pendientes	
Acciones a reportar	**2. Estado [Tiempo asignado: 15 minutos]** David: Acciones de David	
Asuntos programados para su discusión (y resolución) durante la reunión	**3. Asuntos a discusión [Tiempo asignado: 35 minutos]** 3.1 Asuntos de requerimientos 3.2 Asuntos de diseño 3.3 Asuntos de implementación	
El periodo de cierre es el mismo para todas las reuniones	**4. Cierre [Tiempo asignado: 5 minutos]** 4.1 Revisión y asignación de nuevas acciones 4.2 Crítica de la reunión	

Figura 3-11 Un ejemplo de una agenda de reunión deficiente.

3.4.5 Groupware en diferentes lugares y al mismo tiempo

El **groupware en diferentes lugares y al mismo tiempo** es una herramienta que permite que usuarios que se encuentran distribuidos colaboren en forma síncrona. Aunque durante mucho tiempo estas herramientas sólo estuvieron disponibles en el campo de la investigación [Grudin, 1988], se han estado moviendo poco a poco hacia el mundo comercial con la popularidad reciente de los salones de conversación y foros basados en Internet. Herramientas como Teamwave [Teamwave, 1997] o NetMeeting permiten que un grupo de participantes colabore en forma síncrona en un espacio de trabajo compartido. Proporcionan una metáfora de reunión: los usuarios "entran" a un salón de conversación que les permite ver un gráfico o un texto a considerar. Todos los usuarios ven el mismo estado. Por lo general, sólo uno puede modificarlo en un momento dado. El control puede ser anárquico (cualquiera puede tomar el control de la voz) o secuencial (quien tiene la voz se la pasa al siguiente usuario).

Una debilidad del groupware al mismo tiempo es la dificultad para coordinar a los usuarios. El tecleo se lleva más tiempo del que los usuarios están preparados a invertir. Las palabras escritas necesitan escogerse con más cuidado, tomando en cuenta que se pierde la información no verbal. Además, ligeras fallas en la conexión de red pueden representar suficiente interferencia para que se pierda la coordinación de usuarios.

Sin embargo, el groupware al mismo tiempo puede llegar a ser un sustituto útil para las videoconferencias cuando se combina con un canal de audio: los usuarios pueden ver el mismo documento y discutirlo normalmente mientras se desplazan y apuntan a áreas específicas del documento. En todos los casos, la colaboración en diferentes lugares todavía es un ejercicio no trivial que necesita tener un guión y estar planeado por anticipado. El desarrollo cooperativo de procedimientos

para el apoyo de la colaboración es una tarea retadora cuando no se dispone de la proximidad ni de la comunicación no verbal.

3.4.6 Correo electrónico

El **correo electrónico** permite que un emisor transmita un mensaje de texto arbitrario a uno o más receptores. La transmisión real del mensaje se lleva desde unos cuantos segundos hasta unas cuantas horas o días. El mensaje se recibe en un buzón y se mantiene ahí hasta que el receptor lo lee y lo archiva. El correo electrónico en un proyecto de desarrollo con frecuencia se usa en vez de memorándum de oficina o llamadas telefónicas. Con este mecanismo los participantes pueden intercambiar un amplio rango de artefactos, desde noticias informales cortas hasta documentos grandes. Por último, el correo electrónico está estandarizado por varias peticiones de comentarios (RFCs, por sus siglas en inglés) de Internet que están implementadas y se usan en forma amplia. En consecuencia, el correo electrónico se usa a través de diferentes organizaciones y plataformas.

Debido a su naturaleza asíncrona y latencia corta, el correo electrónico es ideal para el apoyo de modos de comunicación manejados por evento, como las peticiones de aclaraciones, peticiones de cambio y lluvia de ideas. El correo electrónico también se usa a menudo para el anuncio de lanzamientos y el intercambio de versiones intermedias o documentos. A diferencia del teléfono, el correo electrónico permite que los participantes lo reciban cuando no se encuentran disponibles sin introducir restricciones en el emisor.

Una desventaja del correo electrónico es que la comunicación puede percibirse como una tarea trivial. El tecleo de un mensaje de una línea y su envío se lleva unos cuantos segundos. Pero el mismo mensaje de una línea puede tomarse fuera de contexto y entenderse mal, enviarse a la persona errónea o, con mayor frecuencia, perderse o no ser leído. Debido a la naturaleza asíncrona del correo electrónico, es más alto el potencial de pérdida de información. Los participantes en el proyecto por lo general se ajustan con rapidez a estas desventajas después de que han sucedido varios malos entendidos serios.

3.4.7 Grupos de noticias

Los **grupos de noticias** proporcionan una metáfora similar al correo electrónico. Sin embargo, en vez de enviar un mensaje a varios receptores, el emisor envía un mensaje a un grupo de noticias (al que también se le conoce como foro). Los receptores se subscriben al grupo de noticias que quieren leer. Se pueden controlar los accesos para envío y lectura a un grupo de noticias específico. Los grupos de noticias siguen un conjunto de estándares aceptados en forma amplia, similares a los del correo electrónico. Hay muchas herramientas comerciales y de dominio público disponibles para administrar un servidor de noticias. Con frecuencia el software para la lectura y envío de correo electrónico ya está integrado con lectores de grupos de noticias, lo cual facilita la integración entre el correo electrónico y los grupos de noticias.

Los grupos de noticias son adecuados para notificar y discutir entre personas que comparten un interés común. En un proyecto de software los equipos pueden tener varios grupos de noticias. Por ejemplo, en la figura 3-12 hay uno para enviar agendas y minutas de reuniones, otro para aclaraciones y otro más para otros asuntos. En vez de enumerar las direcciones de los miembros del equipo cada vez que se envía un mensaje, el emisor sólo tiene que nombrar al grupo de noticias de destino. Los modos típicos que pueden soportarse por medio de grupos de noticias incluyen petición de aclaraciones, petición de cambios, lanzamientos y lluvias de ideas.

Grupo de noticias para comunicación con el cliente	`cmu.academic.15-413.cliente`

Grupos de noticias globales leídos por todos los desarrolladores	`cmu.academic.15-413.anuncio`
	`cmu.academic.15-413.discusion`
	`cmu.academic.15-413.asuntos`
	`cmu.academic.15-413.equipo`

Grupos de noticias de equipos leídos principalmente por los miembros de un solo equipo (por ejemplo, el equipo de interfaz de usuario)	`...`
	`cmu.academic.15-413.iu.anuncio`
	`cmu.academic.15-413.iu.discusion`
	`cmu.academic.15-413.iu.minutas`

Grupos de noticias de propósito especial	`...` `cmu.academic.15-413.integracion-pruebas`

Figura 3-12 Un ejemplo de una estructura de un grupo de noticias.

3.4.8 World Wide Web

La **World Wide Web** (WWW, que también es llamada la "Web") proporciona al usuario una metáfora de hipertexto. Un navegador WWW despliega ante el usuario un documento que puede contener hipervínculos hacia otros documentos. Un localizador de recursos universal (URL, por sus siglas en inglés) está asociado con cada vínculo, y le describe al navegador la ubicación del documento de destino y su método de recuperación. La WWW ha tenido una popularidad creciente conforme se amplían los documentos de hipertexto para que contengan imágenes incrustadas, animaciones y scripts ejecutables.

En un proyecto de desarrollo de software, la WWW es ideal para la organización y para proporcionar acceso a información lanzada, como versiones preliminares y finales de documentos a entregar, código, referencias (por ejemplo, páginas iniciales de vendedores) y documentación sobre herramientas.

Aunque el tipo de documentos y la funcionalidad de los navegadores WWW evolucionan con rapidez, la Web no soporta de una forma igualmente directa el intercambio de información que cambia en forma rápida, como los mensajes de grupos de noticias. Sin embargo, con la integración de los grupos de noticias en Netscape y otros navegadores, la WWW puede desempeñar una función significativa en la infraestructura de comunicaciones de un proyecto.

La figura 3-13 despliega la página inicial del proyecto OWL [OWL, 1996]. Contiene referencias a las partes más importantes del sitio Web, como la versión más reciente de los documentos del proyecto y presentaciones de revisión, grupos de noticias, organigramas del equipo, páginas Web del equipo y referencias. El sitio Web del proyecto está administrado por un solo webmaster. Las páginas asociadas con cada equipo están administradas por los equipos con lineamientos del webmaster del proyecto.

3.4.9 Lotus Notes

Lotus Notes es una herramienta comercial (a diferencia de un conjunto de estándares, como en el caso del correo electrónico, los grupos de noticias y la WWW) que proporciona funcionalidad comparable a la de la Web. Sin embargo, proporciona una metáfora que está más cercana a un sistema de administración de base de datos que a un sistema de correo electrónico. Cada

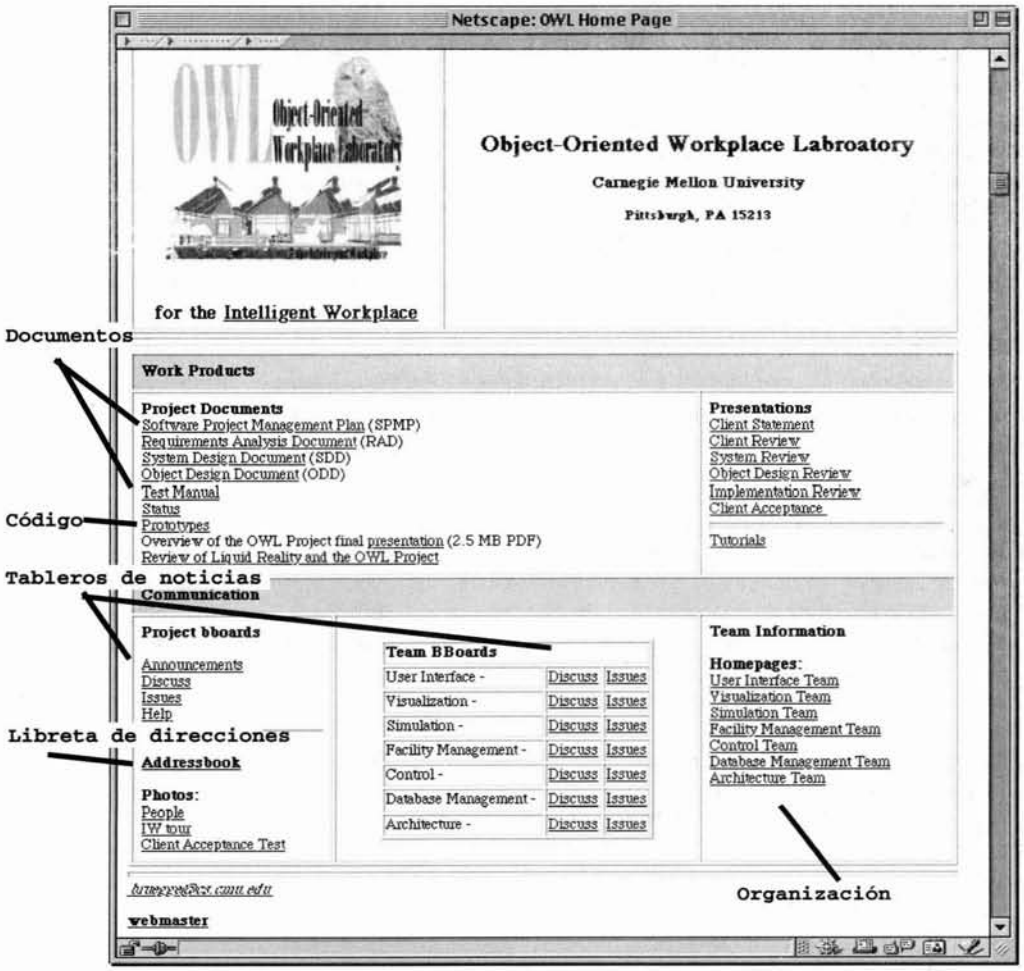


Figura 3-13 Un ejemplo de una página inicial de un proyecto para el proyecto OWL.

usuario ve el espacio de información como un conjunto de bases de datos. Las bases de datos contienen documentos compuestos por un conjunto de campos. Los documentos se crean y modifican mediante un conjunto de formularios. Los usuarios colaboran creando, compartiendo y modificando documentos. Cada base de datos puede tener diferentes plantillas y soportar funcionalidad específica de la aplicación. Por ejemplo, una base de datos puede soportar discusiones (vea la figura 3-8), y en este caso sus documentos son mensajes y respuestas. Otra base de datos puede soportar un sistema de seguimiento de peticiones de cambio, y en este caso los documentos pueden ser formularios para petición de cambios, bitácoras de revisiones o notas de lanzamiento.

Lotus Notes difiere con respecto a la Web atacando bien dos aspectos adicionales de la administración de información: control de acceso y réplica. El acceso puede controlarse al nivel de una base de datos (por ejemplo, negando a los desarrolladores el acceso a una base de datos de administración), de un documento (por ejemplo, impidiendo que un desarrollador modifique una petición de cambio enviada por un usuario) o de un campo (por ejemplo, permitiendo que sólo un administrador modifique el campo de autor de un documento). El control de acceso también puede especificarse por grupos de usuarios. El concepto de grupos permite que el administrador exprese el control de acceso según la organización del proyecto y no según sus individuos.

La principal diferencia entre el Lotus Notes y la Web ha sido que Notes es un estándar propio: tanto los programas cliente como servidor necesitan comprarse a IBM. La Web, por otro lado, es muy ubicua, debido a que se apoya en un conjunto de estándares públicos. Es posible construir una aplicación Web, accesible a usuarios con diferentes navegadores, por completo con software de dominio público (con los costos de confiabilidad, soporte y alguna funcionalidad). Para protegerse de la competencia de la Web y aprovechar el amplio uso de los navegadores Web, Lotus Notes introdujo Domino, un producto que permite que los navegadores Web tengan acceso a las bases de datos Lotus Notes mediante un navegador estándar.

Ejemplo. Cada equipo puede tener una base de datos para anuncios del equipo. Los documentos en la base de datos de anuncios son anuncios o réplicas. Sólo los miembros del equipo pueden colocar anuncios, pero cualquiera puede leer los anuncios y colocar réplicas. Se elige a un administrador de Lotus Notes para que organice a los participantes en el proyecto en grupos de acceso. Cada grupo de acceso corresponde a un equipo. Además, el administrador especifica el acceso a cada base de datos de anuncios de tal forma que sólo los miembros del equipo a los que pertenece la base de datos puedan colocar anuncios. Supongamos que Sally, un miembro del equipo de interfaz de usuario, es reasignada al equipo de notificación. El administrador sólo necesita quitar a Sally del grupo de acceso de interfaz de usuario y añadirla al grupo de acceso de notificación para reflejar el cambio en la organización. Si el administrador no hubiera organizado el acceso por grupos hubiera tenido que revisar y cambiar el acceso a todas las bases de datos a las que tenía acceso Sally.

3.4.10 Discusión

La comunicación del proyecto es omnipresente. Sucede en muchos niveles y bajo diferentes formas, y puede estar soportada con herramientas diferentes. La comunicación también es una herramienta fundamental para los participantes en el proyecto. Puede ser que no todos los participantes estén familiarizados con el último desarrollo tecnológico usado en el proyecto, pero todos deben tener alguna idea de la manera en que se disemina la información por el proyecto. Por ejemplo, los participantes que han estado usando correo electrónico a diario en proyectos anteriores pueden ser reacios a cambiar hacia una infraestructura alterna, como los grupos de noticias o Lotus Notes. Por esta razón, la infraestructura de comunicaciones, incluyendo la colección de herramientas que se usa para soportar la comunicación, necesita estar diseñada con meticulosidad y puesta en su lugar antes de iniciar el proyecto. Para facilitar esto se pueden usar los siguientes criterios:

- *Suficiencia.* ¿Todos los modos de comunicación presentes en el proyecto están soportados por una o más herramientas?
- *Complejidad.* ¿Se están usando demasiadas herramientas para soportar la comunicación? ¿Puede duplicarse o fragmentarse la información hasta el punto en que sea difícil de localizar?

- *Confiabilidad.* ¿Es confiable la herramienta de comunicación principal? ¿Cuál es su impacto en el proyecto en caso de que falle?
- *Mantenimiento.* ¿La infraestructura necesita un administrador de tiempo completo? ¿Se dispone de recursos para ese papel? ¿Se tienen las habilidades necesarias para ese papel?
- *Facilidad de transición.* ¿La metáfora presentada por la infraestructura es suficientemente familiar para los usuarios?

La transición es un paso crítico en la puesta a punto de la infraestructura de comunicaciones, mucho más que con otras herramientas: la infraestructura se usará sólo si la mayoría de los participantes la usa con efectividad. De no ser así, la comunicación sucederá mediante canales alternos *ad hoc*, como las conversaciones en los pasillos, y la diseminación de información será inconsistente y poco confiable. En la siguiente sección proporcionamos un ejemplo de infraestructura de comunicaciones de un proyecto y su transición.

3.5 Actividades de comunicación del proyecto

En esta sección examinamos las necesidades de comunicación de un proyecto de ejemplo, seleccionamos modos y mecanismos de comunicación para soportarlo y describimos su integración y transición de la infraestructura en el proyecto.

3.5.1 Identificación de las necesidades de comunicación

Este proyecto de ejemplo es un esfuerzo de ingeniería para crear algo que aún no existe (greenfield) con todos los nuevos desarrolladores. Las primeras fases de definición del problema y diseño preliminar las maneja en forma directa la administración en reuniones frente a frente. Los desarrolladores tienen muy poco o ningún acceso directo al cliente durante estas fases. Tampoco se espera que los equipos realicen inspecciones o ensayos sino hasta que haya madurado lo suficiente su nivel de experiencia.

El diseño preliminar permite que la administración defina la descomposición especial del sistema en subsistemas (vea el capítulo 11, *Administración del proyecto*). Cada subsistema se asigna a un equipo de desarrolladores. Además se forman equipos de funcionalidad cruzada (por ejemplo, equipo de documentación, equipo de integración) para apoyar a los equipos de subsistemas. Por último, cada equipo tiene coordinación con los equipos de funcionalidad cruzada para facilitar la transferencia de información entre equipos. La organización inicial del proyecto se muestra en la figura 3-14.

Cuando definen subsistemas, los administradores tratan de identificar grupos de funcionalidad que puedan ser desacoplados con facilidad para reducir las dependencias entre equipos de subsistemas. En consecuencia, se espera que la mayoría de las comunicaciones asíncronas sucedan entre los equipos y las coordinaciones, mientras que se espera que la comunicación en el ámbito del proyecto se realice en eventos calendarizados bien definidos, como revisiones del proyecto y lanzamientos. A priori, la administración ve los siguientes modos como críticos para la comunicación entre equipos:

- Revisión del cliente: trimestral
- Revisión del proyecto: mensual

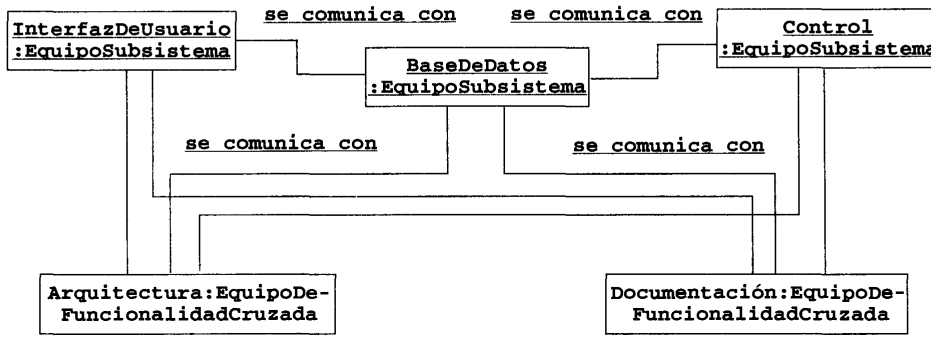


Figura 3-14 Un ejemplo de organización del proyecto usada como base para la infraestructura de comunicación (diagrama de objeto UML). Las asociaciones representan canales de comunicación por medio de coordinaciones.

- Lanzamientos: semanal
- Peticiones de aclaraciones: cuando se necesiten
- Peticiones de cambio: cuando se necesiten
- Resolución de problemas del proyecto: cuando se necesiten

En forma similar, la administración ve los siguientes modos como críticos para la comunicación dentro de los equipos:

- Revisión del estado: semanal
- Lluvia de ideas: semanal
- Resolución de problemas del equipo: cuando se necesite
- Peticiones de aclaraciones: cuando se necesiten
- Peticiones de cambio: cuando se necesiten

Dada la importancia cada vez mayor de las revisiones en la perspectiva de comunicación de la administración, deciden seleccionar las reuniones formales para realizar las revisiones del cliente y del proyecto hasta que se adquiera más experiencia en el uso de las herramientas groupware síncronas. El estado del equipo y la lluvia de ideas se manejan en una reunión de equipo semanal. La administración selecciona una herramienta groupware basada en foro para dar soporte a los lanzamientos y a la mayoría de las comunicaciones asíncronas. Por último, la administración espera que las conversaciones informales en pasillos y las coordinaciones manejen la comunicación durante las crisis en caso de que no funcionen los canales de comunicación formales.

3.5.2 Instalación de una infraestructura

Se crean dos conjuntos de foros para apoyar la comunicación del proyecto y de equipos, respectivamente. Los miembros se suscriben a todos los foros del proyecto y a los foros de sus equipos. Los foros del proyecto incluyen:

- *Anuncio.* Los eventos principales (por ejemplo, agenda de revisión, lanzamientos) son anunciados por la administración poniéndolos en este foro. Sólo la administración puede poner anuncios en este foro, y los miembros del proyecto pueden poner réplicas y leer todos los documentos.
- *Discusión.* Las peticiones de aclaraciones y peticiones de cambio del proyecto se colocan en este foro. Las discusiones acerca de las peticiones (por ejemplo, argumentos y soluciones alternas) se ponen como respuesta a los mensajes originales. Todos los miembros del proyecto pueden enviar a este foro y leer sus documentos.
- *Problemas.* En este foro se ponen problemas abiertos y su estado actual. Todos los miembros del proyecto pueden enviar a este foro y leer sus documentos.
- *Documentos.* En este foro se colocan las versiones más recientes de lo disponible del proyecto (por ejemplo, el documento de análisis de requerimientos, el documentos de diseño del sistema) y otros documentos internos del proyecto (por ejemplo, el plan de administración del proyecto de software). Sólo el equipo de documentación puede colocar documentos en este foro. Todos los miembros del proyecto pueden colocar réplicas (es decir, comentarios a los documentos) y leer los documentos.
- *Lista de equipo.* Este foro contiene descripciones del equipo disponible y su estado (por ejemplo, disponibilidad, propietario actual). Sólo el administrador del equipo puede enviar a este foro.

Los foros de equipos son similares a los foros del proyecto, a excepción de que soportan la comunicación del equipo. Los foros de equipos incluyen:

- *Discusiones del equipo*
- *Problemas del equipo*
- *Documentos del equipo*

Cada miembro del proyecto puede leer el foro de cualquier otro equipo. Los miembros del equipo sólo pueden enviar a los foros de su propio equipo. Observe que los foros pueden crearse tan pronto como sea relativamente estable la descomposición en subsistemas. Una vez que los foros y grupos de acceso están especificados se pueden crear las cuentas para miembros individuales conforme se asigna el personal al proyecto.

3.5.3 Organización de las revisiones del cliente y del proyecto

Las revisiones del cliente se realizan después del lanzamiento del documento de análisis de requerimientos y después de la entrega del sistema. Las revisiones del proyecto se realizan para revisar los documentos de diseño del sistema, el diseño detallado de objetos y las pruebas. Una revisión del proyecto también puede realizarse antes de la entrega como una prueba en seco para la prueba de aceptación del cliente.

El administrador del proyecto decide calendarizar todas las revisiones durante la fase de planeación (vea la tabla 3-5).

Tabla 3-5 Un ejemplo de un calendario de revisiones.

Revisión	Fecha	Producto disponible (debe estar listo una semana antes de la revisión)
Revisión del cliente	Semana 7	Documento de análisis de requerimientos
Revisión del diseño del sistema	Semana 9	Documento de diseño del sistema
Revisión del diseño de objetos	Semana 13 (2 sesiones)	Documento de diseño de objetos
Revisión interna	Semana 16	Pruebas unitarias y de integración
Prueba en seco de aceptación del cliente	Semana 17	Todo lo disponible del proyecto
Prueba de aceptación del cliente	Semana 17	Todo lo disponible del proyecto

La administración también presenta procedimientos para la organización de las revisiones:

1. Los productos disponibles que se están revisando deben estar listos una semana[2] antes de la revisión.
2. Poco después del lanzamiento, la administración publica un borrador de agenda en donde lista los temas a presentar por cada equipo. El borrador inicial de la agenda se coloca como un documento Lotus Notes en el foro *Anuncios* del proyecto.
3. Los candidatos para la presentación hacen comentarios a las agendas originales y refinan los temas de la presentación. La administración modifica la agenda con base en los comentarios.
4. Los presentadores envían sus diapositivas en respuesta a la agenda e incluyen las diapositivas en la respuesta. La administración coteja las diapositivas antes de la presentación y actualiza la agenda.

La administración también asigna la responsabilidad de secretario de actas, usando el mismo procedimiento, a un miembro del proyecto, a quien le enseñará la manera de levantar la minuta. Durante la revisión, el secretario de actas usa una laptop y registra con cuidado todas las preguntas de la audiencia y las respuestas. Por último, al día siguiente de la revisión, el secretario de actas y la administración reúnen sus notas y generan una lista de conceptos de acción a realizarse como resultado de la revisión y una lista de problemas abiertos que no se pudieron responder durante la revisión. Después de procesar las minutas se colocan en el foro *Anuncios*.

El énfasis en el uso de la infraestructura de comunicación para la coordinación de la organización de la revisión y el envío de diapositivas permite que se capture más información y, por lo tanto, se tenga accesible en el espacio de información del proyecto.

2. Esto deja un tiempo de holgura para los últimos documentos. Siendo realistas, algunos de los productos disponibles con frecuencia se entregan el día anterior a la reunión. Aquí, el asunto crítico es: 1) ¿Se puede poner a disposición de todos los participantes en la revisión el material disponible? y 2) ¿Tienen el tiempo suficiente para revisarlo?

3.5.4 Organización de reuniones de equipo semanales

La administración decide establecer una reunión de equipo semanal para todos los equipos, a fin de tener revisiones de estado, lluvia de ideas y solución de problemas. La reunión de equipo semanal se organiza y captura como se describe en la sección 3.4.3. Tomando en cuenta que el proyecto de ejemplo es un desarrollo de primera vez con personal nuevo, pocos miembros se conocen desde el punto de vista social y técnico. Además, pocos de ellos están familiarizados con los papeles y procedimientos de las reuniones formales. La administración aprovecha la oportunidad de la primera reunión de equipo semanal para presentar los procedimientos de la reunión, explicar la importancia de estos procedimientos y motivar a los miembros del equipo para que los usen. La figura 3-15 muestra la agenda enviada por la administración para la primera reunión.

El objetivo de la primera reunión es entrenar a los participantes mediante el ejemplo. Se motiva la discusión acerca de los procedimientos. A los participantes se les explica la reunión y los papeles del grupo, y se asignan al equipo para el resto del proyecto. Se enfatiza que el papel del moderador tiene como propósito incrementar la eficiencia de la reunión y no imponer decisiones. Se les enseña a los miembros del equipo que cualquier participante en la reunión puede asumir el papel de segundo moderador; esto es, cualquier participante puede intervenir en la discusión para que la reunión regrese a lo que indica la agenda. Se les enseñan a los participantes frases hechas para situaciones estándar (por ejemplo, *déjame asumir el papel de segundo moderador* significa "el tema de la discusión actual está fuera de la agenda. Regresemos al camino". *¿Podemos subir un nivel?* significa "la discusión se ha ido a un nivel de detalle que no es necesario para esta audiencia. De hecho, la mayoría de nosotros estamos perdidos"). En términos más generales, se les enseña a los miembros del equipo que es fácil perder el tiempo durante una reunión, y que el objetivo principal de cualquier reunión es comunicarse en forma eficiente y precisa para que puedan regresar a sus tareas respectivas.

La administración decide que se cambien los papeles con regularidad para que todos los participantes tengan oportunidad de desempeñar cada uno de los papeles. Esto tiene la ventaja de crear habilidades redundantes entre los participantes y compartir mejor la información. La desventaja es que, en el corto plazo, los participantes no tendrán tiempo para madurar en su papel y llegar así a ser muy eficientes en una tarea dada. El requerimiento de la asignación temprana de papeles, de la rotación de papeles y, en términos más generales, que los procedimientos de reunión estén establecidos desde el principio puede introducir turbulencia al inicio del proyecto, pero representa una inversión saludable a largo plazo. La administración toma la posición de que las reuniones diarias y las habilidades de comunicación deben estar bien establecidas antes de que aflore la necesidad de la comunicación manejada por crisis durante las actividades de implementación y codificación.

Los equipos son responsables de la asignación de los papeles en la reunión y de ponerlos en su foro *Anuncios* del equipo respectivo. Se requiere que el moderador de la reunión ponga el borrador inicial de la agenda, compuesto por las acciones tomadas de las minutas de la reunión anterior y los temas tomados del foro *Asuntos*.

El día anterior a la reunión de estado debe aparecer en el foro *Anuncios* del equipo: se requiere que el secretario de actas envíe las minutas el día siguiente a la reunión como una respuesta a la agenda correspondiente. Los demás miembros del equipo pueden hacer comentarios sobre la agenda y las minutas enviando respuestas. Luego, el moderador o el secretario de actas puede corregir el documento correspondiente.

Cuándo y dónde	Papel
Fecha: 9/01	**Moderador principal:** Alicia
Inicio: 4:30 p.m.	**Tomador de tiempo:** David
Fin: 5:30 p.m.	**Secretario de actas:** Eduardo
Edificio: Wean Hall	
Salón: 3420	

1. Objetivo

Familiarizarse con los papeles de administración del proyecto para un proyecto de mediana escala con una jerarquía de dos niveles. En particular:

- Comprender la diferencia entre un papel y una persona
- Asignación de papeles a personas
- Finalizar la reunión
- Primer conjunto de acciones para la siguiente reunión

2. Intercambio de información sobre el estado del proyecto [tiempo asignado: 40 minutos]

2.1 La manera en que se organiza una reunión

Reglas básicas de una reunión

- Escucha activa
- Participación activa
- Puntualidad
- No debe haber reuniones de uno a uno o con el que está al lado
- Respetar la agenda
- Ahorrar tiempo
- Disposición para lograr el consenso
- Libertad para revisar el proceso y las reglas básicas

Papeles de la reunión

- Moderador principal
- Tomador de tiempo
- Secretario de actas
- Escribano

2.2 A continuación se presenta la agenda

Se omite por brevedad

3. Temas a discusión [tiempo asignado: 15 minutos]

3.1 Directorio del equipo

3.2 Asignación de papeles para la reunión

3.3 Asignación de papeles de grupo

4. Cierre [tiempo asignado: 5 minutos]

4.1 Revisión y asignación de nuevas acciones

4.2 Crítica de la reunión

Figura 3-15 Agenda para la primera reunión semanal de equipo.

Con frecuencia, los papeles y procedimientos de la reunión se perciben como una sobrecarga. La administración está consciente de esa percepción e invierte tiempo al inicio del proyecto para ilustrar los beneficios de los procedimientos de reunión. En la primera semana del proyecto la administración revisa de manera sistemática las agendas y minutas de las primeras reuniones semanales, les sugiere mejoras para ahorrar tiempo a los moderadores (por ejemplo, mantener un documento activo que contiene temas abiertos y acciones activas desde donde se pueden cortar y pegar para formar la agenda) y a los secretarios de actas (por ejemplo, enfocarse primero en la captura de las acciones y problemas no resueltos y luego en la discusión).

3.5.5 Revisión de los asuntos de transición

Puede ser difícil el aprendizaje del uso de una nueva infraestructura de comunicación en un proyecto [Orlikowski, 1992]. Como se dijo antes, es crítico que la transición de la infraestructura de comunicación suceda al principio del proyecto. Por otro lado, una vez que se obtiene una masa crítica de información y usuarios, los demás usuarios y más información se abrirán paso en la infraestructura de comunicaciones. Por otro lado, si no se obtiene rápido la masa crítica, se establecerán canales informales alternos y se consolidarán, dificultando después su corrección.

La administración puede motivar la transición poniendo a disposición un máximo de información estructurada (por ejemplo, el enunciado del problema, el plan de administración del software, plantillas de agenda y documentos), requiriendo que los participantes en el proyecto realicen tareas simples usando la infraestructura (por ejemplo, colocando sus papeles dentro del equipo en el foro *Anuncios*) y proporcionando retroalimentación utilizando la infraestructura de comunicaciones.

3.6 Ejercicios

1. Usted es un miembro del equipo de interfaz de usuario. Es responsable del diseño e implementación de formularios para recopilar información acerca de los usuarios del sistema (por ejemplo, nombre, apellido, dirección, dirección de correo electrónico, nivel de experiencia). La información que se recopila se guarda en la base de datos y la usa el subsistema de reportes. No está seguro de cuáles campos son información requerida y cuáles son opcionales.

   ¿Cómo lo investiga?
2. Usted ha sido reasignado del equipo de interfaz de usuario al equipo de base de datos, debido a falta de personal y replaneación. La fase de implementación ya va algo adelantada.

   ¿En cuál papel sería más productivo tomando en cuenta su conocimiento del diseño e implementación de la interfaz de usuario?
3. Suponga que el ambiente de desarrollo son estaciones de trabajo Unix y que el equipo de documentación usa plataformas Macintosh para la escritura de la documentación. El cliente requiere que los documentos se tengan disponibles en plataformas Windows. Los desarrolladores producen la documentación de diseño usando FrameMaker. El equipo de documentación usa Microsoft Word para la documentación en el nivel de usuario. El cliente envía las correcciones en papel y no necesita modificar los documentos entregados.

¿Cómo podría especificarse el flujo de información entre los desarrolladores, los escritores técnicos y el cliente (por ejemplo, formato, herramientas, etc.) de forma tal que se minimice la duplicación de archivos y, al mismo tiempo, se satisfagan las preferencias de herramientas y requerimientos de plataforma de cada cual?

4. ¿Cuáles cambios de organización e infraestructura de comunicaciones le recomendaría a un sucesor del proyecto Ariane 5 como consecuencia de la falla del Ariane 501 que se describe al inicio de este capítulo?

Referencias

[FRIEND, 1994] *FRIEND Project Documentation.* School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1994.

[Grudin, 1988] J. Grudin, "Why CSCW applications fail: Problems in design and evaluation of organization interfaces", *Proceedings CSCW '88,* Portland, OR, 1988.

[Kayser, 1990] T. A. Kayser, *Mining Group Gold.* Serif, El Segundo, CA, 1990.

[Lions, 1996] J.-L. Lions, *ARIANE 5 Flight 501 Failure: Report by the Inquiry Board,* http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 1996.

[Orlikowski, 1992] W. J. Orlikowski, "Learning from Notes: Organizational issues in groupware implementation", *Conference on Computer-Supported Cooperative Work Proceedings*, Toronto, Canadá, 1992.

[OWL, 1996] *OWL Project Documentation,* School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1996.

[Reeves y Shipman, 1992] B. Reeves y F. Shipman, "Supporting communication between designers with artifact-centered evolving information spaces", *Conference on Computer-Supported Cooperative Work Proceedings*, Toronto, Canadá 1992.

[Saeki, 1995] M. Saeki, "Communication, collaboration, and cooperation in software development—How should we support group work in software development?", in *Asia-Pacific Software Engineering Conference Proceedings,* Brisbane, Australia, 1995.

[Seaman y Basili, 1997] C. B. Seaman y V. R. Basili, "An empirical study of communication in code inspections", *Proceedings of the 19th International Conference on Software Engineering,* Boston, MA, 1997.

[Subrahmanian *et al.*, 1997] E. Subrahmanian, Y. Reich, S. L. Konda, A. Dutoit, D. Cunningham, R. Patrick, M. Thomas, y A. W. Westerberg, "The *n*–dim approach to building design support systems", *Proceedings of ASME Design Theory and Methodology DTM '97,* ASME, Nueva York, 1997.

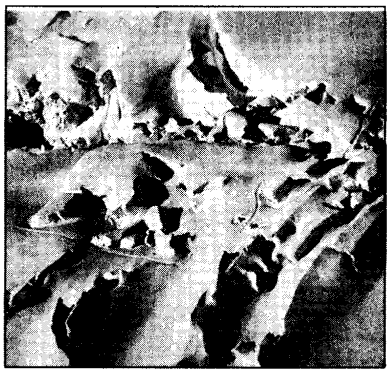
[Teamwave, 1997] Teamwave Inc., http://www.teamwave.com, 1997.

PARTE II

Manejo de la complejidad

4

4



Obtención de requerimientos

Nadie está exento de cometer errores.
Lo importante es aprender de ellos.

—Karl Popper, en Objective Knowledge: An Evolutionary Approach

Un *requerimiento* es una característica que debe tener el sistema o una restricción que debe satisfacer para que sea aceptado por el cliente. La *ingeniería de requerimientos* pretende definir los requerimientos del sistema que se está construyendo. La ingeniería de requerimientos incluye dos actividades principales: la *obtención de los requerimientos*, que da como resultado una especificación del sistema que el cliente comprende, y el *análisis*, que da como resultado un modelo de análisis que los desarrolladores pueden interpretar sin ambigüedad. La obtención de requerimientos es la más retadora de las dos, debido a que requiere la colaboración de varios grupos de participantes con diferentes niveles de conocimientos. Por un lado, el cliente y los usuarios son expertos en sus dominios y tienen una idea general de lo que debe hacer el sistema. Sin embargo, a menudo tienen muy poca experiencia en el desarrollo de software. Por otro lado, los desarrolladores tienen experiencia en la construcción de sistemas, pero con frecuencia tienen muy poco conocimiento del ambiente diario de los usuarios.

Los escenarios y los casos de uso proporcionan herramientas para llenar este hueco. Un *escenario* describe un ejemplo del uso del sistema desde el punto de vista de una serie de interacciones entre el usuario y el sistema. Un *caso de uso* es una abstracción que describe una clase de escenarios. Tanto los escenarios como los casos de uso se escriben en lenguaje natural, una forma que es comprensible para el usuario.

En este capítulo nos enfocamos en la obtención de requerimientos basados en escenarios. Los desarrolladores obtienen los requerimientos observando a los usuarios y entrevistándolos. Primero los desarrolladores representan los procesos de trabajo actuales del usuario como escenarios tal como son, y luego desarrollan escenarios visionarios en donde se describe la funcionalidad que proporcionará el sistema futuro. El cliente y los usuarios validan la descripción del sistema revisando los escenarios y probando prototipos pequeños proporcionados por los desarrolladores. Conforme madura y se estabiliza la definición del sistema, los desarrolladores y el cliente se ponen de acuerdo en una especificación del sistema en forma de casos de uso.

4.1 Introducción: ejemplos de utilidad

Considere los siguientes ejemplos:[1]

¿Metros o kilómetros?

Durante un experimento se dirigió un rayo láser hacia un espejo que estaba en el transbordador espacial Discovery. La prueba trataba de que el rayo láser se reflejara hacia la cima de una montaña. El usuario introdujo la elevación de la montaña como "3057", suponiendo que las unidades eran metros. La computadora interpretó la cifra en kilómetros, y el rayo láser fue reflejado lejos de la Tierra, hacia una montaña hipotética de 3057 kilómetros de alto.

Punto decimal frente a separador de millares

En Estados Unidos, los puntos decimales se representan con un punto (".") y los separadores de millares se representan con una coma (","). En Alemania, el punto decimal se representa con una coma y el separador de miles con un punto. Supongamos que un usuario alemán, que está consciente de ambas convenciones, está viendo un catálogo en línea con los precios listados en dólares. ¿Cuál convención debería usarse para evitar confusiones?

Patrones estándares

En el editor de textos Emacs, el comando `<Control-x><Control-q>` produce la salida del programa. Si necesita guardar algún archivo, el editor le preguntará al usuario "¿Se guarda el archivo miDocumento.txt? (s o n)". Si el usuario responde `y` el editor guarda el archivo antes de salir. Muchos usuarios confían en este patrón y en forma sistemática teclean la secuencia `<control-x><control-q>y` cuando salen de un editor. Sin embargo, otros editores al cerrarse hacen la pregunta: "¿Está seguro de que quiere salir? (s o n)". Cuando los usuarios cambian de Emacs a uno de estos editores, no guardarán su trabajo hasta que se las arreglen para romper este patrón.

La obtención de requerimientos trata sobre la comunicación entre desarrolladores, clientes y usuarios para definir un nuevo sistema. Si no hay una comunicación y comprensión del dominio de cada uno de ellos se tendrá como resultado un sistema difícil de usar o que simplemente no apoya el trabajo del usuario. Resulta caro corregir los errores que se introducen durante la obtención de requerimientos ya que, por lo general, se les descubre tarde en el proceso, con frecuencia tan tarde como en la entrega. Tales errores incluyen funcionalidad faltante que el sistema debería haber soportado, funcionalidad que se especificó de manera incorrecta, interfaces de usuario que son confusas o inútiles y funcionalidad que es obsoleta. Los métodos para la obtención de requerimientos pretenden mejorar la comunicación entre desarrolladores, clientes y usuarios. Los desarrolladores construyen un modelo del dominio de aplicación observando a los usuarios en su ambiente (por ejemplo, análisis de tareas). Los desarrolladores seleccionan una representación que sea comprensible para los clientes y usuarios (por ejemplo, escenarios y casos de uso). Los desarrolladores validan el modelo del dominio de aplicación construyendo prototipos simples de la interfaz de usuario y recopilando retroalimentación de los usuarios potenciales (por ejemplo, elaboración rápida de prototipo, pruebas de utilidad).

En la siguiente sección proporcionamos una panorámica de la obtención de requerimientos y su relación con las demás actividades de desarrollo de software. En la sección 4.3 definimos los conceptos principales usados en este capítulo. En la sección 4.4 tratamos las actividades para la obtención de requerimientos. En la sección 4.5 tratamos las actividades administrativas relacionadas con la obtención de requerimientos.

1. Ejemplos tomados y adaptados de [Nielsen, 1993] y del foro RISK.

4.2 Una panorámica de la obtención de requerimientos

La obtención de requerimientos se enfoca en la descripción del propósito del sistema. El cliente, los desarrolladores y los usuarios identifican un área problema y definen un sistema que ataca el problema. A tal definición se le llama **especificación del sistema** y sirve como contrato entre el cliente y los desarrolladores. La especificación del sistema se estructura y formaliza durante el análisis (capítulo 5, *Análisis*) para producir un modelo de análisis (vea la figura 4-1). Tanto la especificación del sistema como el modelo de análisis representan la misma información. Difieren sólo en el lenguaje y la notación que usan. La especificación del sistema está escrita en lenguaje natural, mientras que el modelo de análisis se expresa, por lo general, en una notación formal o semiformal. La especificación del sistema soporta la comunicación con el cliente y los usuarios. El modelo de análisis soporta la comunicación entre desarrolladores. Ambos son modelos del sistema en el sentido de que tratan de representar con precisión los aspectos externos del sistema. Tomando en cuenta que ambos modelos representan los mismos aspectos del sistema, la obtención de requerimientos y el análisis suceden en forma concurrente e iterativa.

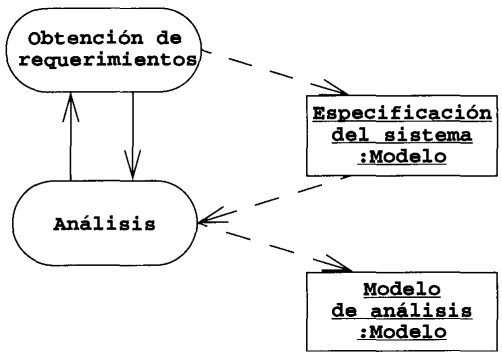


Figura 4-1 Productos de la obtención de requerimientos y el análisis (diagrama de actividad UML).

La obtención de requerimientos y el análisis se enfocan sólo en la visión del sistema que tiene el usuario. Por ejemplo, la funcionalidad del sistema, la interacción entre el usuario y el sistema, los errores que el sistema puede detectar y manejar y las condiciones ambientales en las que funciona el sistema son parte de los requerimientos. La estructura del sistema, la tecnología de implementación seleccionada para construir el sistema, el diseño del sistema, la metodología de desarrollo y otros aspectos que no son visibles en forma directa para el usuario no son parte de los requerimientos.

La obtención de requerimientos incluye las siguientes actividades.

- **Identificación de actores.** Durante esta actividad los desarrolladores identifican los diferentes tipos de usuario que soportará el sistema futuro.
- **Identificación de escenarios.** Durante esta actividad los desarrolladores observan a los usuarios y desarrollan un conjunto de escenarios detallados para la funcionalidad típica que proporcionará el sistema futuro. Los escenarios son ejemplos concretos del uso del sistema

futuro. Los desarrolladores usan estos escenarios para comunicarse con los usuarios y profundizar su comprensión del dominio de aplicación.

- **Identificación de casos de uso.** Una vez que los desarrolladores y usuarios se ponen de acuerdo en un conjunto de escenarios, los desarrolladores derivan a partir de los escenarios un conjunto de casos de uso que representa por completo al sistema futuro. Mientras que los escenarios son ejemplos concretos que ilustran un solo caso, los casos de uso son abstracciones que describen todos los casos posibles. Cuando se describen los casos de uso los desarrolladores determinan el alcance del sistema.
- **Refinamiento de los casos de uso.** Durante esta actividad los desarrolladores se aseguran que la especificación del sistema esté completa detallando cada caso de uso y describiendo el comportamiento del sistema en presencia de errores y condiciones excepcionales.
- **Identificación de las relaciones entre casos de uso.** Durante esta actividad los desarrolladores consolidan el modelo de caso de uso eliminando redundancias. Esto asegura que la especificación del sistema sea consistente.
- **Identificación de requerimientos no funcionales.** Durante esta actividad los desarrolladores, usuarios y clientes se ponen de acuerdo en aspectos que son visibles ante el usuario pero que no están relacionados en forma directa con la funcionalidad. Esto incluye restricciones en el desempeño del sistema, su documentación, los recursos que consume, su seguridad y su calidad.

Durante la obtención de requerimientos los desarrolladores consultan muchas fuentes de información diferentes, incluyendo documentos proporcionados por el cliente acerca del dominio de aplicación, manuales y documentación técnica de sistemas heredados que serán reemplazados por el sistema futuro y, lo más importante, a los clientes y usuarios mismos. La mayor interacción de los desarrolladores con los usuarios y los clientes se da durante la obtención de requerimientos. Nos enfocamos en tres métodos para la obtención de información y la toma de decisiones con los usuarios y clientes:

- El **diseño conjunto de aplicaciones** (JAD) se enfoca en la obtención de consenso entre desarrolladores, usuarios y clientes mediante el desarrollo conjunto de la especificación del sistema.
- El **conocimiento del análisis de tareas** (KAT) se enfoca en la obtención de información de los usuarios mediante la observación.
- Las **pruebas de utilidad** se enfocan en la validación del modelo de obtención de requerimientos con el usuario mediante métodos diversos.

4.3 Conceptos de la obtención de requerimientos

En esta sección describimos los principales conceptos de la obtención de requerimientos que usamos en este capítulo. En particular describimos:

- Requerimientos funcionales (sección 4.3.1)
- Requerimientos no funcionales y seudorrequerimientos (sección 4.3.2)
- Niveles de descripción (sección 4.3.3)
- Corrección, suficiencia, consistencia, claridad y realismo (sección 4.3.4)

- Verificabilidad y rastreabilidad (sección 4.3.5)
- Ingeniería a partir de cero (greenfield), reingeniería e ingeniería de interfaz (sección 4.3.6)

Describimos las actividades para la obtención de requerimientos en la sección 4.4.

4.3.1 Requerimientos funcionales

Los **requerimientos funcionales** describen las interacciones entre el sistema y su ambiente, en forma independiente a su implementación. El ambiente incluye al usuario y cualquier otro sistema externo con el cual interactúe el sistema. Por ejemplo, el siguiente es un ejemplo de requerimientos funcionales del RelojSat, un reloj que se reajusta a sí mismo sin intervención del usuario:

Requerimientos funcionales del RelojSat

El RelojSat es un reloj de pulsera que muestra el tiempo basado en su ubicación actual. El RelojSat usa los satélites GPS (sistema de posicionamiento global) para determinar su ubicación y estructuras de datos internos para convertir esta ubicación en una zona horaria. La información que está guardada en el reloj y su precisión en la medición del tiempo (una incertidumbre de un centésimo de segundo a lo largo de cinco años) son tales que el propietario del reloj nunca necesita volver a ajustar la hora. El RelojSat ajusta la hora y la fecha mostradas conforme el propietario del reloj cruza zonas horarias y fronteras políticas (por ejemplo, el tiempo estándar contra el tiempo de ahorro de luz de día). Por esta razón, el RelojSat no tiene botones ni controles para el usuario.

El RelojSat tiene una pantalla de dos renglones y muestra en el renglón superior la hora (hora, minuto, segundo y zona horaria) y en la línea inferior la fecha (día de la semana, día, mes, año). La tecnología usada en la pantalla es tal que el propietario del reloj puede ver la fecha y la hora aun bajo condiciones de poca iluminación.

Cuando un nuevo país o estado instituye reglas diferentes para el tiempo de ahorro de luz de día, el propietario del reloj puede mejorar el software del reloj usando el dispositivo serial WebificarReloj (que se proporciona cuando se compra el reloj) y una computadora personal conectada a Internet. El RelojSat se apega a las interfaces física, eléctrica y de software definidas por la API 2.0 de WebificarReloj.

Los requerimientos funcionales anteriores se enfocan sólo en las interacciones posibles entre el RelojSat y su mundo externo (es decir, el propietario del reloj, los GPS y WebificarReloj). La descripción anterior no se enfoca en ninguno de los detalles de implementación (por ejemplo, procesador, lenguaje, tecnología de pantalla).

4.3.2 Requerimientos no funcionales y seudorrequerimientos

Los **requerimientos no funcionales** describen aspectos del sistema visibles por el usuario que no se relacionan en forma directa con el comportamiento funcional del sistema. Los requerimientos no funcionales incluyen restricciones cuantitativas, como el tiempo de respuesta (es decir, qué tan rápido reacciona el sistema ante los comandos del usuario) o precisión (es decir, qué tan precisas son las respuestas numéricas del sistema). Los siguientes son los requerimientos no funcionales del RelojSat:

Requerimientos no funcionales del RelojSat

El RelojSat determina su ubicación usando satélites GPS y, por lo tanto, sufre las mismas limitaciones que todos los demás dispositivos GPS (por ejemplo, precisión de cien pies, incapacidad para determinar la ubicación en determinados momentos del día en regiones montañosas). Durante los periodos sin señal el RelojSat supone que no cruza una zona horaria o una frontera política. El RelojSat corrige su zona horaria tan pronto como recupera la señal.

La vida de la batería del RelojSat está limitada a cinco años, que es el ciclo de vida estimado de la caja del RelojSat. La caja del RelojSat no está diseñada para que se abra después de haberlo fabricado, impidiendo el reemplazo de la batería y las reparaciones. En vez de ello, el RelojSat tiene un precio tal que se espera que el propietario del reloj compre uno nuevo para reemplazar a uno antiguo o defectuoso.

Los **seudorrequerimientos** son requerimientos impuestos por el cliente que restringen la implementación del sistema. Los seudorrequerimientos típicos son el lenguaje de implementación y la plataforma en que se implementará el sistema. Para desarrollos que son críticos para la vida, los seudorrequerimientos incluyen, con frecuencia, requerimientos de procesos y documentación (por ejemplo, el uso de un método de especificación formal, la versión completa terminada de todos los productos de trabajo). Los seudorrequerimientos no tienen, por lo general, un efecto directo en la opinión del usuario acerca del sistema. Los siguientes son los seudorrequerimientos del RelojSat:

Seudorrequerimientos del RelojSat

Todo el software relacionado asociado con el RelojSat, incluyendo el software a bordo, se escribirá usando Java para apegarse a la política actual de la compañía.

El análisis es una actividad de modelado. El desarrollador construye un modelo que describe la realidad tal como se ve desde el punto de vista del usuario. El modelado consiste en la identificación y clasificación de fenómenos del mundo real (por ejemplo, aspectos del sistema en construcción) hacia conceptos. La figura 4-2 es un diagrama de clase UML que representa las relaciones entre los modelos y la realidad. En este diagrama se dice que un modelo es correcto si cada concepto del modelo corresponde a un fenómeno relevante. El modelo está completo si todos los fenómenos relevantes están representados por, al menos, un concepto. El modelo es consistente si todos los conceptos representan fenómenos de la misma realidad (es decir, si un modelo es inconsistente debe representar aspectos de dos realidades diferentes).

4.3.3 Niveles de descripción

Los requerimientos describen un sistema y su interacción con el ambiente que lo rodea, como los usuarios, sus procesos de trabajo y otros sistemas. La mayoría de los métodos de análisis de requerimientos se han enfocado en la descripción del sistema. Sin embargo, cuando se usan casos de uso y escenarios se hace evidente que también es necesario describir el ambiente en que operará el sistema. Primero, los desarrolladores, por lo general, no conocen ni comprenden al principio el ambiente operativo y necesitan revisar su comprensión con los usuarios. Segundo, es probable que el ambiente cambie y, por lo tanto, los desarrolladores deberán capturar todas las suposiciones que hacen acerca del ambiente. En general, hay cuatro niveles de descripción, que pueden explicarse de manera uniforme con casos de uso [Paech, 1998]. A continuación los listamos desde el más general hasta el más específico:

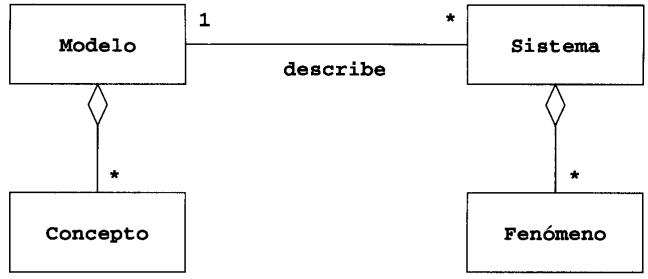


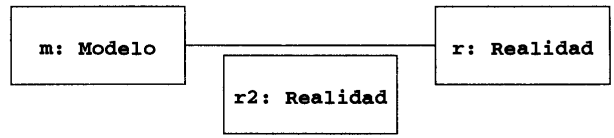
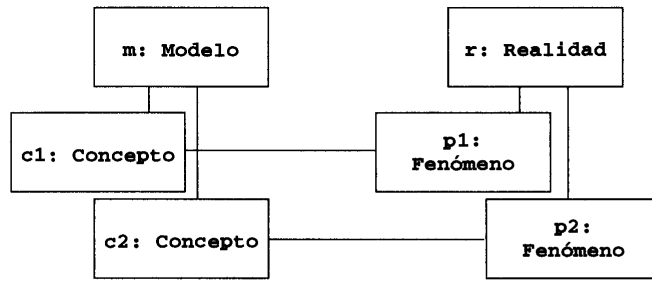
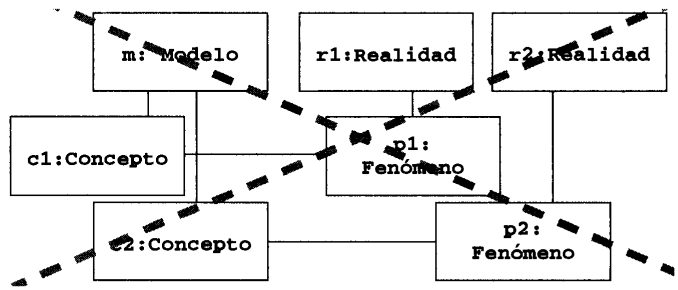
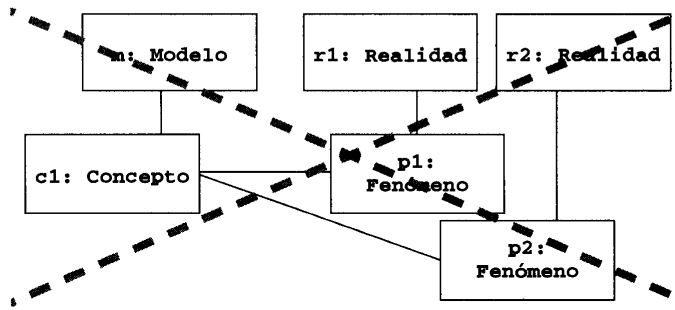
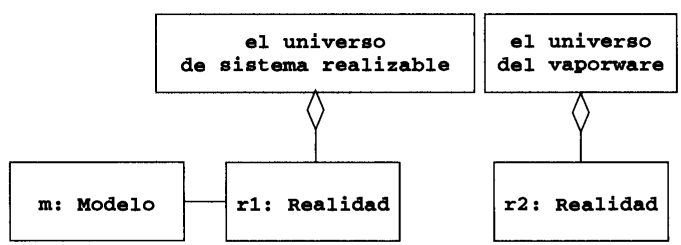
Figura 4-2 Un `Sistema` es una colección de `Fenómenos` del mundo real. Un modelo es una colección de conceptos que representan a los fenómenos del sistema. Muchos modelos pueden representar diferentes aspectos del sistema. Un modelo no ambiguo corresponde sólo a un sistema.

- **División del trabajo.** Este conjunto de casos de uso describe los procesos de trabajo de los usuarios que son relevantes para el sistema. También se describe la parte del proceso soportada por el sistema, pero el meollo está en la definición de las fronteras entre los usuarios y el sistema.
- **Funciones del sistema específicas de la aplicación.** Este conjunto de casos de uso describe las funciones proporcionadas por el sistema que están relacionadas con el dominio de aplicación.
- **Funciones del sistema específicas del trabajo.** Este conjunto de casos de uso describe las funciones de apoyo del sistema que no están relacionadas con el dominio de aplicación. Éstas incluyen funciones para administración de archivos, funciones de agrupamiento, funciones para deshacer, etc. Estos casos de uso se extenderán durante el diseño del sistema cuando estemos discutiendo condiciones de frontera conocidas, como la inicialización del sistema, el apagado y las políticas para el manejo de excepciones.
- **Diálogo.** Este conjunto de casos de uso describe las interacciones entre los usuarios y la interfaz de usuario del sistema. El enfoque está en el diseño de la resolución del flujo de control y asuntos de disposición.

4.3.4 Corrección, suficiencia, consistencia, claridad y realismo

Los requerimientos se validan en forma continua con el cliente y el usuario. La validación es un paso crítico en el proceso de desarrollo, tomando en cuenta que tanto el cliente como el desarrollador dependen de la especificación del sistema. La validación de requerimientos involucra la revisión para ver si la especificación es correcta, completa, consistente, realista y no es ambigua. Una especificación es **correcta** si representa la visión del cliente del sistema (es decir, todo lo que hay en el modelo de requerimientos representa con precisión un aspecto del sistema). Es **completa** si se describen todos los escenarios posibles que hay en el sistema, incluyendo el comportamiento excepcional (es decir, en el modelo de requerimientos están representados todos los aspectos del sistema). La especificación del sistema es **consistente** si no se contradice a sí misma. La especificación del sistema **no es ambigua** si está definido exactamente un sistema (es decir, no es posible interpretar la especificación en dos o más formas diferentes). Por último, es **realista** si el sistema puede implementarse dentro de esas restricciones. Estas propiedades se ilustran con diagramas de instancia UML en la tabla 4-1.

Tabla 4-1 Propiedades de la especificación que se revisan durante la validación.

Correcta: el modelo describe la realidad de interés para el cliente y no otra realidad	
Completa: todo fenómeno de interés está descrito en el modelo mediante un concepto	
Consistente: todos los conceptos del modelo corresponden a fenómenos de la misma realidad	
No es ambigua: todos los conceptos del modelo corresponden exactamente a un fenómeno	
Realista: el modelo describe una realidad que puede existir	

La corrección y suficiencia de la especificación del sistema con frecuencia son difíciles de establecer, en especial antes de que exista el sistema. Tomando en cuenta que la especificación del sistema sirve como una base contractual entre el cliente y los desarrolladores, ambas partes deben revisar en forma minuciosa la especificación del sistema. Además, en las partes del sistema que presentan un alto riesgo deben elaborarse prototipos o hacerse simulaciones para demostrar su factibilidad o para obtener retroalimentación del usuario. En el caso del RelojSat descrito antes, se construiría una maqueta del reloj usando un reloj tradicional y se realizaría una encuesta entre los usuarios para recopilar sus impresiones iniciales. Un usuario podría hacer la observación de que quisiera que el reloj pudiera desplegar los formatos de fecha americano y europeo.

4.3.5 Verificabilidad y rastreabilidad

Dos propiedades deseables de una especificación del sistema son que sea verificable y rastreable. La especificación es **verificable** si, una vez que se construye el sistema, se puede diseñar una prueba repetible para demostrar que el sistema satisface los requerimientos. Por ejemplo, una falla de tiempo medio durante cien años para el RelojSat sería difícil de lograr (suponiendo, en primer lugar, que fuera realista). Los siguientes requerimientos son ejemplos adicionales de requerimientos no verificables:

- *El producto debe tener una buena interfaz de usuario* (no se define buena).
- *El producto debe estar libre de errores* (se requieren muchos recursos para determinarlo).
- *El producto debe responder al usuario en menos de un segundo en la mayoría de los casos* (no se define "en la mayoría de los casos").

Una especificación del sistema es **rastreable** si cada función del sistema puede rastrearse hasta su conjunto de requerimientos correspondiente. La rastreabilidad no es una restricción en el contenido de la especificación sino, más bien, en su organización. La rastreabilidad facilita el desarrollo de pruebas y la validación sistemática del diseño contra los requerimientos.

4.3.6 Ingeniería a partir de cero (greenfield), reingeniería e ingeniería de interfaz

Las actividades para la obtención de requerimientos pueden clasificarse en tres categorías dependiendo del origen de los mismos. En la **ingeniería a partir de cero (greenfield)** el desarrollo comienza sin nada, no existe sistema anterior y los requerimientos se extraen de los usuarios y del cliente. Un proyecto de ingeniería a partir de cero se activa por una necesidad del usuario o por la creación de un nuevo mercado. El RelojSat es un proyecto de ingeniería a partir de cero.

Un proyecto de **reingeniería** es el rediseño y reimplementación de un sistema existente activado por los coordinadores de tecnología o por nuevo flujo de información [Hammer y Champy, 1993]. A veces se amplía la funcionalidad del nuevo sistema, pero el propósito inicial del sistema sigue siendo el mismo. Los requerimientos del nuevo sistema se extraen de un sistema existente.

Un proyecto de **ingeniería de interfaz** es el rediseño de la interfaz de usuario de un sistema existente. El sistema heredado se deja intacto, a excepción de su interfaz, la cual se rediseña y

reimplementa. Este tipo de proyecto es un proyecto de reingeniería en el cual el sistema heredado no puede descartarse sin entrañar altos costos. En esta sección examinamos la manera en que se realiza la obtención de requerimientos en estas tres situaciones.

En ambas reingenierías y en la ingeniería a partir de cero, los desarrolladores necesitan recopilar la mayor cantidad posible de información a partir del dominio de aplicación. Esta información puede encontrarse en manuales de procedimientos, documentación distribuida a los nuevos empleados, el manual del sistema anterior, glosarios, anotaciones de trampas y notas desarrolladas por los usuarios, y entrevistas con los usuarios y el cliente. Debe señalarse que aunque las entrevistas con los usuarios son una herramienta inestimable, no recopilan la información necesaria si no se hacen las preguntas relevantes. Los desarrolladores primero deben obtener un conocimiento firme del dominio de aplicación antes que puedan usar el enfoque directo.

A continuación describimos las actividades para la obtención de requerimientos que dan como resultado una especificación del sistema.

4.4 Actividades para la obtención de requerimientos

En esta sección describimos las actividades para la obtención de requerimientos. Con ellas se establece la correspondencia entre un enunciado del problema y una especificación del sistema que representamos como un conjunto de actores, escenarios y casos de uso (vea el capítulo 2, *Modelado con UML*). Exponemos la heurística y métodos para la extracción de requerimientos a partir de los usuarios y modelamos el sistema en función de estos conceptos. Las actividades para la obtención de requerimientos incluyen:

- Identificación de los actores (sección 4.4.1)
- Identificación de los escenarios (sección 4.4.2)
- Identificación de los casos de uso (sección 4.4.3)
- Refinamiento de los casos de uso (sección 4.4.4)
- Identificación de las relaciones entre casos de uso (sección 4.4.5)
- Identificación de los objetos participantes (sección 4.4.6)
- Identificación de los requerimientos no funcionales (sección 4.4.7)

Los métodos que se describen en esta sección están adaptados de *Object-Oriented Software Engineering (OOSE)* [Jacobson *et al.*, 1992], *The Unified Software Development Process* [Jacobson *et al.*, 1999] y "Design objects and their interactions: A brief look at responsability-driven design" [Wirfs-Brock *et al.*, 1990].

4.4.1 Identificación de los actores

Los actores representan entidades externas que interactúan con el sistema. Un actor puede ser un sistema humano o uno externo. En el ejemplo RelojSat, el propietario del reloj, los satélites GPS y el dispositivo serial WebificarReloj, son actores (vea la figura 4-3). Todos ellos interactúan e intercambian información con el RelojSat. Sin embargo, observe que todos ellos tienen interacciones específicas con el RelojSat: el propietario del reloj lo porta y observa, el reloj rastrea la señal de los satélites GPS y WebificarReloj transfiere nuevos datos hacia el reloj. Los actores definen clases de funcionalidad.

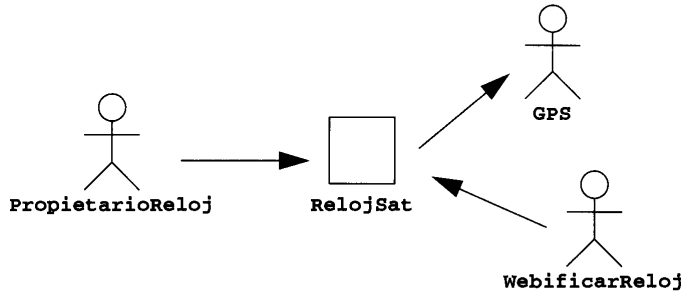


Figura 4-3 Los actores del sistema RelojSat. `PropietarioReloj` mueve el reloj (posiblemente cruzando diferentes zonas horarias) y lo consulta para saber la hora. RelojSat interactúa con `GPS` para calcular su posición. `WebificarReloj` actualiza los datos que están contenidos en el reloj para reflejar cambios en su política del tiempo (por ejemplo, cambios en las fechas inicial y final del tiempo de ahorro de luz de día).

Considere un ejemplo más complejo, FRIEND, un sistema de información distribuido para la administración de accidentes [FRIEND, 1994] [Bruegge *et al.*, 1994]. Éste incluye muchos actores, como `OficialCampo` que representa a los oficiales de policía y bomberos que están respondiendo a un incidente, y `Despachador`, el oficial de policía responsable de atender las llamadas 911 y despachar recursos hacia un incidente. FRIEND apoya a ambos actores llevando la cuenta de los incidentes, recursos y planes de tarea. También tiene acceso a varias bases de datos, como una base de datos de materiales peligrosos y de procedimientos de operación de emergencias. Los actores `OficialCampo` y `Despachador` interactúan mediante interfaces diferentes: `OficialCampo` tiene acceso a FRIEND mediante un ayudante personal móvil, y `Despachador` tiene acceso a FRIEND mediante una estación de trabajo (vea la figura 4-4).

Los actores son abstracción de papeles y no necesariamente tienen una correspondencia directa con personas. La misma persona puede ocupar el papel de `OficialCampo` o `Despachador` en momentos diferentes. Sin embargo, la funcionalidad a que tienen acceso es sustancialmente diferente. Por esta razón, a estos dos papeles se les modela como dos actores diferentes.

El primer paso de la obtención de requerimientos es la identificación de actores. Esto sirve para definir las fronteras del sistema y para encontrar todas las perspectivas desde las cuales los desarrolladores necesitan considerarlo. Cuando el sistema se despliega en una organización existente (como una compañía), por lo general ya existen la mayoría de los actores antes de que se desarrolle el sistema y corresponden a los papeles dentro de la organización.

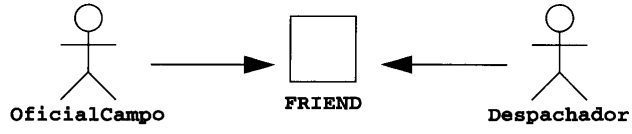


Figura 4-4 Actores del sistema FRIEND. `OficialCampo` no sólo tiene acceso a una funcionalidad diferente, sino que usa una computadora diferente para tener acceso al sistema.

Cuando se trata de identificar a los actores, los desarrolladores pueden hacer las siguientes preguntas:

Preguntas para la identificación de actores

- ¿Cuáles grupos de usuarios son apoyados por el sistema para realizar su trabajo?
- ¿Cuáles grupos de usuarios ejecutan las funciones principales del sistema?
- ¿Cuáles grupos de usuarios realizan funciones secundarias, como el mantenimiento y la administración?
- ¿Interactuará el sistema con algún sistema de hardware o software externo?

En el ejemplo de FRIEND, estas preguntas conducen a una larga lista de actores potenciales: bombero, oficial de policía, despachador, investigador, alcalde, gobernador, una base de datos de materiales peligrosos EPA, administrador del sistema, etc. Luego necesitamos consolidar esta lista en una pequeña cantidad de actores que son diferentes desde el punto de vista de su uso del sistema. Por ejemplo, un bombero y un oficial de policía pueden compartir la misma interfaz del sistema, ya que ambos están involucrados con un solo incidente en el campo. Por otro lado, un despachador administra varios incidentes concurrentes y requiere acceso a una mayor cantidad de información. Es probable que el alcalde y el gobernador no interactuarán en forma directa con el sistema sino que usarán los servicios de un operador entrenado.

Una vez identificados los actores, el siguiente paso en la actividad de obtención de requerimientos es determinar la funcionalidad a la que tiene acceso cada actor. Esta información puede extraerse usando escenarios y formalizando el uso de los casos de uso.

4.4.2 Identificación de escenarios

Un escenario es "una descripción narrativa de lo que la gente hace y experimenta cuando trata de utilizar sistemas y aplicaciones de computadora" [Carroll, 1995]. Un escenario es una descripción concreta, enfocada e informal de una sola característica del sistema desde el punto de vista de un solo actor. El uso de escenarios para la obtención de requerimientos es una desviación conceptual con respecto a las representaciones tradicionales que son genéricas y abstractas. Las representaciones tradicionales se centran en el sistema y no en el trabajo al cual da apoyo el sistema. Por último, su enfoque es la suficiencia, consistencia y precisión, mientras que los escenarios son abiertos e informales. Un enfoque basado en escenarios no puede reemplazar por completo (y no se pretende que lo haga) a los enfoques tradicionales. Sin embargo, mejora la obtención de requerimientos proporcionando una herramienta que es comprensible con facilidad para usuarios y clientes.

La figura 4-5 es un ejemplo de un escenario del sistema FRIEND [FRIEND, 1994], un sistema de información para responder a incidentes. En este escenario, un oficial de policía reporta un incendio y un `Despachador` inicia la respuesta al incidente. Observe que este escenario es concreto en el sentido de que describe una sola instancia. No trata de describir todas las situaciones posibles en las que se reporta un incidente de incendio.

Nombre del escenario	`bodegaEnLlamas`
Instancias de actores participantes	`roberto, alicia: OficialCampo` `juan: Despachador`
Flujo de eventos	1. Roberto, manejando por la calle principal en su patrulla, observa que sale humo de una bodega. Su compañera, Alicia, activa la función "Reportar emergencia" en su laptop FRIEND. 2. Alicia captura la dirección del edificio, una breve descripción de su ubicación (es decir, esquina noroeste) y un nivel de emergencia. Además de un carro de bomberos, solicita varias ambulancias, ya que el área parece estar algo atareada. Confirma lo capturado y espera el acuse de recibo. 3. Juan, el `Despachador`, es alertado de que hay una emergencia mediante un sonido de su estación de trabajo. Revisa la información enviada por Alicia y da el acuse de recibo del reporte. Asigna un carro de bomberos y dos ambulancias al lugar del `Incidente` y envía la hora estimada de llegada (ETA) a Alicia. 4. Alicia recibe el acuse de recibo y la ETA.

Figura 4-5 Escenario `bodegaEnLlamas` para el caso de uso `ReportarEmergencia`.

Los escenarios pueden tener muchos usos diferentes durante la obtención de requerimientos y durante otras actividades del ciclo de vida. A continuación se tiene una cantidad seleccionada de tipos de escenarios tomados de [Carroll, 1995]:

- Los *escenarios tal como son* describen una situación actual. Por ejemplo, durante la reingeniería se comprende al sistema actual observando a los usuarios y describiendo sus acciones como escenarios. Luego se pueden validar estos escenarios con los usuarios para ver si son correctos y precisos.
- Los *escenarios visionarios* describen un sistema futuro, ya sea que se le esté aplicando reingeniería o se esté diseñando a partir de cero. Los desarrolladores usan los escenarios como una representación de diseño conforme refinan su idea del sistema futuro y como un medio de comunicación para obtener requerimientos de los usuarios. Los escenarios visionarios pueden verse como un prototipo barato.
- Los *escenarios de evaluación* describen las tareas del usuario contra las que se va a evaluar el sistema. La colaboración por parte de usuarios y desarrolladores para la elaboración de los escenarios de evaluación también mejora la definición de la funcionalidad que se prueba mediante estos escenarios.
- Los *escenarios de entrenamiento* son cursos prácticos que se usan para introducir a los nuevos usuarios al sistema. Son instrucciones paso a paso diseñadas para llevar de la mano al usuario a través de tareas comunes.

En la obtención de requerimientos, los desarrolladores y usuarios escriben y refinan una serie de escenarios para obtener una comprensión compartida de lo que debe ser el sistema. En un principio, cada escenario puede ser de nivel alto e incompleto, como el escenario de `bodegaEnLlamas`. Se pueden usar las siguientes preguntas para identificar escenarios:

Preguntas para identificar escenarios

- ¿Cuáles son las tareas que el actor quiere que realice el sistema?
- ¿Qué información consulta el actor? ¿Quién crea esos datos? ¿Se les puede modificar o eliminar? ¿Quién lo hace?
- ¿Qué cambios externos necesita informar el actor al sistema? ¿Con cuánta frecuencia? ¿Cuándo?
- ¿Cuáles eventos necesita el actor que le informe el sistema? ¿Con cuánta latencia?

Los desarrolladores usan los documentos existentes acerca del dominio de aplicación para responder estas preguntas. Esos documentos incluyen manuales de usuario de sistemas anteriores, manuales de procedimientos, estándares de la compañía, notas y trucos del usuario y entrevistas con usuarios y clientes. Los desarrolladores siempre deben redactar los escenarios usando términos del dominio de aplicación en vez de sus propios términos. Conforme los desarrolladores obtienen una mayor visión del dominio de aplicación y las posibilidades de la tecnología disponible, refinan los escenarios en forma iterativa e incremental para incluir mayor cantidad de detalles. El trazo de maquetas de interfaz de usuario a menudo ayuda a encontrar omisiones en la especificación y ayuda a que los usuarios se formen una imagen más completa del sistema.

En el ejemplo FRIEND podemos identificar cuatro escenarios que abarcan el tipo de tareas que se espera que apoye el sistema:

- `bodegaEnLlamas` (figura 4-5): se detecta un incendio en una bodega y dos oficiales de campo llegan a la escena y solicitan recursos.
- `dobladoDeDefensas`. Sucede un accidente automovilístico sin víctimas en la autopista. Los oficiales de policía documentan el incidente y manejan el tráfico mientras son retirados los vehículos dañados.
- `gatoEnÁrbol`. Un gato queda atrapado en un árbol. Se llama a un camión de bomberos para que recupere al gato. Debido a que el incidente es de baja prioridad, el carro de bomberos se toma su tiempo para llegar a la escena. Mientras tanto, el impaciente propietario del gato se sube al árbol, se cae y se rompe una pierna, requiriendo que se envíe una ambulancia.
- `Temblor`. Un temblor sin precedentes daña seriamente edificios y carreteras, abarcando varios incidentes y disparando la activación de un plan de operaciones de emergencia a escala estatal. Se le notifica al gobernador. El daño a las carreteras impide la respuesta al incidente.

El énfasis en la identificación de actores y escenarios es para que los desarrolladores comprendan el dominio de aplicación y definan el sistema correcto. Esto da como resultado una comprensión compartida de los procesos de trabajo del usuario que necesitan ser apoyados y el alcance del sistema. Una vez que los desarrolladores identifican y describen a los actores y escenarios, formalizan los escenarios hacia casos de uso.

4.4.3 Identificación de casos de uso

Un escenario es una instancia de un caso de uso, esto es, un caso de uso especifica todos los escenarios posibles para una parte de funcionalidad dada. Un caso de uso es iniciado por un actor.

Nombre del caso de uso	`ReportarEmergencia`
Actor participante	Iniciado por `OficialCampo` Se comunica con `Despachador`
Condición inicial	1. El `OficialCampo` activa la función "Reportar Emergencia" de su terminal.
Flujo de eventos	2. FRIEND responde presentando un formulario al oficial. 3. El `OficialCampo` llena el formulario, seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia. Una vez que ha llenado el formulario, el `OficialCampo` lo envía y en ese momento se le notifica al `Despachador`. 4. El `Despachador` revisa la información enviada y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. El `Despachador` selecciona una respuesta y da un acuse de recibo del reporte de emergencia.
Condición de salida	5. El `OficialCampo` recibe el acuse de recibo y la respuesta seleccionada.
Requerimientos especiales	Se da acuse de recibo del reporte del `OficialCampo` en menos de 30 segundos. La respuesta seleccionada llega antes de que transcurran 30 segundos a partir de que la envía el `Despachador`.

Figura 4-6 Un ejemplo de un caso de uso: el caso de uso `ReportarEmergencia`.

Después de haber sido iniciado, un caso de uso también puede interactuar con otros actores. Un caso de uso representa un flujo de eventos completo a través del sistema, en el sentido de que describe una serie de interacciones relacionadas que resultan de la iniciación del caso de uso.

La figura 4-6 muestra el caso de uso `ReportarEmergencia` del cual es una instancia el escenario `bodegaEnLlamas` (vea la figura 4-5). El actor `OficialCampo` inicia este caso de uso activando la función "Reportar Emergencia" de FRIEND. El caso de uso se termina cuando el actor `OficialCampo` recibe un acuse de recibo de que se ha creado un incidente. Este caso de uso es general y comprende un rango de escenarios. Por ejemplo, el caso de uso `ReportarEmergencia` también podría aplicarse al escenario `dobladoDeDefensas`. Los casos de uso pueden escribirse con varios niveles de detalle, como sucede con los escenarios. El caso de uso `ReportarEmergencia` puede ser lo bastante ilustrativo como para describir la manera en que FRIEND apoya el reporte de emergencias y para obtener retroalimentación general del usuario, pero no proporciona detalles suficientes para una especificación del sistema.

4.4.4 Refinamiento de los casos de uso

La figura 4-7 es una versión refinada del caso de uso `ReportarEmergencia`. Ha sido ampliada para incluir detalles acerca del tipo de incidente que conoce FRIEND e interacciones detalladas que indican la manera en que el `Despachador` da el acuse de recibo al `OficialCampo`.

Ubicación	*Descripción del caso de uso*
Estación del oficial de campo	1. El `OficialCampo` activa la función "Reportar Emergencia" de su terminal. 2. FRIEND responde presentando un formulario al oficial. El formulario incluye un menú de tipos de emergencia (emergencia general, incendio, transporte), y campos para ubicación, descripción del incidente, petición de recursos y materiales peligrosos. 3. El `OficialCampo` llena el formulario, especificando en forma mínima el tipo de emergencia y los campos de descripción. El `OficialCampo` también puede describir respuestas posibles a la situación de emergencia y solicitar recursos específicos. Una vez que ha llenado el formulario, el `OficialCampo` lo envía oprimiendo el botón "Enviar Reporte" y en ese momento se le notifica al `Despachador`.
Estación del despachador	4. Al `Despachador` se le notifica un nuevo reporte de incidente mediante un cuadro de diálogo desplegable. El `Despachador` revisa la información enviada y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. Toda la información contenida en el formulario del `OficialCampo` se incluye en forma automática en el `Incidente`. El `Despachador` selecciona una respuesta asignando recursos al incidente (con el caso de uso `AsignarRecursos`) y da un acuse de recibo al reporte de emergencia enviando un mensaje breve al `OficialCampo`.
Estación del oficial de campo	5. El `OficialCampo` recibe el acuse de recibo y la respuesta seleccionada.

Figura 4-7 Descripción refinada del caso de uso `ReportarEmergencia`.

El uso de escenarios y casos de uso para definir la funcionalidad del sistema ayuda en la creación de requerimientos que son validados por el usuario al inicio del desarrollo. Conforme empiezan el diseño y la implementación del sistema se incrementa el costo de los cambios a la especificación del sistema y la adición de nuevas funcionalidades no previstas. Aunque los requerimientos cambian hasta cerca del final del desarrollo, los desarrolladores y usuarios deben esforzarse para manejar desde el principio la mayoría de los requerimientos. Esto implica muchos cambios y experimentación durante la obtención de requerimientos. Observe que muchos casos de uso se escriben varias veces, otros se refinan mucho y otros, incluso, se eliminan por completo. Para ahorrar tiempo se puede realizar mucho trabajo de exploración usando escenarios y maquetas de interfaz. Se puede usar la siguiente heurística para la escritura de escenarios y casos de uso.

Heurística para la escritura de escenarios y casos de uso

- Use escenarios para comunicarse con los usuarios y para validar la funcionalidad.
- Primero refine una rebanada vertical reducida (es decir, un escenario) para comprender el estilo de interacción preferido por el usuario.
- Luego defina una rebanada horizontal (es decir, muchos escenarios no muy detallados) para definir el alcance del sistema. Valídelos con el usuario.
- Use maquetas sólo como un apoyo visual, ya que el diseño de interfaz de usuario debe darse como una tarea separada una vez que la funcionalidad sea lo suficientemente estable.
- Presente varias alternativas al usuario (en vez de extraer una sola alternativa del usuario).
- Detalle una rebanada vertical amplia cuando el alcance del sistema y las preferencias del usuario se tengan bien comprendidas. Valídelo con el usuario.

El enfoque de esta actividad es la suficiencia y la corrección. Los desarrolladores identifican la funcionalidad que no está cubierta por los escenarios y la documentan con nuevos casos de uso. Los desarrolladores describen casos que suceden rara vez y el manejo de excepciones tal como es visto por los actores. Si los actores requieren un sistema de soporte de ayuda en línea, los desarrolladores lo describen con casos de uso durante esta actividad.

Una vez que la especificación del sistema llega a ser estable se pueden tratar los asuntos de rastreabilidad y redundancia, consolidando y reorganizando los actores y casos de uso.

4.4.5 Identificación de las relaciones entre actores y casos de uso

Aun los sistemas de tamaño medio tienen muchos casos de uso. Las relaciones entre actores y casos de uso permiten que los desarrolladores y usuarios reduzcan la complejidad del modelo e incrementen su comprensibilidad. Usamos las relaciones de comunicación entre actores y casos de uso para describir el sistema en capas de funcionalidad. Usamos relaciones extendidas para separar el flujo común de eventos del excepcional. Usamos relaciones de inclusión para reducir la redundancia entre casos de uso.

Relaciones de comunicación entre actores y casos de uso

Las relaciones de comunicación entre actores y casos de uso representan el flujo de información durante el caso de uso. Se debe distinguir entre el actor que inicia el caso de uso y los demás actores con los que se comunica el caso de uso. Por lo tanto, el control de acceso (es decir, cuál actor tiene acceso a cuál funcionalidad de clase) puede representarse a este nivel. Las relaciones entre actores y casos de uso se identifican cuando se identifican los casos de uso. La figura 4-8 muestra un ejemplo de relaciones de comunicación en el caso del sistema FRIEND.

Relaciones extendidas entre casos de uso

Un caso de uso extiende otro caso de uso si el caso de uso extendido puede incluir el comportamiento de la extensión bajo determinadas condiciones. En el ejemplo FRIEND, suponga que se corta la conexión entre la estación del `OficialCampo` y la estación del `Despachador` mientras el `OficialCampo` está llenando el formulario (por ejemplo, el automóvil del `OficialCampo` entra a un túnel). La estación del `OficialCampo` necesita comunicarle al `OficialCampo` que su formulario no fue entregado y las medidas que debe tomar. El caso de uso `ConexiónPerdida` está modelado como una extensión a `ReportarEmergencia` (vea la figura 4-9). Las condiciones bajo las cuales se

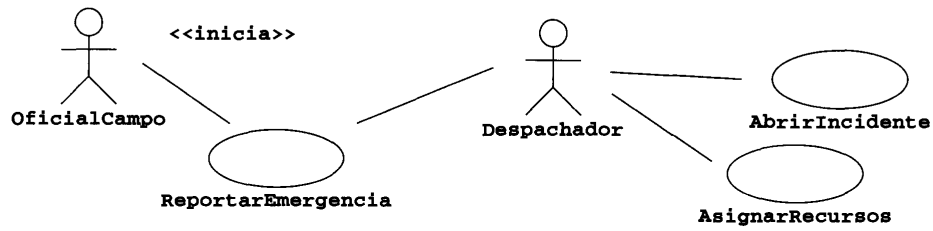


Figura 4-8 Ejemplo de relaciones de comunicación entre actores y casos de uso en FRIEND (diagrama de caso de uso UML). El `OficialCampo` inicia el caso de uso `ReportarEmergencia` y el `Despachador` inicia los casos de uso `AbrirIncidente` y `AsignarRecursos`. Los `OficialCampo` no pueden abrir en forma directa un incidente o asignar recursos por sí mismos.

inicia el caso de uso `ConexiónPerdida` se describen en `ConexiónPerdida` en vez de en `ReportarEmergencia`. La separación del flujo de eventos excepcional y opcional con respecto al caso de uso básico tiene dos ventajas. Primera, el caso de uso básico se hace más pequeño y más fácil de comprender. Segunda, se hace distinción entre el caso común y el excepcional, y esto permite que los desarrolladores traten cada tipo de funcionalidad en forma diferente (por ejemplo, optimizar el caso común en su tiempo de respuesta, optimizar el caso excepcional en su claridad). Tanto el caso de uso extendido como las extensiones son casos de uso completos por sí mismos. Deben tener una condición inicial y otra final, y ser comprensibles por el usuario como un conjunto independiente.

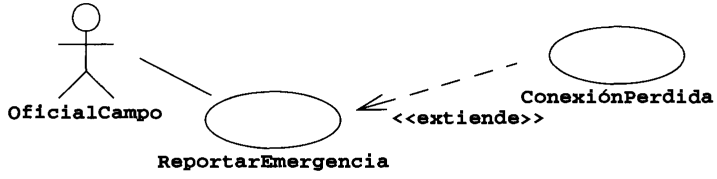


Figura 4-9 Ejemplo del uso de una relación extendida (diagrama de caso de uso UML). `ConexiónPerdida` extiende al caso de uso `ReportarEmergencia`. El caso de uso `ReportarEmergencia` se hace más corto y se enfoca sólo en reportar la emergencia.

Relaciones de inclusión entre casos de uso

Las redundancias entre casos de uso pueden factorizarse usando relaciones de inclusión. Supongamos, por ejemplo, que un `Despachador` necesita consultar el plano de la ciudad cuando abre un incidente (por ejemplo, para verificar cuáles áreas tienen riesgo durante un incendio) y cuando asigna recursos (por ejemplo, para saber cuáles recursos están más cercanos al incidente). En esta circunstancia, el caso de uso `VerPlano` describe el flujo de eventos que se requieren cuando se ve el plano de la ciudad, y es utilizado por los casos de uso `AbrirIncidente` y `AsignarRecursos` (figura 4-10).

La factorización del comportamiento compartido de casos de uso tiene muchos beneficios, incluyendo descripciones más cortas y menos redundancias. El comportamiento *sólo* deberá fac-

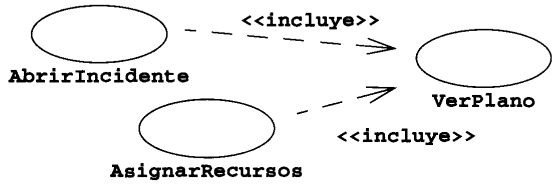


Figura 4-10 Ejemplo de relaciones de inclusión entre casos de uso. `VerPlano` describe el flujo de eventos para ver un mapa de la ciudad (por ejemplo, desplazamiento, acercamiento, consulta por nombre de calle) y es utilizado por los casos de uso `AbrirIncidente` y `AsignarRecursos`.

torizarse en casos de uso separados si es compartido entre dos o más casos de uso. La fragmentación excesiva de la especificación del sistema a través de una gran cantidad de casos de uso hace que la especificación sea confusa para los usuarios y clientes.

Relaciones extendidas frente a inclusión

Las construcciones de inclusión y extendidas son similares, y al principio puede ser que no le quede claro al desarrollador cuándo hay que usar cada una de ellas [Jacobson *et al.*, 1992]. La principal distinción entre estas construcciones es la dirección de la relación. En el caso de una relación de inclusión, las condiciones bajo las cuales se inicia el caso de uso están descritas en el caso de uso iniciador como un evento en el flujo de eventos. En el caso de una relación extendida, las condiciones bajo las cuales se inicia la extensión están descritas en la extensión como una condición inicial. La figura 4-11 muestra el ejemplo `ConexiónPerdida` descrito con una relación de inclusión (columna izquierda) y con una relación extendida (columna derecha). En la columna izquierda necesitamos insertar texto en dos lugares del flujo de eventos en donde puede llamarse al caso de uso `ConexiónPerdida`. También, si se describen situaciones excepcionales adicionales (por ejemplo, una función Ayuda en la estación `OficialCampo`) el caso de uso `ReportarEmergencia` tendrá que modificarse y llegará a atiborrarse con condiciones. En la columna derecha sólo necesitamos describir las condiciones bajo las cuales se llama al caso de uso. Además, se pueden añadir situaciones excepcionales adicionales sin modificar el caso de uso básico (por ejemplo, `ReportarEmergencia`). La habilidad para extender el sistema sin modificar las partes existentes es crítica, ya que nos permite asegurarnos que el comportamiento original queda intacto.

En resumen, se puede usar la siguiente heurística para seleccionar una relación extendida o de inclusión.

Heurística para las relaciones extendidas y de inclusión

- Use relaciones extendidas para comportamientos excepcionales, opcionales o que rara vez suceden.
- Use relaciones de inclusión para comportamientos que se comparten entre dos o más casos de uso.

`ReportarEmergencia` (relación de inclusión)	`ReportarEmergencia` (relación extendida)
1. ... 2. ... 3. El `OficialCampo` llena el formulario seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia. Una vez que está lleno el formulario, el `OficialCampo` lo envía y en ese momento se le notifica al `Despachador`. *Si se perdió la conexión con el Despachador se usa el caso de uso ConexiónPerdida.* 4. Si la conexión existe, el `Despachador` revisa la información enviada y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. El `Despachador` selecciona una respuesta y da un acuse de recibo al reporte de emergencia. *Si se perdió la conexión se usa el caso de uso ConexiónPerdida.* 5. ...	1. ... 2. ... 3. El `OficialCampo` llena el formulario seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia. Una vez que está lleno el formulario, el `OficialCampo` lo envía y en ese momento se le notifica al `Despachador`. 4. El `Despachador` revisa la información enviada y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. El `Despachador` selecciona una respuesta y da un acuse de recibo al reporte de emergencia. 5. ...
`ConexiónPerdida` (relación de inclusión)	`ConexiónPerdida` (relación extendida)
1. Se le notifica al `OficialCampo` y al `Despachador` que la conexión se ha perdido. Se les avisa de las posibles razones por las cuales pudo ocurrir el evento (por ejemplo, "¿está en un túnel la estación del `OficialCampo`?"). 2. Se registra la situación en el sistema y se recupera cuando se vuelve a establecer la conexión. 3. El `OficialCampo` y el `Despachador` se ponen en contacto por otros medios y el `Despachador` inicia `ReportarEmergencia` desde la estación del `Despachador`.	*El caso de uso ConexiónPerdida extiende a ReportarEmergencia cuando se pierde la conexión entre el OficialCampo y el Despachador.* 1. Se le notifica al `OficialCampo` y al `Despachador` que la conexión se ha perdido. Se les avisa de las posibles razones por las cuales pudo ocurrir el evento (por ejemplo, "¿está en un túnel la estación del `OficialCampo`?"). 2. Se registra la situación en el sistema y se recupera cuando se vuelve a establecer la conexión. 3. El `OficialCampo` y el `Despachador` se ponen en contacto por otros medios y el `Despachador` inicia `ReportarEmergencia` desde la estación del `Despachador`.

Figura 4-11 Adición de la condición excepcional `ConexiónPerdida` a `ReportarEmergencia`. Se usa una relación extendida para el flujo de eventos excepcional y opcional, ya que produce una descripción más modular.

En todos los casos el propósito de la adición de relaciones de inclusión y extendidas es reducir o eliminar redundancias del modelo del caso de uso, eliminando, por lo tanto, inconsistencias potenciales.

4.4.6 Identificación inicial de los objetos de análisis

Uno de los primeros obstáculos que encuentran los desarrolladores y los usuarios cuando colaboran es la terminología diferente. Se produce una falta de comprensión por usar los mismos términos en contextos diferentes y con diferente significado. Aunque los desarrolladores con el tiempo aprenden la terminología de los usuarios, es probable que vuelvan a encontrar este problema cuando añadan nuevos desarrolladores al proyecto.

Una vez que se han consolidado los casos de uso, los desarrolladores identifican los **objetos participantes** en cada caso de uso. Los objetos participantes corresponden a los conceptos principales del dominio de aplicación. Los desarrolladores los identifican, nombran y describen sin ambigüedad, y los reúnen en un glosario.

Este glosario se incluye en la identificación del sistema y más adelante en los manuales de usuario. Los desarrolladores mantienen actualizado este glosario conforme evoluciona la especificación del sistema. Son muchos los beneficios del glosario: los nuevos desarrolladores quedan expuestos a un conjunto de definiciones consistentes, un solo término se usa para cada contexto (en vez de un término del desarrollador y un término del usuario) y cada término tiene un significado oficial preciso y claro.

La identificación de los objetos participantes da como resultado el modelo de análisis inicial. La identificación de los objetos participantes durante la obtención de requerimientos constituye solamente un primer paso hacia el modelo de análisis completo. El modelo de análisis completo no se usa, por lo general, como medio de comunicación entre usuarios y desarrolladores, ya que con frecuencia los usuarios no están familiarizados con los conceptos orientados a objetos. Sin embargo, la descripción de los objetos (es decir, la definición de los términos en el glosario) y sus atributos son visibles para los usuarios y se revisan. En el capítulo 5, *Análisis*, describimos con más detalle los refinamientos posteriores del modelo de análisis.

En la literatura se han propuesto muchas heurísticas para la identificación de objetos. Éstas son unas cuantas seleccionadas:

Heurísticas para la identificación inicial de los objetos de análisis

- Términos que los desarrolladores o los usuarios necesitan aclarar para comprender el caso de uso.
- Nombres recurrentes en los casos de uso (por ejemplo, `Incidente`).
- Entidades del mundo real de las que el sistema necesita llevar cuenta (por ejemplo, `OficialCampo`, `Recurso`).
- Procesos del mundo real de los que el sistema necesita llevar cuenta (por ejemplo, `PlanOperacionesEmergencia`).
- Casos de uso (por ejemplo, `ReportarEmergencia`).
- Orígenes o destinos de datos (por ejemplo, `Impresora`).
- Artefactos de interfaz (por ejemplo, `Estación`).
- *Siempre* hay que usar términos del dominio de aplicación.

Durante la obtención de requerimientos se generan objetos participantes para cada caso de uso. Si dos casos de uso se refieren al mismo concepto, el objeto correspondiente deberá ser el mismo. Si dos objetos comparten el mismo nombre y no corresponden al mismo concepto, se le cambia el nombre a uno de los conceptos o a ambos para reconocer y enfatizar su diferencia. Esta consolidación elimina cualquier ambigüedad en la terminología usada. Por ejemplo, la tabla 4-2 muestra los objetos participantes iniciales que identificamos para el caso de uso `ReportarEmergencia`.

Tabla 4-2 Objetos participantes en el caso de uso `ReportarEmergencia`.

`Despachador`	Es un oficial de policía que administra `Incidentes`. Un `Despachador`, abre, documenta y cierra incidentes en respuesta a reportes de emergencia y otras comunicaciones con los `OficialCampo`. Los `Despachador` se identifican mediante sus números de identificación.
`ReporteDeEmergencia`	Es el reporte inicial acerca de un `Incidente` que hace un `OficialCampo` hacia un `Despachador`. Por lo general, un `ReporteDeEmergencia` activa la creación de un `Incidente` por parte del `Despachador`. Un `ReporteDeEmergencia` está compuesto de un nivel de emergencia, un tipo (incendio, accidente de carretera u otro), una ubicación y una descripción.
`OficialCampo`	Es un oficial de policía o bombero que está trabajando. Un `OficialCampo` puede asignarse, a lo mucho, a un `Incidente` a la vez. Los `OficialCampo` se identifican mediante sus números de identificación.
`Incidente`	Es una situación que requiere atención de un `OficialCampo`. Un `Incidente` puede ser reportado al sistema por un `OficialCampo` o por alguna persona ajena al sistema. Un `Incidente` está compuesto por una descripción, una respuesta, un estado (abierto, cerrado, documentado), una ubicación y una cantidad de `OficialCampo`.

Una vez que los objetos participantes están identificados y consolidados, los desarrolladores pueden usarlos como lista de verificación para asegurarse que esté completo el conjunto de casos de uso identificados.

Heurística para la revisión cruzada de casos de uso y objetos participantes

- ¿Cuál caso de uso crea a este objeto (es decir, durante cuáles casos de uso se dan al sistema los valores de los atributos del objeto)? ¿Cuáles actores pueden tener acceso a esta información?
- ¿Cuáles casos de uso modifican y destruyen este objeto (es decir, cuáles casos de uso editan o eliminan esta información del sistema)? ¿Cuál actor puede iniciar estos casos de uso?
- ¿Es necesario este objeto?, es decir, ¿hay al menos un caso de uso que depende de esta información?

Cuando los desarrolladores identifican nuevos casos de uso, describen e integran el nuevo caso de uso en el modelo siguiendo el proceso descrito antes. Con frecuencia, en la actividad de obtención de requerimientos, los cambios de perspectiva introducen modificaciones en la especificación del sistema (por ejemplo, la localización de nuevos objetos participantes activa la adición de nuevos casos de uso, y la adición de nuevos casos de uso activa la adición o refinamiento de los nuevos objetos participantes). Se debe tener prevista esta inestabilidad y motivar el cambio de perspectivas. Por la misma razón, las tareas que se llevan mucho tiempo, como la descripción de casos excepcionales y el refinamiento de interfaces de usuario, se deben posponer hasta que el conjunto de casos de uso sea estable.

4.4.7 Identificación de requerimientos no funcionales

Los requerimientos no funcionales describen aspectos del sistema visibles para el usuario que no están relacionados en forma directa con el comportamiento funcional del sistema. Los requerimientos no funcionales abarcan varios asuntos, desde la apariencia de la interfaz de usuario hasta

los requerimientos de tiempo de respuesta y las cuestiones de seguridad. Los requerimientos no funcionales se definen al mismo tiempo que los funcionales, debido a que tienen mucho impacto en el desarrollo y costo del sistema.

Por ejemplo, considere una pantalla en mosaico que usa un controlador de tráfico aéreo para el seguimiento de los aviones. Un sistema de pantalla en mosaico recopila datos de una serie de radares y bases de datos (por eso el término "mosaico") en una pantalla de resumen que indica todas las aeronaves que están en determinada área, incluyendo su identificación, velocidad y altitud. La cantidad de aeronaves que puede mostrar tal sistema restringe el desempeño del controlador de tráfico aéreo y el costo del sistema. Si el sistema sólo puede manejar unas cuantas aeronaves en forma simultánea, no podrá usarse en aeropuertos con mucho tráfico. Por otro lado, es más costoso y complejo construir un sistema capaz de manejar una gran cantidad de aeronaves.

Los requerimientos no funcionales pueden obtenerse investigando los siguientes temas:

- **Interfaz de usuario y factores humanos.** ¿Qué tipo de interfaz debe proporcionar el sistema? ¿Cuál es el nivel de experiencia de los usuarios?
- **Documentación.** ¿Qué nivel de documentos se requiere? ¿Sólo deberá proporcionarse documentación para los usuarios? ¿Deberá haber documentación técnica para mantenimiento? ¿Deberá documentarse el proceso de desarrollo?
- **Consideraciones de hardware.** ¿Hay requerimientos de compatibilidad de hardware? ¿Interactuará el sistema con otros sistemas de hardware?
- **Características de desempeño.** ¿Qué tan sensible debe ser el sistema? ¿Qué tantos usuarios concurrentes debe soportar? ¿Cuál es la carga típica o extrema?
- **Manejo de errores y condiciones extremas.** ¿Cómo debe manejar el sistema las excepciones? ¿Qué excepciones debe manejar el sistema? ¿Cuál es el peor ambiente en que se espera que se desempeñe el sistema? ¿Hay requerimientos de seguridad en el sistema?
- **Asuntos de calidad.** ¿Qué tan confiable, disponible y robusto debe ser el sistema? ¿Cuál es la participación del cliente en la valoración de la calidad del sistema o en el proceso de desarrollo?
- **Modificaciones al sistema.** ¿Cuál es el alcance previsto de cambios futuros? ¿Quién realizará los cambios?
- **Ambiente físico.** ¿Cuándo se entregará el sistema? ¿Hay factores externos, como las condiciones climáticas, que debe resistir el sistema?
- **Cuestiones de seguridad.** ¿El sistema deberá estar protegido contra intrusiones externas o usuarios malintencionados? ¿En qué nivel?
- **Cuestiones de recursos.** ¿Cuáles son las restricciones en los recursos consumidos por el sistema?

Una vez que se han identificado y descrito todos los requerimientos no funcionales se les asigna prioridad de acuerdo con su importancia. Aunque la mayoría de los requerimientos no funcionales son muy deseables, algunos de ellos necesitan satisfacerse para que el sistema opere en forma correcta.

4.5 Administración de la obtención de requerimientos

En la sección anterior describimos las cuestiones técnicas del modelado de un sistema desde el punto de vista de los casos de uso. Sin embargo, el modelado de casos de uso no constituye, por sí mismo, la obtención de requerimientos. Aun después de que se convierten en expertos modeladores de casos de uso, los desarrolladores todavía necesitan obtener requerimientos de los usuarios y llegar a un acuerdo con el cliente. En esta sección describimos métodos para la obtención de información de los usuarios y la negociación de un acuerdo con un cliente. En particular describimos:

- La obtención de requerimientos a partir de los usuarios: conocimiento del análisis de tareas (KAT) (sección 4.5.1).
- Negociación de una especificación con los clientes: diseño conjunto de aplicaciones (JAD) (sección 4.5.2).
- Validación de requerimientos: pruebas de utilidad (sección 4.5.3).
- Documentación de la obtención de requerimientos (sección 4.5.4).

4.5.1 Obtención de información de los usuarios: conocimiento del análisis de tareas

El análisis de tareas se originó en Estados Unidos y en el Reino Unido en los años cincuenta y sesenta [Johnson, 1992]. En un principio el análisis de tareas no se preocupaba por los requerimientos o el diseño de sistemas. El análisis de tareas se usó para identificar la manera en que debía entrenarse a las personas. En Estados Unidos los militares se interesaron en el análisis de tareas sobre todo para disminuir el costo del entrenamiento. En el Reino Unido el Departamento de Comercio e Industria se interesó en el análisis de tareas para desarrollar métodos que permitieran a las personas moverse entre industrias. Más recientemente, el análisis de tareas ha llegado a ser importante en el campo de la interacción entre humanos y computadoras (HCI, por sus siglas en inglés) para identificar y describir las tareas del usuario que debe apoyar el sistema.

El análisis de tareas se basa en la suposición de que es poco eficiente pedir a los usuarios que describan lo que hacen y la manera en que lo hacen. Los usuarios, por lo general, no piensan en forma explícita en la secuencia de tareas que se requieren para realizar su trabajo, ya que con frecuencia repiten estas tareas muchas veces. Cuando se pregunta a los usuarios la manera en que realizan su trabajo, describen, en el mejor de los casos, la manera en que se supone que lo realizan, y esto puede estar muy lejano de la realidad. En consecuencia, el análisis de tareas usa la observación como una alternativa para construir un modelo de tarea inicial. Este modelo de tarea inicial se refina luego preguntándole a los usuarios *por qué* realizan una tarea de determinada forma.

El **conocimiento del análisis de tareas** (KAT, por sus siglas en inglés) es un método de análisis de tareas propuesto por Johnson [Johnson, 1992]. Se interesa en la recolección de datos a partir de una diversidad de fuentes (por ejemplo, análisis de protocolo, procedimientos estándar, libros de texto, entrevistas), analizándolos para identificar elementos individuales involucrados en la tarea (por ejemplo, objetos, acciones, procedimientos, objetivos y subobjetivos) y construye un modelo del conocimiento general que usa la gente para realizar las tareas que le interesan. El KAT es una técnica de análisis orientado a objetos, ya que representa el dominio de aplicación desde el punto de vista de objetos y acciones.

Se puede resumir al KAT con los cinco pasos siguientes:

1. **Identificación de objetos y acciones.** Se identifican los objetos y acciones asociadas con los objetos usando técnicas similares a las de la identificación de objetos en el análisis

orientado a objetos, como el análisis de libros de texto, manuales, reglamentos, reportes, entrevistas con quien realiza la tarea y la observación de quien realiza la tarea.

2. **Identificación de procedimientos.** Un procedimiento es un conjunto de acciones, una condición previa necesaria para activar el procedimiento y una condición posterior. Las acciones pueden estar ordenadas en forma parcial. Los procedimientos se identifican mediante la redacción de escenarios, la observación de quien realiza la tarea y pidiéndole a quien realiza la tarea que seleccione y ordene tarjetas en las que se escriben acciones individuales.
3. **Identificación de objetivos y subobjetivos.** Un objetivo es un estado a lograr para que la tarea sea satisfactoria. Los objetivos se identifican mediante entrevistas durante la realización de una tarea o después de ella. Los subobjetivos se identifican descomponiendo los objetivos.
4. **Identificación del carácter típico y la importancia.** Cada elemento identificado se califica de acuerdo con la frecuencia con que se le encuentra y si es necesario para la realización de un objetivo.
5. **Construcción de un modelo de la tarea.** La información recopilada antes se generaliza para tomar en cuenta las características comunes que hay entre las tareas. Se relacionan los objetivos, procedimientos y objetos correspondientes usando una notación textual o una gráfica. Por último se valida el modelo con quien realiza la tarea.

Aunque el análisis de tareas y el KAT no son métodos para la obtención de requerimientos por sí mismos (no producen una descripción del sistema de software futuro), pueden beneficiar en gran medida la actividad de obtención de requerimientos en varias formas:

- Durante la obtención proporcionan técnicas para obtener y describir el conocimiento del dominio de aplicación, incluyendo información como el carácter típico y la importancia de acciones específicas, y el resultado final es comprensible para quien realiza la tarea.
- Cuando se definen las fronteras de un sistema, los modelos de tarea ayudan a determinar cuáles partes de la tarea deben seguir siendo manuales y cuáles deben automatizarse; además, el modelo de tarea puede revelar áreas problemáticas en el sistema actual.
- Cuando se diseña la interfaz del sistema los modelos de tareas sirven como una fuente de inspiración para metáforas comprensibles por parte del usuario [Nielsen y Mack, 1994].

Para mayor información sobre el KAT, el lector deberá consultar literatura especializada [Johnson, 1992].

4.5.2 Negociación de especificaciones con los clientes: diseño conjunto de aplicaciones

El **diseño conjunto de aplicaciones** (JAD, por sus siglas en inglés) es un método de requerimientos desarrollado por IBM a finales de los setenta. Su efectividad radica en que el trabajo de obtención de requerimientos se realiza en una sola sesión de trabajo en la que participan todos los involucrados. Usuarios, clientes, desarrolladores y un líder de sesión entrenado se sientan juntos en un salón para presentar sus puntos de vista, escuchar los puntos de vista de los demás, negociar y ponerse de acuerdo en una solución mutuamente aceptable. El resultado de la sesión de trabajo, el documento JAD final, es un documento de especificación de sistema completo que incluye definiciones de elementos de datos, flujos de trabajo y pantallas de interfaz. Debido a que el

documento final es desarrollado en forma conjunta por todos los interesados (es decir, los participantes que no sólo tienen un interés en el éxito del proyecto sino que también pueden tomar decisiones sustanciales), el documento JAD final representa un acuerdo entre usuarios, clientes y desarrolladores y, por lo tanto, minimiza los cambios de requerimientos posteriores en el proceso de desarrollo.

El JAD está compuesto de cinco actividades (resumidas en la figura 4-12):

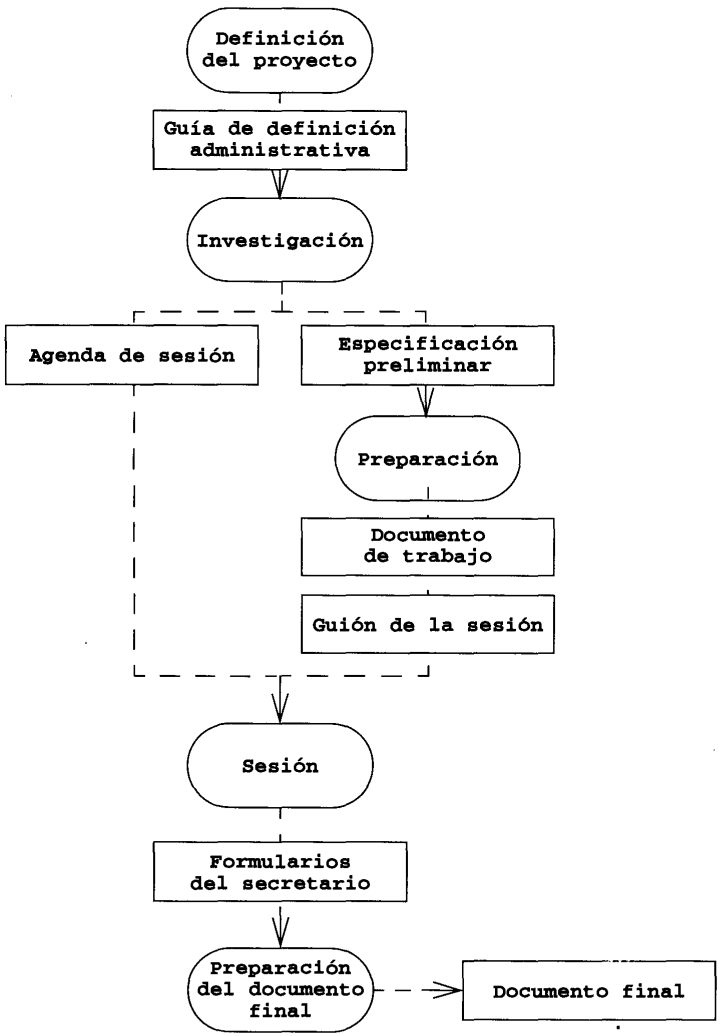


Figura 4-12 Actividades del JAD (diagrama de actividad UML). El corazón del JAD es la actividad `Sesión` durante la cual todos los interesados diseñan y se ponen de acuerdo en una especificación del sistema. Las actividades anteriores a la `Sesión` incrementan al máximo su eficiencia. La producción del documento final asegura que se capturen las decisiones tomadas durante la `Sesión`.

1. **Definición del proyecto.** Durante esta actividad el coordinador JAD entrevista a gerentes y clientes para determinar los objetivos y el alcance del proyecto. Lo que encuentra en las entrevistas se recopila en la *guía de definición administrativa.* Durante esta actividad, el coordinador JAD forma un equipo compuesto por usuarios, clientes y desarrolladores. Están representados todos los interesados, y los participantes tienen la capacidad de tomar decisiones conjuntas.
2. **Investigación.** Durante esta actividad el coordinador JAD entrevista a los usuarios presentes y futuros, recopila información del dominio y describe los flujos de trabajo. El coordinador JAD también inicia una lista de asuntos que necesitarán tratarse durante la reunión. Los resultados principales de la actividad de investigación son una *agenda de sesión* y una *especificación preliminar* que lista el flujo de trabajo y la información del sistema.
3. **Preparación.** Durante esta actividad el coordinador JAD prepara la sesión. El coordinador JAD crea un *documento de trabajo*, primer borrador del documento final, una agenda para la sesión y cualquier cantidad de filminas o gráficas que representan la información recopilada durante la actividad de investigación.
4. **Sesión.** Durante esta actividad el coordinador JAD guía al equipo para la creación de la especificación del sistema. Una sesión JAD dura de tres a cinco días. El equipo define y se pone de acuerdo en el flujo de trabajo, los elementos de datos, las pantallas y los reportes del sistema. Todas las decisiones se documentan mediante un escribano que llena formularios JAD.
5. **Documento final.** El coordinador JAD prepara el *documento final* revisando el documento de trabajo para que incluya todas las decisiones tomadas durante la sesión. El documento final representa una especificación completa del sistema acordada durante la sesión. El documento final se distribuye a los participantes en la sesión para que lo revisen. Luego los participantes se reúnen durante una a dos horas para discutir las revisiones y finalizar el documento.

El JAD ha sido usado en forma satisfactoria por IBM y otras compañías. El JAD eleva la dinámica de grupo para mejorar la comunicación entre participantes y para acelerar el consenso. Al final de una sesión JAD, los desarrolladores tienen mayor conocimiento de las necesidades de los usuarios y los usuarios tienen más conocimiento de los compromisos del desarrollo. Resultan ganancias adicionales de la reducción de actividades de rediseño a lo largo del desarrollo. Debido a que se apoya en la dinámica social, el éxito de una sesión JAD depende a menudo de las habilidades del coordinador JAD como facilitador en la reunión.

4.5.3 Validación de requerimientos: prueba de utilidad

Las **pruebas de utilidad** examinan la comprensión que tiene el usuario del modelo de los casos de uso. Las pruebas de utilidad encuentran problemas en la especificación del sistema permitiendo que los usuarios exploren el sistema o sólo parte de él (por ejemplo, la interfaz de usuario). Las pruebas de utilidad también se interesan en los detalles de la interfaz de usuario, como la apariencia de la interfaz de usuario, la disposición geométrica de las pantallas y el hardware. Por ejemplo, en el caso de una computadora que se lleva en el cuerpo, una prueba de utilidad podría comprobar la habilidad del usuario para darle comandos al sistema mientras se encuentra en una posición difícil, como el caso de un mecánico viendo una pantalla debajo de un automóvil mientras revisa el silenciador del escape.

Las pruebas de utilidad se basan en experimentos que identifican deficiencias que los desarrolladores corrigen en forma iterativa [Rubin, 1994]. La técnica para la realización de las pruebas de utilidad se basa en el enfoque clásico para la realización de un experimento controlado. Los desarrolladores formulan un objetivo de la prueba y luego lo verifican manipulando parámetros experimentales seleccionados bajo condiciones controladas. Los desarrolladores estudian en forma minuciosa los valores de estos parámetros para identificar las relaciones de causa y efecto estadísticamente significativas. Aunque este tipo de enfoque podría usarse para cualquier parámetro, las pruebas de utilidad se enfocan en parámetros de utilidad, como la facilidad para el aprendizaje del sistema, el tiempo para realizar una tarea o la tasa de errores que comete un usuario cuando realiza una tarea.

Hay dos diferencias importantes entre los experimentos clásicos y las pruebas de utilidad. Mientras que el método experimental clásico está diseñado para confirmar o refutar una hipótesis, el objetivo de las pruebas de utilidad es obtener información cualitativa sobre la manera de resolver problemas de utilidad y la manera de mejorar el sistema. Otra diferencia es el rigor con que necesitan realizarse los experimentos. Se ha mostrado que es de mucha ayuda incluso una serie de pruebas enfocadas rápidas que se inician desde la obtención de requerimientos. Nielsen usa el término "ingeniería de utilidad descontada" para indicar que unas cuantas pruebas de utilidad son mejor que ninguna [Nielsen, 1993].

Hay tres tipos de pruebas de utilidad:

- **Prueba de escenario.** Durante esta prueba se presenta un escenario visionario del sistema ante uno o más usuarios. Los desarrolladores identifican qué tan rápido pueden comprender los usuarios el escenario, con cuánta precisión representa su modelo de trabajo y qué tan positivamente reaccionan a la descripción del nuevo sistema. Los escenarios seleccionados deberán ser lo más realistas y detallados posible. Una prueba de escenario permite una retroalimentación rápida y frecuente del usuario. Las pruebas de escenario pueden realizarse como maquetas en papel[2] o con un ambiente prototipo simple, el cual con frecuencia es más fácil de aprender que el ambiente de programación que se usa para el desarrollo. La ventaja de las pruebas de escenario es que es barato realizarlas y repetirlas. La desventaja es que el usuario no puede interactuar en forma directa con el sistema y que los datos son fijos.
- **Prueba de prototipo.** Durante este tipo de prueba se presenta ante uno o más usuarios un fragmento de software que prácticamente implementa al sistema. Un prototipo vertical implementa un caso de uso completo a lo largo del sistema y un prototipo horizontal presenta una interfaz para la mayoría de los casos de uso (proporcionando poca o ninguna funcionalidad). Las ventajas de las pruebas de prototipo consisten en que proporcionan una vista realista del sistema ante el usuario y que se pueden implementar los prototipos para recolectar datos detallados. Las desventajas de los prototipos son los altos costos de producirlos y modificarlos.
- **Prueba de producto.** Esta prueba es similar a la de prototipo, a excepción de que se usa una versión funcional del sistema en vez del prototipo. Una prueba de producto sólo puede

2. El uso del guión sinóptico, una técnica de la industria de los dibujos animados, consiste en bocetar una secuencia de imágenes de la pantalla en diferentes puntos del escenario. Luego se ordenan las imágenes de cada escenario en forma cronológica en un tablero colocado en la pared. Los desarrolladores y usuarios pueden desplazarse por el salón mientras revisan y discuten los escenarios. Si se tiene un salón de buen tamaño, los participantes pueden ver varios cientos de bocetos.

realizarse una vez que ya se ha desarrollado la mayor parte del sistema. También requiere que el sistema sea fácil de modificar para que los resultados de la prueba de utilidad puedan tomarse en cuenta.

En los tres tipos de pruebas, los elementos básicos para la prueba de utilidad incluyen [Rubin, 1994]:

- Desarrollo de los objetivos de la prueba.
- Uso de una muestra representativa de usuarios finales.
- Uso del sistema en el ambiente de trabajo actual o simulado.
- Participación de los usuarios finales.
- Interrogatorio extenso y controlado, y prueba de los usuarios por la persona que realiza la prueba de utilidad.
- Recopilación y análisis de resultados cuantitativos y cualitativos.
- Recomendaciones sobre la manera de mejorar el sistema.

Los objetivos típicos de la prueba en una prueba de utilidad abordan la comparación de dos estilos de interacción del usuario, la identificación de la mejor y la peor características en un escenario o prototipo, los obstáculos principales, la identificación de características útiles para usuarios novatos y expertos, cuándo se necesita la ayuda y el tipo de información de entrenamiento que se requiere.

Uno de los principales problemas de las pruebas de utilidad es conseguir a los participantes. Hay varios obstáculos que enfrentan los gerentes de proyecto para la selección de usuarios finales reales [Grudin, 1990]:

- El gerente del proyecto, por lo general, tiene miedo de que los clientes hagan a un lado a las organizaciones de soporte técnico establecidas y llamen directamente a los desarrolladores una vez que saben la manera de llegar a ellos. Una vez que se establece esta línea de comunicación puede ser que los desarrolladores sean distraídos con mucha frecuencia de sus trabajos asignados.
- El personal de ventas no quiere que los desarrolladores hablen con sus "clientes". Los vendedores tienen miedo de que puedan ofender al cliente o de crear insatisfacción con la generación actual de productos (que todavía deben vender).
- Los usuarios finales no tienen tiempo.
- A los usuarios finales no les gusta que los estudien. Por ejemplo, un mecánico de automóviles puede pensar que un sistema que comprenda más aspectos de la realidad lo dejará sin trabajo.

Interrogar a los participantes es la clave para llegar a una comprensión sobre la manera de mejorar la utilidad del sistema que se está probando. Aunque las pruebas de utilidad descubren y exponen problemas, con frecuencia es el interrogatorio el que ilustra, en primer lugar, por qué han sucedido estos problemas. Es importante escribir las recomendaciones sobre la manera de mejorar los componentes probados lo más pronto posible después de que termina la prueba de utilidad para que puedan usarlas los desarrolladores a fin de poner en práctica cualquier cambio necesario en los modelos del sistema del componente probado.

Para mayores detalles sobre las pruebas de utilidad, el lector deberá acudir a la literatura especializada [Rubin, 1994], [Nielsen y Mack, 1994].

4.5.4 Documentación de la obtención de requerimientos

Los resultados de la actividad de obtención de requerimientos y la actividad de análisis se documentan en el documento de análisis de requerimientos (RAD, por sus siglas en inglés). Este documento describe por completo al sistema desde el punto de vista de los requerimientos funcionales y no funcionales, y sirve como una base contractual entre el cliente y los desarrolladores. La audiencia del RAD incluye al cliente, los usuarios, los administradores del proyecto, los analistas del sistema (es decir, los desarrolladores que participan en los requerimientos) y los diseñadores del sistema (es decir, los desarrolladores que participan en el diseño del sistema). La primera parte del documento, incluyendo los casos de uso y los requerimientos no funcionales, se escribe durante la obtención de requerimientos. La formalización de la especificación en términos de modelos de objetos se escribe durante el análisis. La siguiente es una plantilla de ejemplo para un RAD:

Documento de análisis de requerimientos

1. Introducción
   - 1.1 Propósito del sistema
   - 1.2 Alcance del sistema
   - 1.3 Objetivos y criterios de éxito del proyecto
   - 1.4 Definiciones, siglas y abreviaturas
   - 1.5 Referencias
   - 1.6 Panorama
2. Sistema actual
3. Sistema propuesto
   - 3.1 Panorama
   - 3.2 Requerimientos funcionales
   - 3.3 Requerimientos no funcionales
     - 3.3.1 Interfaz de usuario y factores humanos
     - 3.3.2 Documentación
     - 3.3.3 Consideraciones de hardware
     - 3.3.4 Características de desempeño
     - 3.3.5 Manejo de errores y condiciones extremas
     - 3.3.6 Cuestiones de calidad
     - 3.3.7 Modificaciones al sistema
     - 3.3.8 Ambiente físico
     - 3.3.9 Cuestiones de seguridad
     - 3.3.10 Cuestiones de recursos
   - 3.4 Seudorrequerimientos
   - 3.5 Modelos del sistema
     - 3.5.1 Escenarios
     - 3.5.2 Modelo de caso de uso
     - 3.5.3 Modelo de objetos
       - 3.5.3.1 Diccionario de datos
       - 3.5.3.2 Diagramas de clase
     - 3.5.4 Modelos dinámicos
     - 3.5.5 Interfaz de usuario: rutas de navegación y maquetas de pantallas
4. Glosario

La primera sección del RAD es una *Introducción*. Su propósito es proporcionar un panorama breve de las funciones del sistema y las razones para su desarrollo, de su alcance y de las referencias al contexto de desarrollo (por ejemplo, enunciado de problemas relacionados, referencias a sistemas existentes, estudios de factibilidad). La introducción también incluye los objetivos y criterios para el éxito del proyecto.

La segunda sección, *Sistema actual*, describe el estado actual de las cosas. Si el nuevo sistema reemplazará a uno existente, esta sección describe la funcionalidad y los problemas del sistema actual. Si no es así, esta sección describe cómo se realizan ahora las tareas soportadas por el nuevo sistema. Por ejemplo, en el caso de RelojSat, en la actualidad el usuario ajusta su reloj cada vez que pasa por una zona horaria. Debido a la naturaleza manual de esta operación, el usuario de vez en cuando pone la hora equivocada. Por el contrario, el RelojSat asegura en forma continua un tiempo preciso durante vida útil. En el caso de FRIEND, el sistema actual se basa en papel: los despachadores llevan la cuenta de la asignación de recursos llenando formularios. La comunicación entre los despachadores y los oficiales de campo se hace por radio. El sistema actual requiere altos costos de documentación y administración que pretende reducir el sistema FRIEND.

La tercera sección, *Sistema propuesto*, documenta la obtención de requerimientos y el modelo de análisis del nuevo sistema. Se divide en cinco subsecciones:

- *Panorama* presenta una vista funcional del sistema.
- *Requerimientos funcionales* describe, en lenguaje natural, la funcionalidad de alto nivel del sistema.
- *Requerimientos no funcionales* describe los requerimientos en el nivel de usuario que no están relacionados en forma directa con la funcionalidad. Esto incluye el desempeño, la seguridad, la modificabilidad, el manejo de errores, la plataforma de hardware y el ambiente físico.
- *Seudorrequerimientos* describe la restricciones de diseño e implementación impuestas por el cliente. Esto incluye la especificación de la plataforma de entrega, el lenguaje de implementación o el sistema de administración de base de datos.
- *Modelos del sistema* describe los escenarios, casos de uso, modelo de objetos y modelos dinámicos que describen al sistema. Esta sección contiene la especificación funcional completa del sistema, incluyendo maquetas y gráficas de navegación que ilustran la interfaz de usuario del sistema. Esta sección se escribe durante la actividad de *Análisis* que se describe en el siguiente capítulo.

El RAD deberá escribirse después de que sea estable el modelo de caso de uso, esto es, cuando la cantidad de modificaciones a los requerimientos sea mínima. Sin embargo, el RAD se actualiza a lo largo del proceso de desarrollo cuando se descubren problemas de especificaciones o cuando cambia el alcance del sistema. El RAD, una vez que se publica, es tomado como la línea base y se pone bajo la administración de configuración. La sección de historia de revisiones del RAD proporciona una historia de los cambios en forma de una lista de los autores responsables de los cambios, la fecha del cambio y una breve descripción del cambio.

4.6 Ejercicios

1. Considere a su reloj como un sistema y adelántelo 2 minutos. Escriba en forma de escenario cada interacción entre usted y el reloj. Registre todas las interacciones, incluyendo cualquier retroalimentación que le dé el reloj.
2. Considere el escenario que escribió en el ejercicio 1. Identifique al actor del escenario. Luego escriba el caso de uso `AjustarHora` correspondiente. Incluya todos los casos, e incluya adelantar la hora y atrasarla, y poner las horas, minutos y segundos.
3. Suponga que el sistema de reloj que describió en los ejercicios 1 y 2 también tiene alarma. Describa el ajuste de la hora de la alarma como un caso de uso autocontenido llamado `AjustarHoraAlarma`.
4. Examine los casos de uso `AjustarHora` y `AjustarHoraAlarma` que escribió en los ejercicios 2 y 3. Examine cualquier redundancia usando una relación de inclusión. Justifique por qué se prefiere, en este caso, una relación de inclusión en vez de una extendida.
5. Suponga que `OficialCampo` puede llamar una característica de Ayuda cuando llena un `ReporteDeEmergencia`. La característica `AyudaReportarEmergencia` proporciona una descripción detallada de cada campo y especifica cuáles campos se requieren. Modifique el caso de uso `ReportarEmergencia` (descrito en la figura 4-11) para incluir la funcionalidad de ayuda. ¿Cuál relación deberá usarse para vincular a `ReportarEmergencia` y `AyudaReportarEmergencia`?
6. A continuación se tienen ejemplos de requerimientos no funcionales. Especifique cuáles de los requerimientos son verificables y cuáles no.
   - "El sistema debe ser utilizable".
   - "El sistema debe proporcionar retroalimentación visual al usuario en menos de un segundo después de haber dado un comando".
   - "La disponibilidad del sistema debe ser superior a 95%".
   - "La interfaz de usuario del nuevo sistema debe ser lo suficientemente similar a la del sistema antiguo para que los usuarios familiarizados con el sistema antiguo puedan ser entrenados con facilidad en el uso del nuevo sistema".
7. Explique por qué los cuestionarios de opción múltiple no son efectivos como medio principal para la extracción de información para la obtención de requerimientos.
8. Desde su punto de vista, describa las fortalezas y debilidades de los usuarios durante la actividad de obtención de requerimientos. Describa también las fortalezas y debilidades de los desarrolladores durante la actividad de obtención de requerimientos.
9. Describa brevemente el término "menú". Escriba su respuesta en un pedazo de papel y póngalo boca abajo en la mesa junto con las definiciones de otros cuatro estudiantes. Compare las cinco definiciones y discuta cualquier diferencia sustancial.

Referencias

[Bruegge *et al.*, 1994] B. Bruegge, K. O'Toole y D. Rothenberger, "Design considerations for an accident management system", en *Proceedings of the Second International Conference on Cooperative Information Systems*, M. Brodie, M. Jarke y M. Papazoglou (eds.), University of Toronto Press, Toronto, Canadá, mayo de 1994, págs. 90-100.

[Carroll, 1995] J. M. Carroll (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development.* Wiley, Nueva York, 1995.

[Christel y Kang, 1992] M. G. Christel y K. C. Kang, "Issues in requirements elicitation", *Software Engineering Institute, Technical Report CMU/SEI-92-TR-12*, Carnegie Mellon Univ., Pittsburgh, PA, 1992.

[Constantine y Lockwood, 1999] L. L. Constantine y L. A. D. Lockwood, *Software for Use*. Addison-Wesley, Reading, MA, 1999.

[Dumas y Redish, 1998] Dumas y Redish, *A Practical Guide to Usability Testing*. Ablex, NJ, 1993.

[FRIEND, 1994] *FRIEND Project Documentation*. School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1994.

[Grudin, 1990] J. Grudin, "Obstacles to user involvement in interface design in large product development organizations", *Proceeding IFIP INTERACT'90 Third International Conference on Human-Computer Interaction*, Cambridge, Inglaterra, agosto de 1990.

[Hammer y Champy, 1993] M. Hammer y J. Champy, *Reengineering The Corporation: a Manifesto For Business Revolution.* Harper Business, Nueva York, 1993.

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, Reading, MA, 1995.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[Jacobson *et al.*, 1999] I. Jacobson, G. Booch y J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[Johnson, 1992] P. Johnson, *Human Computer Interaction: Psychology, Task Analysis and Software Engineering.* McGraw-Hill Int., Londres, 1992.

[Macaulay, 1996] L. Macaulay, *Requirements Engineering*. Springer Verlag, Londres, 1996.

[Nielsen, 1993] J. Nielsen, *Usability Engineering*. Academic, 1993.

[Nielsen y Mack, 1994] J. Nielsen y R. L. Mack (eds.), *Usability Inspection Methods*. Wiley, Nueva York, 1994.

[Paech, 1998] B. Paech, "The Four Levels of Use Case Description", *4th Int. Workshop on Requirements Engineering: Foundations for Software Quality*, Pisa, junio de 1998.

[Rubin, 1994] J. Rubin, *Handbook of Usability Testing*. Wiley, Nueva York, 1994.

[Wirfs-Brock, 1995] R. Wirfs-Brock, "Design objects and their interactions: A brief look at responsibility-driven design", *Scenario-Based Design: Envisioning Work and Technology in System Development.* J. M. Carroll (ed.), Wiley, Nueva York, 1995.

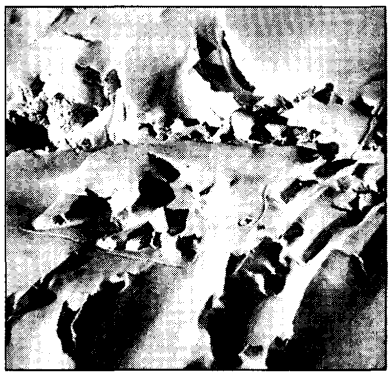
[Wirfs-Brock *et al.*, 1990] R. Wirfs-Brock, B. Wilkerson y Lauren Wiener, *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Wood y Silver, 1989] J. Wood y D. Silver, *Joint Application Design®*. Wiley, Nueva York, 1989.

[Zultner, 1993] R. E. Zultner, "TQM for Technical Teams", *Communications of the ACM*, vol. 36, núm. 10, 1993.

5

5



Análisis

Soy Foo con un nombre, si tan sólo pudiera recordarlo.

—Un programador con muy poco cerebro

El *análisis* da como resultado un modelo del sistema que pretende ser correcto, completo, consistente y verificable. Los desarrolladores formalizan la especificación del sistema producida durante la obtención de requerimientos y examinan con mayor detalle las condiciones de frontera y los casos excepcionales. También corrigen y aclaran la especificación del sistema si es que encuentran errores o ambigüedades. El cliente y el usuario están involucrados, por lo general, en esta actividad, en especial cuando se necesita cambiar la especificación del sistema y cuando se necesita recopilar información adicional.

En el análisis orientado a objetos, los desarrolladores construyen un modelo que describe al dominio de aplicación. Por ejemplo, el modelo de análisis de un reloj describe la manera en que el reloj representa al tiempo (por ejemplo, ¿Sabe el reloj acerca de los años bisiestos? ¿Sabe acerca del día de la semana? ¿sabe acerca de las fases de la luna?). El modelo de análisis se extiende luego para describir la manera en que interactúan los actores y el sistema para manipular el modelo del dominio de aplicación (por ejemplo, ¿cómo ajusta la hora el propietario del reloj? ¿Cómo ajusta el día de la semana?). Los desarrolladores usan el modelo de análisis, junto con los requerimientos no funcionales, para preparar la arquitectura del sistema que se desarrolla durante el diseño de alto nivel (capítulo 6, *Diseño del sistema*).

En este capítulo tratamos con mayor detalle las actividades de análisis. Nos enfocamos en la identificación de objetos, su comportamiento, sus relaciones, su clasificación y su organización. Revisamos en forma breve las presentaciones y métodos del análisis que no está orientado a objetos. Por último, describimos asuntos de administración relacionados con el análisis en el contexto de un proyecto de desarrollo de varios equipos.

5.1 Introducción: una ilusión óptica

En 1915, Rubin mostró una imagen similar a la figura 5-1 para ilustrar el concepto de imágenes multiestables. ¿Qué ve usted? ¿Dos caras, una frente a otra? Si se enfoca más en el área blanca, en vez de ello puede ver un jarrón. Una vez que puede percibir ambas formas de manera individual es fácil alternar entre el jarrón y las caras.

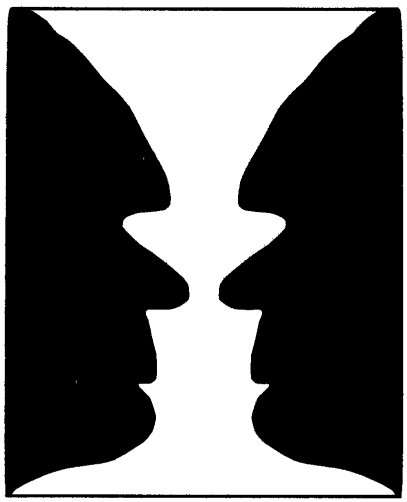


Figura 5-1 Ambigüedad: ¿Qué ve usted?

Si la imagen de la figura 5-1 hubiera sido una especificación de sistema, ¿cuáles modelos habría que construir? Las especificaciones, al igual que las imágenes multiestables, contienen ambigüedades causadas por las imprecisiones inherentes al lenguaje natural y por las suposiciones de los autores acerca de las especificaciones. Por ejemplo, una cantidad especificada sin unidades es ambigua (por ejemplo, el caso de "¿metros o kilómetros?" de la sección 4.1). Una hora sin zona horaria es ambigua (por ejemplo, programar una llamada telefónica entre países diferentes).

La formalización ayuda a identificar áreas de ambigüedad, así como inconsistencias y omisiones en una especificación de sistema. Una vez que los desarrolladores identifican problemas con la especificación, los resuelven obteniendo más información de los usuarios y el cliente. La obtención de requerimientos y el análisis son actividades iterativas e incrementales que suceden en forma concurrente.

5.2 Un panorama del análisis

El análisis se enfoca en la producción de un modelo del sistema, llamado el modelo de análisis, que es correcto, completo, consistente y verificable. El análisis se diferencia de la obtención de requerimientos en que los desarrolladores se enfocan en la estructuración y formalización de los

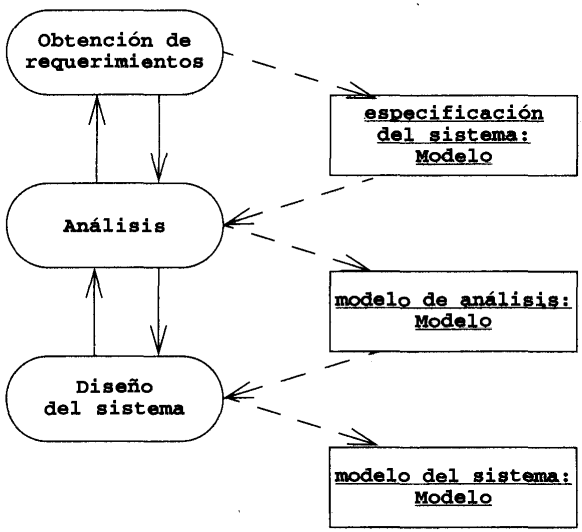


Figura 5-2 Productos de la obtención de requerimientos y el análisis (diagrama de actividad UML).

requerimientos obtenidos de los usuarios (figura 5-2). Aunque puede ser que el modelo de análisis no sea comprensible para los usuarios y el cliente, ayuda a que los desarrolladores verifiquen la especificación del sistema producida durante la obtención de requerimientos.

Hay una tendencia natural para que los usuarios y desarrolladores pospongan las decisiones difíciles hasta lo último del proyecto. Una decisión puede ser difícil debido a la falta de conocimiento del dominio, falta de conocimiento tecnológico o tan sólo a causa de desacuerdos entre usuarios y desarrolladores. Posponer las decisiones permite que el proyecto avance con suavidad y evita la confrontación con la realidad o entre los participantes. Por desgracia, tarde o temprano se tendrán que tomar las decisiones difíciles, a menudo con costos más elevados, cuando se descubran problemas intrínsecos durante las pruebas o, lo que es peor, durante la evaluación del usuario. La traducción de una especificación de sistema a un modelo formal o semiformal obliga a los desarrolladores a que identifiquen y resuelvan asuntos difíciles al principio del desarrollo.

El **modelo de análisis** está compuesto por tres modelos individuales: el **modelo funcional**, representado por casos de uso y escenarios, el **modelo de objetos de análisis**, representado por diagramas de clase y objeto, y el **modelo dinámico**, representado por gráficas de estado y diagramas de secuencia (figura 5-3). En el capítulo anterior describimos la manera de obtener requerimientos de los usuarios y los describimos como casos de uso y escenarios. En este capítulo describimos la manera de refinar el modelo funcional y derivar el modelo de objetos y el dinámico. Esto conduce a una descripción más precisa y completa conforme se añaden detalles al modelo de análisis. Concluimos el capítulo describiendo las actividades administrativas que están relacionadas con el análisis. En la siguiente sección, primero definimos los conceptos principales del análisis.

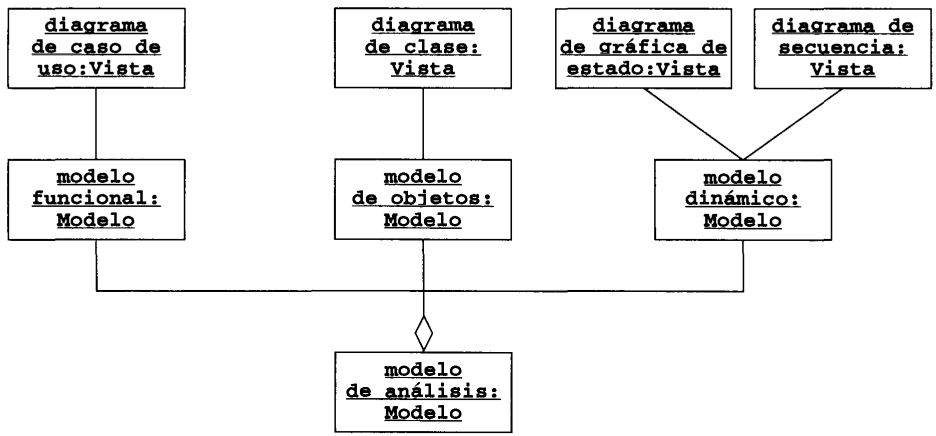


Figura 5-3 El modelo de análisis está compuesto por el modelo funcional, el modelo de objetos y el modelo dinámico. En UML, el modelo funcional se representa con diagramas de caso de uso, el modelo de objetos con diagramas de clase y el modelo dinámico con gráficas de estado y diagramas de secuencia.

5.3 Conceptos de análisis

En esta sección describimos los principales conceptos de análisis que se usan en este capítulo. Particularmente describimos:

- Objetos de entidad, frontera y control (sección 5.3.1)
- Multiplicidad de asociación (sección 5.3.2)
- Asociaciones calificadas (sección 5.3.3)
- Generalización (sección 5.3.4)

5.3.1 Objetos de entidad, frontera y control

El modelo de objetos de análisis consiste en objetos de entidad, frontera y control [Jacobson *et al.*, 1999]. Los **objetos de entidad** representan la información persistente rastreada por el sistema. Los **objetos de frontera** representan la interacción entre los actores y el sistema. Los **objetos de control** representan las tareas realizadas por el usuario y soportadas por el sistema. En el ejemplo Reloj2B, `Año`, `Mes`, `Día` son objetos de entidad, `FronteraBotón` y `FronteraPantallaLCD` son objetos de frontera, `ControlCambioFecha` es un objeto de control que representa la actividad de cambiar la fecha oprimiendo combinaciones de botones.

Modelar el sistema con objetos de entidad, frontera y control tiene varias ventajas. Primero, proporciona a los desarrolladores heurística simple para distinguir conceptos diferentes pero relacionados. Por ejemplo, la hora que lleva un reloj tiene propiedades diferentes que la pantalla que muestra la hora. La diferenciación entre los objetos de frontera y entidad obliga a esta distinción: la hora que es seguida por el reloj está representada por el objeto `Hora`. La pantalla está representada por `FronteraPantallaLCD`. Segundo, el enfoque de tres tipos de objetos da como

resultado objetos más pequeños y más especializados. Tercero, el enfoque de tres tipos de objetos conduce a modelos que son más adaptables al cambio: es más probable que cambie la interfaz del sistema (representada por los objetos de frontera) que la funcionalidad básica (representada por los objetos de entidad y control).

Para distinguir entre diferentes tipos de objetos, UML proporciona el mecanismo de estereotipo para permitir que el desarrollador añada esa metainformación a los elementos del modelado. Por ejemplo, en la figura 5-4 añadimos el estereotipo `<<control>>` al objeto `ControlCambioFecha`. Además de los estereotipos, también podemos usar convenciones de denominación para efectos de claridad, y recomendamos que se haga distinción entre los tres tipos de objetos diferentes sobre una base sintáctica: los objetos de frontera pueden tener el prefijo `Frontera` añadido a su nombre. Los objetos de control pueden tener el prefijo `Control`, y los objetos de entidad, por lo general, no tienen ningún prefijo. Otro beneficio de esta convención de denominación es que está representado el tipo de la clase aun cuando no se disponga del estereotipo UML, por ejemplo, cuando sólo se examina el código fuente.

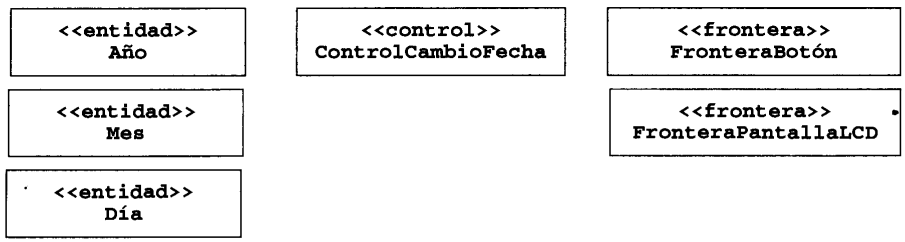


Figura 5-4 Clases de análisis para el ejemplo Reloj2B.

5.3.2 Revisión de la multiplicidad de asociación

Como vimos en el capítulo 2, *Modelado con UML*, el extremo de una asociación puede etiquetarse con un conjunto de enteros llamados **multiplicidad**. La multiplicidad indica la cantidad de vínculos que pueden originarse legítimamente desde una instancia de la clase conectada al extremo de la asociación. Por ejemplo, en la figura 5-5, un Reloj2B tiene exactamente dos `Botones` y una `PantallaLCD`.

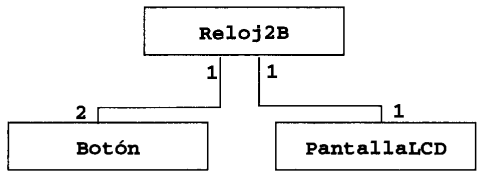


Figura 5-5 Un ejemplo de multiplicidad de asociaciones (diagrama de clase UML). Un Reloj2B tiene dos botones y una `PantallaLCD`.

En UML, un extremo de una asociación puede tener como multiplicidad un conjunto de enteros arbitrarios. Por ejemplo, una asociación podría permitir sólo un número primo de vínculos, esto es, podría tener una multiplicidad de 1, 2, 3, 5, 7, 11, 13... Sin embargo, en la práctica, la mayoría de las asociaciones que encontramos pertenece a alguno de los siguientes tres tipos (vea la figura 5-6).

- **Una asociación de uno a uno** tiene una multiplicidad de 1 a cada extremo. Una asociación de uno a uno entre dos clases (por ejemplo, `OficialPolicía` y `NúmeroIdentificación`) significa que existe solamente un vínculo entre instancias de cada clase (por ejemplo, un `OficialPolicía` tiene exactamente un `NúmeroIdentificación`, y un `NúmeroIdentificación` indica exactamente a un `OficialPolicía`).
- **Una asociación de uno a muchos** tiene una multiplicidad de `1` en un extremo y `0..n` (también representada por asterisco) o `1..n` en el otro. Una asociación de uno a muchos entre dos clases (por ejemplo, `Persona` y `Automóvil`) indica composición (por ejemplo, una `UnidadBomberos` posee uno o más `CamiónBomberos` y un `CamiónBomberos` pertenece exactamente a una `UnidadBomberos`).
- **Una asociación de muchos a muchos** tiene una multiplicidad de `0..n` o `1..n` en ambos extremos. Una asociación de muchos a muchos entre dos clases (por ejemplo, `OficialCampo` y `ReporteDeIncidente`) indica que puede existir una cantidad arbitraria de vínculos entre instancias de las dos clases (por ejemplo, un `OficialCampo` puede escribir muchos `ReporteDeIncidente`, y un `ReporteDeIncidente` puede ser escrito por muchos `OficialCampo`). Éste es el tipo más complejo de asociación.

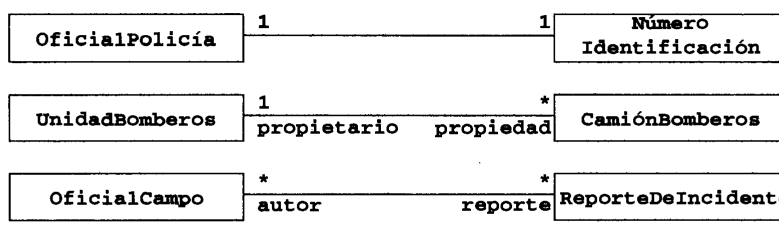


Figura 5-6 Ejemplos de multiplicidad (diagrama de clase UML). La asociación entre `Persona` y `NúmeroSeguroSocial` es de uno a uno. La asociación entre `Persona` y `Automóvil` es de uno a muchos. La asociación entre `Persona` y `Compañía` es de muchos a muchos.

La adición de multiplicidad a las asociaciones incrementa la cantidad de información que capturamos del dominio de aplicación o del dominio de solución. La especificación de la multiplicidad de una asociación llega a ser crítica cuando determinamos cuáles casos de uso se necesitan para manipular objetos del dominio de aplicación. Por ejemplo, considere un sistema de archivo compuesto de `Directorio`(s) y `Archivo`(s). Un `Directorio` puede contener cualquier cantidad de *`ElementoSistemaArchivo`*. Un *`ElementoSistemaArchivo`* es un concepto abstracto que indica un `Directorio` o un `Archivo`. En caso de un sistema estrictamente jerárquico, un *`ElementoSistemaArchivo`* es parte de exactamente un `Directorio`, y esto lo indicamos con una multiplicidad de uno a muchos (figura 5-7).

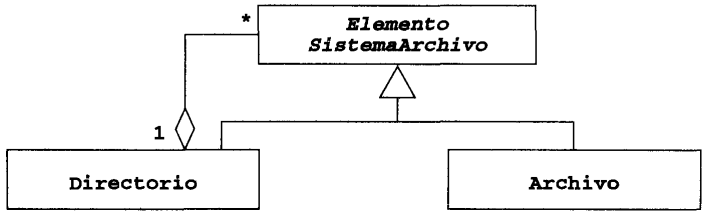


Figura 5-7 Ejemplo de sistema de archivos jerárquico. Un `Directorio` puede contener cualquier cantidad de *`ElementoSistemaArchivo`*. (Un *`ElementoSistemaArchivo`* es un `Archivo` o un `Directorio`.) Sin embargo, un *`ElementoSistemaArchivo`* es parte de exactamente un `Directorio`.

Sin embargo, si `Archivo` o `Directorio` puede ser al mismo tiempo parte de más de un `Directorio`, necesitamos representar la agregación de *`ElementoSistemaArchivo`* con `Directorio` como una asociación de muchos a muchos (vea la figura 5-8).

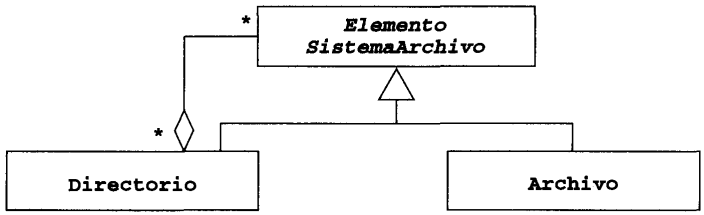


Figura 5-8 Ejemplo de un sistema de archivo no jerárquico. Un `Directorio` puede contener cualquier cantidad de *`ElementoSistemaArchivo`* (un *`ElementoSistemaArchivo`* es un `Archivo` o un `Directorio`). Un *`ElementoSistemaArchivo`* puede ser parte de muchos `Directorio`(s).

Puede parecer que esta discusión está considerando asuntos detallados que podrían dejarse para actividades posteriores en el proceso de desarrollo. Sin embargo, la diferencia entre un sistema de archivo jerárquico y otro no jerárquico también reside en la funcionalidad que proporciona. Si un sistema permite que un `Archivo` dado sea parte de varios `Directorio`(s), necesitamos definir un caso de uso que describa la manera en que un usuario añade un `Archivo` existente a `Directorio`(s) existentes (por ejemplo, el comando `link` de Unix o el concepto de menú `MakeAlias` de Macintosh). Además, los casos de uso que eliminan un `Archivo` de un `Directorio` deben especificar si el `Archivo` se elimina sólo de un `Directorio` o de todos los `Directorio`(s) que hacen referencia a él. Observe que una asociación de muchos a muchos puede dar como resultado un sistema sustancialmente más complejo.

5.3.3 Asociaciones calificadas

La **calificación** es una técnica para la reducción de la multiplicidad usando claves. Las asociaciones que tienen una multiplicidad de `0..1` o `1` son más fáciles de comprender que las asociaciones

con multiplicidad `0..n` o `1..n`. Con frecuencia, en el caso de una asociación de uno a muchos, los objetos del lado de "muchos" pueden distinguirse entre ellos usando un nombre. Por ejemplo, en un sistema de archivos jerárquicos, cada archivo pertenece exactamente a un directorio. Cada archivo está identificado en forma única por un nombre dentro del contexto de un directorio. Muchos archivos pueden tener el mismo nombre dentro del contexto de sistema de archivo, pero dos archivos no pueden compartir un mismo nombre dentro del mismo directorio. Sin calificación (vea la parte superior de la figura 5-9), la asociación entre `Directorio` y `Archivo` tiene multiplicidad de 1 en el lado `Directorio` y multiplicidad cero a muchos del lado `Archivo`. Reducimos la multiplicidad del lado `Archivo` usando el atributo `nombrearchivo` como clave, también llamada **calificador** (vea la parte superior de la figura 5-9). A la relación entre `Directorio` y `Archivo` se le llama **asociación calificada**.

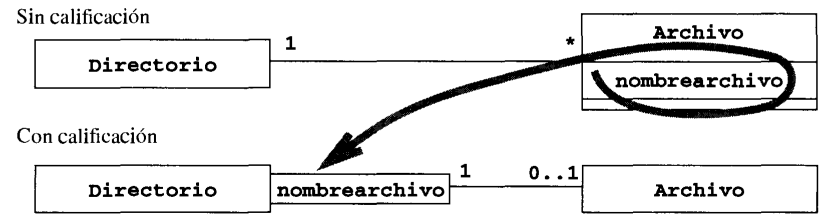


Figura 5-9 Ejemplo de la manera en que la asociación calificada reduce la multiplicidad (diagrama de clase UML). La adición de un calificador aclara el diagrama de clase e incrementa la información que conlleva. En este caso, el modelo que incluye la calificación indica que el nombre del archivo es único dentro de un directorio.

Siempre es preferible la reducción de la multiplicidad, ya que el modelo se hace más claro y se tienen que tomar en cuenta menos casos. Los desarrolladores deben examinar cada asociación que tiene multiplicidad de uno a muchos, o de muchos a muchos, y revisar si se puede añadir un calificador. A menudo, estas asociaciones pueden calificarse con un atributo de la clase de destino (por ejemplo, el atributo `nombrearchivo` en la figura 5-9).

5.3.4 Generalización

Como vimos en el capítulo 2, *Modelado con UML*, la **generalización** nos permite organizar conceptos en jerarquías. En la parte superior de la jerarquía está un concepto general (por ejemplo, un `Incidente`, figura 5-10) y en la parte inferior de la jerarquía están los conceptos más especializados (por ejemplo, `GatoEnÁrbol`, `AccidenteTráfico`, `IncendioEdificio`, `Temblor`, `FugaQuímica`). Puede haber cualquier cantidad de niveles intermedios entre ellos, cubriendo conceptos más o menos generalizados (por ejemplo, `BajaPrioridad`, `Emergencia`, `Desastre`). Tales jerarquías nos permiten hacer referencia con precisión a muchos conceptos. Cuando usamos el término `Incidente` queremos decir todas las instancias de todos los tipos de `Incidente`. Cuando usamos el término `Emergencia` sólo nos referimos a un `Incidente` que requiere una respuesta inmediata. Esta vista de la generalización proviene del modelado.

En un lenguaje de programación orientado a objetos, la **herencia** es una técnica de reutilización. Si una clase `Hijo` hereda de una clase `Padre`, todos los atributos y métodos que están

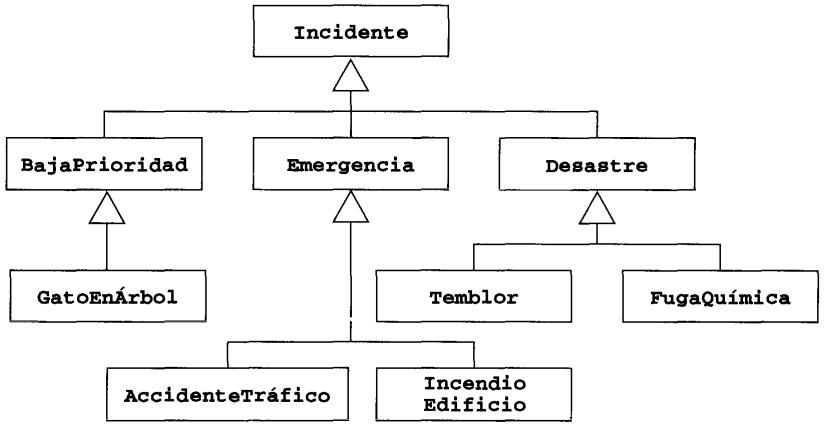


Figura 5-10 Un ejemplo de una jerarquía de generalización (diagrama de clase UML). La raíz de la jerarquía representa el concepto más general, mientras que los nodos hojas representan los conceptos más especializados.

disponibles en `Padre` se tienen disponibles automáticamente en `Hijo`. La clase `Hijo` puede añadir métodos adicionales o sobreponer métodos heredados, refinando así a la clase `Padre`. Como sucede en el caso de la generalización, la clase que está en la parte superior de la jerarquía tiende a ser la más general, mientras las hojas tienden a ser las más especializadas.

La generalización y la herencia son conceptos muy relacionados. Sin embargo, no son idénticos. La herencia es un mecanismo para la reutilización de atributos y comportamientos, aunque las clases involucradas en la herencia no tengan una relación de generalización. Por ejemplo, es posible implementar una colección `Conjunto` refinando una `Tablahash` (aunque puede ser que esto no sea una buena solución para implementar `Conjunto`). Sin embargo, un `Conjunto` no es un concepto más especializado que `Tablahash`, y tampoco `Tablahash` es una generalización de un `Conjunto`.

Durante el análisis sólo nos enfocamos en la organización de conceptos en relaciones de generalización y no se deben interpretar desde el punto de vista de la reutilización. Sin embargo, aunque la jerarquía de herencia se deriva al principio de la jerarquía de generalización, se reestructurará durante el diseño de objetos.

5.4 Actividades de análisis: desde los casos de uso hasta los objetos

En esta sección describimos las actividades que transforman los casos de uso y escenarios producidos durante la obtención de requerimientos hacia un modelo de análisis. Las actividades de análisis incluyen:

- Identificación de objetos de entidad (sección 5.4.1)
- Identificación de objetos de frontera (sección 5.4.2)
- Identificación de objetos de control (sección 5.4.3)
- Modelado de interacciones entre objetos (sección 5.4.4)
- Identificación de asociaciones entre objetos (sección 5.4.5)

- Identificación de atributos de objetos (sección 5.4.6)
- Modelado de comportamiento no trivial con gráficas de estado (sección 5.4.7)
- Modelado de relaciones de generalización (sección 5.4.8)
- Revisión del modelo de análisis (sección 5.4.9)

Ilustramos cada actividad enfocándonos en el caso de uso `ReportarEmergencia` de FRIEND, descrito en el capítulo 4, *Obtención de requerimientos*. Estas actividades están guiadas por heurística. La calidad de sus resultados depende de la experiencia del desarrollador en la aplicación de la heurística y los métodos. Los métodos y la heurística que se presentan en esta sección están adaptados de [De Marco, 1978], [Jacobson *et al.*, 1999], [Rumbaugh *et al.*, 1991] y [Wirfs-Brock *et al.*, 1990].

5.4.1 Identificación de objetos de entidad

Los objetos participantes (vea la sección 4.4.6) forman la base del modelo de análisis. Como se describió en el capítulo 4, *Obtención de requerimientos*, los objetos participantes se encuentran examinando cada caso de uso e identificando objetos candidatos. El análisis del lenguaje natural [Abbott, 1983] es un conjunto de heurística intuitiva para la identificación de objetos, atributos y asociaciones a partir de una especificación del sistema. La heurística de Abbott hace corresponder partes del habla (por ejemplo, nombres, verbos posesivos, verbos de acción, adjetivos) con componentes del modelo (por ejemplo, objetos, operaciones, relaciones de herencia, clases). La tabla 5-1 proporciona ejemplos de tales correspondencias examinando el caso de uso `ReportarEmergencia` de la figura 5-11.

Tabla 5-1 Heurística de Abbott para la correspondencia de partes del habla con los componentes del modelo [Abbott, 1983].

Parte del habla	Componente del modelo	Ejemplos
Sustantivo propio	Objeto	Alicia
Sustantivo común	Clase	OficialCampo
Verbo de acción	Operación	Crea, envía, selecciona
Verbo de ser	Herencia	Es un tipo de, es alguno de
Verbo de tener	Agregación	Tiene, consiste en, incluye
Verbo modal	Restricciones	Debe ser
Adjetivo	Atributo	Descripción del incidente

El análisis del lenguaje natural tiene la ventaja de enfocarse en los términos del usuario. Sin embargo, tiene varias limitaciones. Primero, la calidad del modelo de objetos depende en gran medida del estilo de escritura del analista (por ejemplo, consistencia en los términos usados, transformación de sustantivos en verbos). El lenguaje natural es una herramienta imprecisa, y un modelo de objetos derivado literalmente del texto tiene el riesgo de ser impreciso. Los desarrolladores

pueden atacar esta limitación volviendo a redactar y aclarando la especificación del sistema conforme identifican y estandarizan los objetos y términos. Una segunda limitación del análisis del lenguaje natural es que hay muchos más nombres que clases relevantes. Muchos nombres corresponden a atributos o sinónimos de otros nombres. El ordenamiento de todos los nombres en una especificación de un sistema grande es una actividad que consume tiempo. La heurística de Abbott funciona bien para la generación de una lista de objetos candidatos iniciales a partir de descripciones cortas, como el flujo de eventos de un escenario o un caso de uso. La siguiente heurística puede usarse junto con las reglas de Abbott:

Heurística para la identificación de objetos de identidad

- Términos que los desarrolladores o usuarios necesitan aclarar para comprender el caso de uso.
- Nombres recurrentes en los casos de uso (por ejemplo, `Incidente`).
- Entidades del mundo real de las que necesita llevar cuenta el sistema (por ejemplo, `OficialCampo`, `Despachador`, `Recurso`).
- Actividades del mundo real de las que necesita llevar cuenta el sistema (por ejemplo, `PlanOperacionesEmergencia`).
- Casos de uso (por ejemplo, `ReportarEmergencia`).
- Fuentes o destinos de datos (por ejemplo, `Impresora`).
- *Siempre* hay que usar los términos del usuario.

Los desarrolladores nombran y describen en forma breve los objetos, sus atributos y sus responsabilidades conforme los identifican. La denominación única de los objetos promueve una terminología estándar. La descripción de los objetos, aunque sea breve, permite que los desarrolladores aclaren los conceptos que usan y eviten tergiversaciones (por ejemplo, el uso de un objeto para dos conceptos diferentes pero relacionados). Sin embargo, los desarrolladores no necesitan pasar mucho tiempo detallando objetos o atributos, tomando en cuenta que el modelo de análisis todavía está fluyendo. Los desarrolladores deben documentar atributos y responsabilidades si son obvias. Si no es así, un nombre tentativo y una descripción breve es suficiente para cada objeto. Habrá muchas iteraciones durante las cuales se pueden revisar los objetos. Sin embargo, una vez que el modelo de análisis es estable, la descripción de cada objeto deberá ser tan detallada como se necesite (vea la sección 5.4.9).

Por ejemplo, después de un primer examen del caso de uso `ReportarEmergencia` (figura 5-11) usamos el conocimiento del dominio de aplicación y entrevistas con los usuarios para identificar los objetos `Despachador`, `ReporteDeEmergencia`, `OficialCampo` e `Incidente`. Observe que el objeto `ReporteDeEmergencia` no se menciona de manera explícita en el caso de uso `ReportarEmergencia`. El paso 3 del caso de uso hace referencia a un reporte de emergencia como "información enviada por el `OficialCampo`". Después de revisar con el cliente descubrimos que a esta información se le menciona por lo general como el reporte de emergencia, y se decide nombrar `ReporteDeEmergencia` al objeto correspondiente.

Nombre del caso de uso	`ReportarEmergencia`
Condición inicial	1. El `OficialCampo` activa la función "Reportar Emergencia" de su terminal.
Flujo de eventos	2. FRIEND responde presentando un formulario al oficial. El formulario incluye un menú de tipo de emergencia (emergencia general, incendio, transporte), campos para ubicación, descripción del incidente, petición de recursos y materiales peligrosos.
	3. El `OficialCampo` llena el formulario, seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia y solicita recursos específicos. Una vez que ha llenado el formulario, el `OficialCampo` lo envía oprimiendo el botón "Enviar Reporte" y en ese momento se le notifica al `Despachador`.
	4. El `Despachador` revisa la información enviada por el `OficialCampo`, y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. En el incidente se incluye en forma automática toda la información contenida en el formulario del `OficialCampo`. El `Despachador` selecciona una respuesta asignando recursos al incidente (con el caso de uso `AsignarRecursos`) y da un acuse de recibo del reporte de emergencia enviando un FRIENDgrama al `OficialCampo`.
Condición de salida	5. El `OficialCampo` recibe el acuse de recibo y la respuesta seleccionada.

Figura 5-11 Un ejemplo del caso de uso `ReportarEmergencia`.

La definición de objetos de entidad conduce al modelo de análisis inicial que se describe en la tabla 5-2.

Observe que el modelo de objeto anterior está muy lejos de ser una descripción completa del sistema para la implementación del caso de uso `ReportarEmergencia`. En la siguiente sección describimos la identificación de objetos de frontera.

5.4.2 Identificación de objetos de frontera

Los objetos de frontera representan la interfaz del sistema con los actores. En cada caso de uso, cada actor interactúa con, al menos, un objeto de frontera. El objeto de frontera recopila la información del actor y la traduce hacia una forma neutral de interfaz que puede ser usada por los objetos de entidad y también por los objetos de control.

Los objetos de frontera modelan la interfaz de usuario a un nivel burdo. No describen con detalle los aspectos visuales de la interfaz de usuario. Por ejemplo, objetos de frontera como "botón" o "concepto de menú" están demasiado detallados. Primero, los desarrolladores pueden discutir los detalles de interfaz de usuario con más facilidad con bosquejos y maquetas. Segundo, la idea del diseño de la interfaz de usuario continuará evolucionando a consecuencia de la prueba

Tabla 5-2 Objetos de entidad para el caso de uso `ReportarEmergencia`.

`Despachador`	Es un oficial de policía que administra `Incidente`. Un `Despachador` abre, documenta y cierra `Incidente` en respuesta a `ReporteDeEmergencia` y otras comunicaciones con los `OficialCampo`. Los `Despachador` se identifican mediante sus números de identificación.
`ReporteDeEmergencia`	Es el reporte inicial acerca de un `Incidente` que hace un `OficialCampo` hacia un `Despachador`. Por lo general, un `ReporteDeEmergencia` activa la creación de un `Incidente` por parte del `Despachador`. Un `ReporteDeEmergencia` está compuesto de un nivel de emergencia, un tipo (incendio, accidente de carretera u otro), una ubicación y una descripción.
`OficialCampo`	Es un oficial de policía o bomberos que está trabajando. Un `OficialCampo` puede asignarse, a lo mucho, a un `Incidente` a la vez. Los `OficialCampo` están identificados mediante sus números de identificación.
`Incidente`	Es una situación que requiere atención de un `OficialCampo`. Un `Incidente` puede ser reportado al sistema por un `OficialCampo` o alguna persona externa al sistema. Un `Incidente` está compuesto por una descripción, una respuesta, un estado (abierto, cerrado, documentado), una ubicación y una cantidad de `OficialCampo`.

de utilidad, incluso después de que llegue a ser estable la especificación funcional del sistema. La actualización del modelo de análisis cada vez que se hace un cambio visual a la interfaz consume tiempo y no produce ningún beneficio sustancial.

Heurística para la identificación de objetos de frontera

- Identificar formularios y ventanas que el usuario necesita para dar datos al sistema (por ejemplo, `FormularioReporteDeEmergencia`, `BotónReportarEmergencia`).
- Identificar noticias y mensajes que el sistema usa para responder al usuario (por ejemplo, `NoticiaAcuseDeRecibo`).
- No hay que modelar los aspectos visuales de la interfaz con objetos de frontera (las maquetas son más adecuadas).
- *Siempre* hay que usar los términos del usuario para describir las interfaces en vez de los términos de la tecnología de implementación.

Examinando el caso de uso `ReportarEmergencia` encontramos los siguientes objetos de frontera (tabla 5-3).

Observe que el `FormularioIncidente` no se menciona de manera explícita en ningún lugar del caso de uso `ReportarEmergencia`. Identificamos este objeto observando que el `Despachador` necesita una interfaz para ver el reporte de emergencia enviado por el `OficialCampo` y para regresar un acuse de recibo. Los términos usados para describir los objetos de

Tabla 5-3 Objetos de frontera para el caso de uso `ReportarEmergencia`.

`NoticiaAcuseDeRecibo`	Noticia usada para desplegar el acuse de recibo del `Despachador` hacia el `OficialCampo`.
`EstaciónDespachador`	Computadora usada por el `Despachador`.
`BotónReportarEmergencia`	Botón usado por el `OficialCampo` para iniciar el caso de uso `ReportarEmergencia`.
`FormularioReporteDeEmergencia`	Formulario usado para dar los datos de `ReportarEmergencia`. Este formulario se le presenta al `OficialCampo` en la `EstaciónOficialCampo` cuando se selecciona la función "Reportar Emergencia". El `FormularioReporteDeEmergencia` contiene campos para especificar todos los atributos de un reporte de emergencia y un botón (u otro control) para enviar el formulario cuando está lleno.
`EstaciónOficialCampo`	Computadora móvil usada por el `OficialCampo`.
`FormularioIncidente`	Formulario usado para la creación de `Incidente`. Este formulario se le presenta al `Despachador` en la `EstaciónDespachador` cuando se recibe el `ReporteDeEmergencia`. El `Despachador` también usa este formulario para asignar recursos y dar el acuse de recibo al reporte del `OficialCampo`.

frontera en el modelo de análisis deben apegarse a la terminología del usuario, aunque sea tentador usar términos del dominio de implementación.

Hemos avanzado hacia la descripción del sistema. Ahora hemos incluido la interfaz entre el actor y el sistema. Sin embargo, todavía faltan partes significativas de la descripción, como el orden en que sucede la interacción entre los actores y el sistema. En la siguiente sección describimos la identificación de los objetos de control.

5.4.3 Identificación de objetos de control

Los objetos de control son responsables de la coordinación entre objetos de frontera y de entidad. Por lo general, los objetos de control no tienen una contraparte concreta en el mundo real. A menudo hay una relación cercana entre un caso de uso y un objeto de control. Un objeto de control se crea, por lo general, al inicio del caso de uso y deja de existir cuando termina. Es responsable de la recopilación de información de los objetos de frontera y de enviarla a los objetos de entidad. Por ejemplo, los objetos de control describen el comportamiento asociado con la secuencia de formularios, las colas de deshacer y de historia y el envío de información en un sistema distribuido.

Heurística para la identificación de los objetos de control

- Identifique un objeto de control por caso de uso o más si el caso de uso es complejo y si puede dividirse en flujos de eventos más cortos.
- Identifique un objeto de control por actor en el caso de uso.
- La vida de un objeto de control debe ser la del caso de uso o la de la sesión del usuario. Si es difícil identificar el inicio y final de la activación de un objeto de control, puede ser que el caso de uso correspondiente no tenga condiciones de entrada y salida bien definidas.

Al principio modelamos el flujo de control del caso de uso `ReportarEmergencia` con un objeto de control para cada actor, `ControlReportarEmergencia` para el `OficialCampo` y `ControlAdministrarEmergencia` para el `Despachador`, respectivamente (tabla 5-4).

La decisión de modelar el flujo de control del caso de uso `ReportarEmergencia` con dos objetos de control procede del conocimiento de que `EstaciónOficialCampo` y `EstaciónDespachador` son, de hecho, dos subsistemas que se comunican a través de un vínculo asíncrono. Esta decisión podría haberse pospuesto hasta la actividad de diseño del sistema. Por otro lado, hacer visible este concepto en el modelo de análisis nos permite enfocarnos en el comportamiento excepcional que es la pérdida de comunicación entre ambas estaciones.

Al modelar el caso de uso `ReportarEmergencia` modelamos la misma funcionalidad usando objetos de entidad, frontera y control. Al cambiar de la perspectiva del flujo de eventos hacia una perspectiva estructural incrementamos el nivel de detalle de la descripción y seleccionamos términos estándar para referirnos a las entidades del dominio de aplicación y del

Tabla 5-4 Objetos de control para el caso de uso `ReportarEmergencia`.

`ControlReportarEmergencia`	Maneja la función de reporte de `ReportarEmergencia` en la `EstaciónOficialCampo`. Este objeto se crea cuando el `OficialCampo` selecciona el botón "Reportar Emergencia". Luego crea un `FormularioReporteDeEmergencia` y lo presenta al `OficialCampo`. Después del envío del formulario, este objeto recopila la información del formulario, crea un `ReporteDeEmergencia` y se lo pasa al `Despachador`. El objeto de control luego espera la llegada de un acuse de recibo desde la `EstaciónDespachador`. Cuando llega el acuse de recibo, el objeto `ControlReportarEmergencia` crea una `NoticiaAcuseDeRecibo` y la despliega ante el `OficialCampo`.
`ControlAdministrarEmergencia`	Maneja la función de reporte de `ReportarEmergencia` en la `EstaciónDespachador`. Este objeto se crea cuando se recibe un `ReporteDeEmergencia`. Luego crea un `FormularioIncidente` y lo despliega ante el `Despachador`. Una vez que el `Despachador` ha creado un `Incidente`, le ha asignado `Recursos` y ha enviado un acuse de recibo, `ControlAdministrarEmergencia` envía el acuse de recibo a la `EstaciónOficialCampo`.

sistema. En la siguiente sección construimos un diagrama de secuencia usando el caso de uso `ReportarEmergencia` y los objetos que descubrimos para asegurar la suficiencia de nuestro modelo.

5.4.4 Modelado de interacciones entre objetos: diagramas de secuencia

Un diagrama de secuencia une los casos de uso con los objetos. Muestra cómo se distribuye el comportamiento de un caso de uso (o escenario) entre sus objetos participantes. Los diagramas de secuencia no son, por lo general, un buen medio de comunicación con el usuario. Sin embargo, representan otro cambio de perspectiva y permiten que los desarrolladores encuentren objetos faltantes o áreas grises en la especificación del sistema.

En esta sección modelamos la secuencia de interacciones entre los objetos necesarios para realizar el caso de uso. Las figuras 5-12 a 5-14 son diagramas de secuencia asociados con el caso de uso `ReportarEmergencia`. Las columnas de un diagrama de secuencia representan a los objetos que participan en el caso de uso. La columna de la extrema izquierda es el actor que inicia el caso de uso. Las flechas horizontales a través de columnas representan mensajes, o estímulos, que son enviados de un objeto a otro. El tiempo avanza en forma vertical de arriba hacia abajo. Por ejemplo, la flecha de la figura 5-12 representa el mensaje `oprimir` enviado por un `OficialCampo` a un `BotónReportarEmergencia`. La recepción de un mensaje dispara la activación de una operación. La activación está representada por un rectángulo desde donde se pueden originar otros mensajes. La longitud del rectángulo representa el tiempo que la operación está activa. En la figura 5-12, la operación disparada por el mensaje `oprimir` envía un mensaje `crear` a la clase `ControlReportarEmergencia`. Puede pensarse en una operación como un servicio que el objeto proporciona a otros objetos.

Heurística para el trazado de diagramas de secuencia

- La primera columna debe corresponder al actor que inicia el caso de uso.
- La segunda columna debe ser un objeto de frontera (que usa el actor para iniciar el caso de uso).
- La tercera columna debe ser el objeto de control que maneja el resto del caso de uso.
- Los objetos de control son creados por objetos de frontera que inician casos de uso.
- Los objetos de frontera son creados por objetos de control.
- Los objetos de entidad son accedidos por objetos de control y de frontera.
- Los objetos de entidad *nunca* tienen acceso a los objetos de frontera o control, y esto hace que sea más fácil compartir objetos de entidad entre casos de uso.

En general, la segunda columna de un diagrama de secuencia representa el objeto de frontera con el cual interactúa el actor para iniciar el caso de uso (por ejemplo, `BotónReportarEmergencia`). La tercera columna es un objeto de control que maneja el resto del caso de uso (por ejemplo, `ControlReportarEmergencia`). A partir de ahí el objeto de control crea otros objetos de frontera y también puede interactuar con otros objetos de control (en este caso, el objeto `ControlAdministrarEmergencia`).

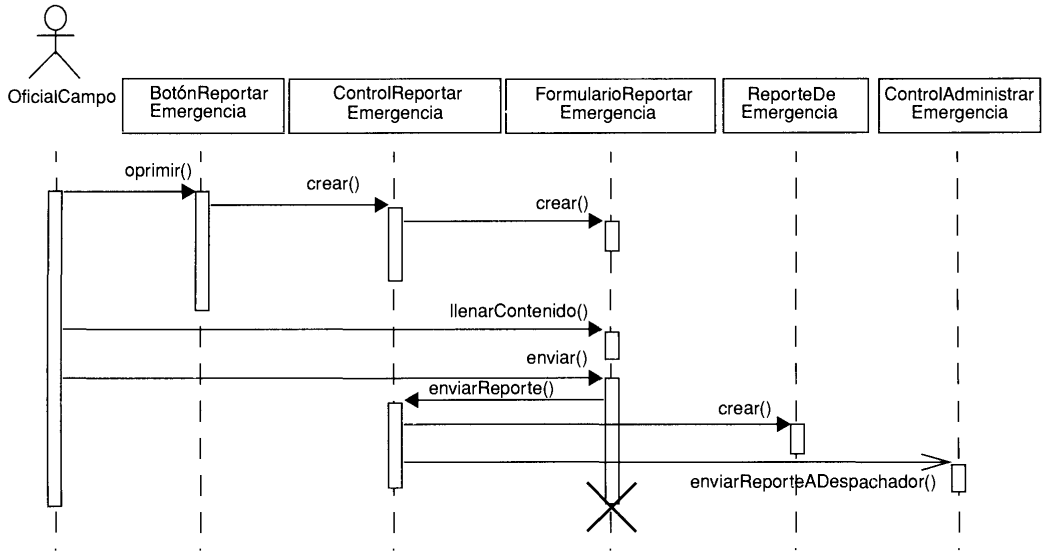


Figura 5-12 Diagrama de secuencia para el caso de uso `ReportarEmergencia` (inicio desde el lado de `EstaciónOficialCampo`).

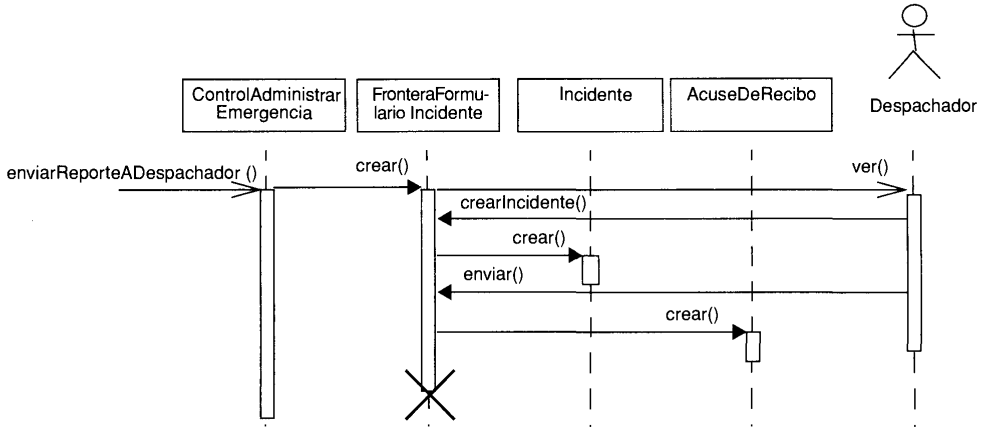


Figura 5-13 Diagrama de secuencia para el caso de uso `ReportarEmergencia` (`EstaciónDespachador`).

En la figura 5-13 descubrimos el objeto de entidad `AcuseDeRecibo` que olvidamos durante nuestro examen inicial del caso de uso `ReportarEmergencia` (en la tabla 5-2). El objeto `AcuseDeRecibo` es diferente de una `NoticiaAcuseDeRecibo`. Guarda la información asociada con un `AcuseDeRecibo` y se crea antes del objeto de frontera `NoticiaAcuseDeRecibo`. Cuando describimos al objeto `AcuseDeRecibo`, también nos damos cuenta que el caso de uso `ReportarEmergencia`

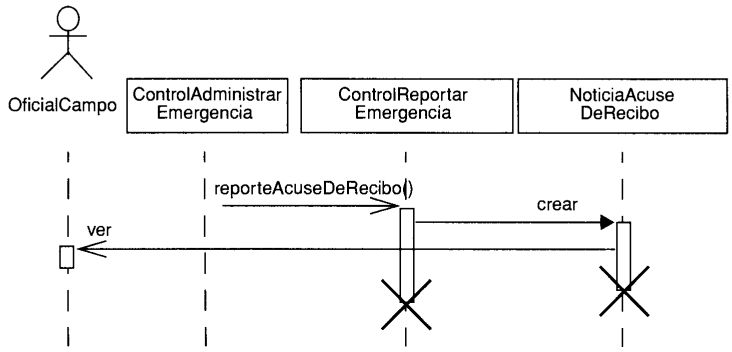


Figura 5-14 Diagrama de secuencia para el caso de uso `ReportarEmergencia` (acuse de recibo para la `EstaciónOficialCampo`).

original (descrito en la figura 5-11) está incompleto. Sólo menciona la existencia de un `AcuseDeRecibo` y no describe la información que tiene asociada. En este caso, los desarrolladores necesitan aclararlo con el cliente para definir la información que necesita aparecer en el `AcuseDeRecibo`. Después de obtener las aclaraciones se añade el objeto `AcuseDeRecibo` al modelo de análisis (tabla 5-5) y se aclara el caso de uso `ReportarEmergencia` para que incluya la información adicional (figura 5-15).

Tabla 5-5 Objeto `AcuseDeRecibo` para el caso de uso `ReportarEmergencia`.

`AcuseDeRecibo`	Es la respuesta de un despachador a un `ReporteDeEmergencia` de un `OficialCampo`. Al enviar un `AcuseDeRecibo`, el `Despachador` comunica al `OficialCampo` que ha recibido el `ReporteDeEmergencia`, ha creado un `Incidente` y le ha asignado recursos. El `AcuseDeRecibo` contiene los recursos asignados y el tiempo de llegada estimado.

Mediante la construcción de diagramas de secuencia no sólo modelamos el orden de la interacción entre objetos sino que también distribuimos el comportamiento del caso de uso. En otras palabras, asignamos responsabilidades a cada objeto en forma de un conjunto de operaciones. Estas operaciones pueden ser compartidas por cualquier caso de uso en donde participe un objeto dado. Observe que la definición de un objeto que está compartido entre dos o más casos de uso debe ser idéntica. En otras palabras, si una operación aparece en más de un diagrama de secuencia, su comportamiento debe ser el mismo.

Compartir operaciones entre casos de uso permite que los desarrolladores eliminen redundancias en la especificación del sistema y mejoren su consistencia. Observe que siempre se debe dar precedencia a la claridad para eliminar redundancias. La fragmentación del comportamiento entre muchas operaciones complica en forma innecesaria la especificación del sistema.

En el análisis, los diagramas de secuencia se usan para que ayuden a identificar nuevos objetos participantes y comportamiento faltante. Se enfocan en comportamiento de alto nivel y, por lo tanto, los asuntos de implementación, como el desempeño, no deben tratarse en este punto. Tomando

Nombre del caso de uso	`ReportarEmergencia`
Condición inicial	1. El `OficialCampo` activa la función "Reportar Emergencia" de su terminal.
Flujo de eventos	2. FRIEND responde presentando un formulario al oficial. El formulario incluye un menú de tipo de emergencia (emergencia general, incendio, transporte), y campos para ubicación, descripción del incidente, petición de recursos y materiales peligrosos.
	3. El `OficialCampo` llena el formulario, seleccionando el nivel de emergencia, tipo, ubicación y una breve descripción de la situación. El `OficialCampo` también describe respuestas posibles a la situación de emergencia y solicita recursos específicos. Una vez que ha llenado el formulario, el `OficialCampo` lo envía oprimiendo el botón "Enviar Reporte" y en ese momento se le notifica al `Despachador`.
	4. El `Despachador` revisa la información enviada por el `OficialCampo` y crea un `Incidente` en la base de datos llamando al caso de uso `AbrirIncidente`. En el incidente se incluye de manera automática toda la información contenida en el formulario del `OficialCampo`. El `Despachador` selecciona una respuesta asignando recursos al incidente (con el caso de uso `AsignarRecursos`) y da un acuse de recibo del reporte de emergencia enviando un FRIENDgrama al `OficialCampo`. **El `AcuseDeRecibo` indica al `OficialCampo` que se recibió el `ReporteDeEmergencia`, se creó un `Incidente` y se asignaron recursos al `Incidente`. El `AcuseDeRecibo` incluye los recursos (por ejemplo, un camión de bomberos) y su tiempo de llegada estimado.**
Condición de salida	5. El `OficialCampo` recibe el acuse de recibo y la respuesta seleccionada.

Figura 5-15 Caso de uso `ReportarEmergencia` refinado. El descubrimiento y la adición del objeto `AcuseDeRecibo` al modelo de análisis revela que el caso de uso `ReportarEmergencia` original no describió con precisión la información asociada con `AcuseDeRecibo`. Los refinamiento se indican en negritas.

en cuenta que la construcción de diagramas de interacción puede consumir mucho tiempo, los desarrolladores deben enfocarse primero en la funcionalidad problemática o no especificada. El trazo de diagramas de interacción para partes del sistema que son simples o están bien definidas no es una buena inversión de los recursos de análisis.

5.4.5 Identificación de asociaciones

Mientras que los diagramas de secuencia permiten que los desarrolladores representen interacciones entre objetos a lo largo del tiempo, los diagramas de clase permiten que los desarrolladores describan la conectividad espacial de los objetos. En el capítulo 2, *Modelado con UML*, describimos la notación de los diagramas de clase UML, y la usamos a lo largo del libro para representar diversos artefactos del proyecto (por ejemplo, actividades, cosas a entregar). En esta sección tratamos el uso de diagramas de clase para representar asociaciones entre objetos. En la sección 5.4.6 tratamos el uso de los diagramas de clase para representar atributos de objetos.

Una asociación muestra una relación entre dos o más clases. Por ejemplo, un `OficialCampo` escribe un `ReporteDeEmergencia` (vea la figura 5-16). La identificación de asociaciones tiene dos ventajas. Primero, aclara el modelo de análisis haciendo explícitas las relaciones entre objetos (por ejemplo, un `ReporteDeEmergencia` puede ser creado por un `OficialCampo` o un `Despachador`). Segundo, permite que el desarrollador descubra casos de frontera asociados con vínculos. Los casos de frontera son excepciones que es necesario aclarar en el modelo. Por ejemplo, es intuitivo asumir que la mayoría de los `ReporteDeEmergencia` son escritos por un `OficialCampo`. Sin embargo, ¿debe soportar el sistema los `ReporteDeEmergencia` escritos por más de uno? ¿Debe permitir el sistema `ReporteDeEmergencia` anónimos? Estas preguntas deben investigarse durante el análisis usando el conocimiento del dominio.

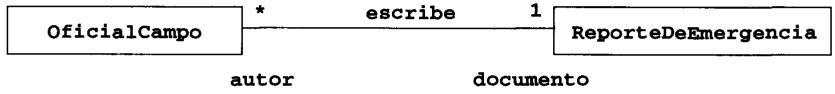


Figura 5-16 Un ejemplo de asociación entre las clases `ReporteDeEmergencia` y `OficialCampo`.

Las asociaciones tienen varias propiedades:

- Un **nombre** para describir la asociación entre las dos clases (por ejemplo, `escribe` en la figura 5-16). Los nombres de asociación son opcionales y no es necesario que sean únicos en forma global.
- Un **papel** a cada extremo, que identifica la operación de cada clase con respecto a las asociaciones (por ejemplo, `autor` es el papel desempeñado por `OficialCampo` en la asociación `escribe`).
- Una **multiplicidad** a cada extremo, que identifica la cantidad posible de instancias (por ejemplo, * indica que un `OficialCampo` puede escribir cero o más `ReporteDeEmergencia`, mientras que `1` indica que cada `ReporteDeEmergencia` tiene exactamente un `OficialCampo` como autor.

Al principio, las asociaciones entre objetos de entidad son lo más importante, ya que revelan más información acerca del dominio de aplicación. De acuerdo con la heurística de Abbott (vea la tabla 5-1), las asociaciones pueden identificarse examinando verbos y frases verbales que indican un estado (por ejemplo, *tiene*, *es parte de*, *administra*, *reporta a*, *es activado por*, *está contenido en*, *habla a*, *incluye*). Cada asociación debe tener un nombre y papeles asignados a cada extremo.

Heurística para la identificación de asociaciones

- Examine las frases verbales.
- Nombre con precisión a las asociaciones y papeles.
- Use calificadores con tanta frecuencia como sea posible para identificar espacios de nombres y atributos principales.
- Elimine cualquier asociación que pueda derivarse de otras asociaciones.
- No se preocupe por la multiplicidad hasta que el conjunto de asociaciones sea estable.
- Evite modelos espagueti: demasiadas asociaciones hacen que el modelo sea ilegible.

El modelo de objetos incluirá al principio demasiadas asociaciones si los desarrolladores incluyen todas las asociaciones identificadas después de examinar las frases verbales. En la figura 5-17, por ejemplo, identificamos dos relaciones: la primera entre un `Incidente` y el `ReporteDeEmergencia` que activó su creación, y la segunda entre el `Incidente` y el `OficialCampo` que lo reportó. Tomando en cuenta que `ReporteDeEmergencia` y `OficialCampo` ya tienen un autor de modelado de asociación, la asociación entre `Incidente` y `OficialCampo` no es necesaria. La adición de asociaciones innecesarias complica el modelo conduciendo a "modelos espagueti" incomprensibles e información redundante.

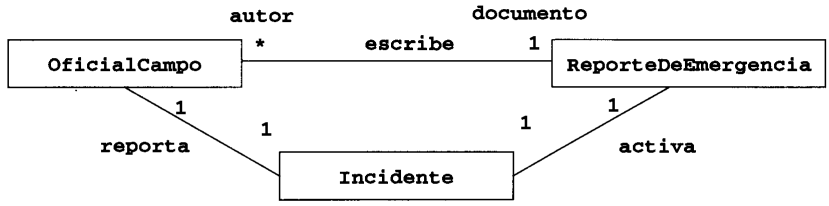


Figura 5-17 Eliminación de asociación redundante. La recepción de un `ReporteDeEmergencia` activa la creación de un `Incidente` por parte de un `Despachador`. Tomando en cuenta que `ReporteDeEmergencia` tiene una asociación con el `OficialCampo` que lo escribe, no es necesario mantener una asociación entre `OficialCampo` e `Incidente`.

La mayoría de los objetos de entidad tienen una característica identificadora que usan los actores para tener acceso a ellos. Los `OficialCampo` y los `Despachador` tienen un número de identificación. Los `Incidente` y `Reporte` tienen números asignados y se archivan por fecha. Una vez que el modelo de análisis incluye a la mayoría de las clases y asociaciones, los desarrolladores deben revisar cada clase para ver la manera en que es identificada por los actores y en cuál contexto. Por ejemplo, ¿son únicos los números de identificación de los `OficialCampo` para todo el universo? ¿Para toda la ciudad? ¿En una estación de policía? Si son únicos entre ciudades, ¿el sistema FRIEND puede saber acerca de `OficialCampo` de más de una ciudad? Este enfoque puede formalizarse examinando cada clase individual, identificando la secuencia de asociaciones que se necesita recorrer para tener acceso a una instancia específica de esa clase.

5.4.6 Identificación de atributos

Los atributos son propiedades de objetos individuales. Por ejemplo, un `ReporteDeEmergencia`, como se describe en la tabla 5-2, tiene propiedades de tipo de emergencia, ubicación y descripción (vea la figura 5-18). Éstas son proporcionadas por el `OficialCampo` cuando reporta una emergencia y son mantenidas en adelante por el sistema. Cuando se identifican propiedades de objetos, sólo deben considerarse los atributos relevantes para el sistema. Por ejemplo, cada `OficialCampo` tiene un número de seguro social que no es relevante para el sistema de información de emergencia. En vez de ello, los `OficialCampo` están identificados por un número de identificación representado por la propiedad `NúmeroIdentificación`.

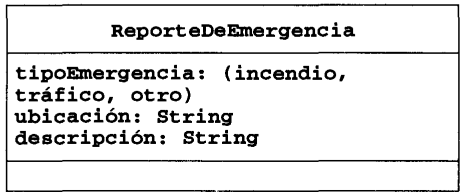


Figura 5-18 Atributos de la clase `ReporteDeEmergencia`.

Las propiedades que están representadas por objetos no son atributos. Por ejemplo, todo `ReporteDeEmergencia` tiene un autor representado por la asociación con la clase `OficialCampo`. Los desarrolladores deben identificar la mayor cantidad de asociaciones posible antes de identificar atributos para evitar confusiones entre atributos y objetos. Los atributos tienen:

- Un **nombre** que los identifica dentro de un objeto. Por ejemplo, un `ReporteDeEmergencia` puede tener un atributo `tipoReporte` y un atributo `tipoEmergencia`. El `tipoReporte` describe el tipo de reporte que se está registrando (por ejemplo, reporte inicial, petición de recursos, reporte final). El `tipoEmergencia` describe el tipo de emergencia (por ejemplo, incendio, tráfico, otro). Para evitar confusiones no se debe llamar `tipo` a estos dos atributos.
- Una **descripción** breve.
- Un **tipo** que describe los valores legales que puede adoptar. Por ejemplo, el atributo `descripción` de un `ReporteDeEmergencia` es una cadena. El atributo `tipoEmergencia` es una enumeración que puede adoptar uno de tres valores: `incendio`, `tráfico` u `otro`.

Los atributos pueden identificarse usando la heurística de Abbott (vea la tabla 5-1). En particular, deben examinarse las frases nominales que están seguidas por frases posesivas (por ejemplo, la descripción de una emergencia) o una frase adjetiva (por ejemplo, descripción de emergencia). En el caso de objetos de entidad, cualquier propiedad que necesite guardarse en el sistema es un atributo candidato.

Observe que los atributos representan la parte menos estable del modelo de objetos. Con frecuencia los atributos se descubren o añaden más adelante en el desarrollo cuando los usuarios evalúan el sistema. A menos que los atributos añadidos estén asociados con funcionalidad adi-

Heurística para la identificación de atributos[a]

- Examine las frases posesivas.
- Represente el estado guardado como atributo de un objeto de entidad.
- Describa cada atributo.
- No represente un atributo como un objeto; en vez de ello use una asociación (vea la sección 5.4.5).
- No pierda tiempo describiendo detalles finos antes de que sea estable la estructura del objeto.

a. Adaptado de [Rumbaugh *et al.*, 1991].

cional, no implican cambios importantes en la estructura del objeto (ni del sistema). Por estas razones, los desarrolladores no necesitan gastar recursos excesivos identificando y detallando los atributos que representan aspectos menos importantes del sistema. Estos atributos pueden añadirse después cuando se valida el modelo de análisis o los bosquejos de interfaz de usuario.

5.4.7 Modelado del comportamiento no trivial de objetos individuales

Los diagramas de secuencia se usan para distribuir comportamiento entre objetos y para identificar operaciones. Los diagramas de secuencia representan el comportamiento del sistema desde la perspectiva de un solo caso de uso. Los diagramas de gráfica de estado representan el comportamiento desde la perspectiva de un solo objeto. La visión del comportamiento desde la perspectiva de cada objeto permite que el desarrollador, por un lado, identifique casos de uso faltantes y, por otro, construya una descripción más formal del comportamiento del objeto. Observe que no es necesario construir gráficas de estado para cada una de las clases del sistema. Sólo vale la pena construir gráficas de estado de los objetos que tienen una vida extendida y un comportamiento no trivial.

La figura 5-19 muestra una gráfica de estado para la clase `Incidente`. El examen de esta gráfica de estado puede ayudar para que el desarrollador revise si hay casos de uso para la documentación, el cierre y el archivado del `Incidente`. Observe que la figura 5-19 es una gráfica de estado de alto nivel y no modela los cambios de estado por los que pasa el `Incidente` mientras está activo (por ejemplo, cuando se le están asignando recursos). Tal comportamiento puede modelarse asociando una gráfica de estado anidada con el estado `Activo`.

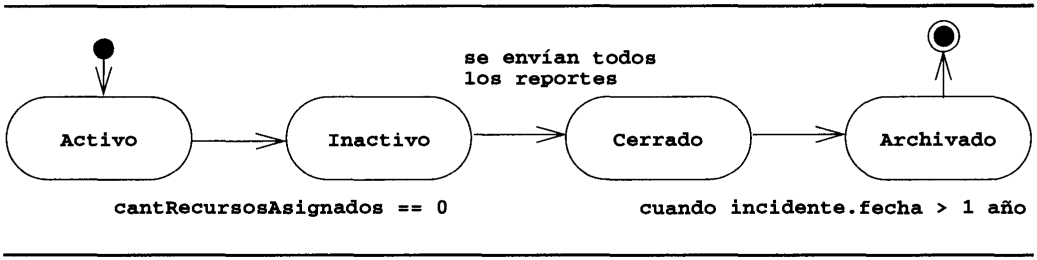


Figura 5-19 Gráfica de estado UML para `Incidente`.

5.4.8 Modelado de las relaciones de generalización entre objetos

La generalización se usa para eliminar redundancia en el modelo de análisis. Si dos o más clases comparten atributos o comportamientos, las similitudes se consolidan en una superclase. Por ejemplo, `Despachador` y `OficialCampo` tienen un atributo, `númeroIdentificación`, que sirve para identificarlos dentro de una ciudad. Los `OficialCampo` y `Despachador` son `OficialPolicía` que tienen asignadas funciones diferentes. Para modelar de manera explícita esta similitud introducimos una clase abstracta, *`OficialPolicía`*, de la cual heredan las clases `OficialCampo` y `Despachador` (vea la figura 5-20).

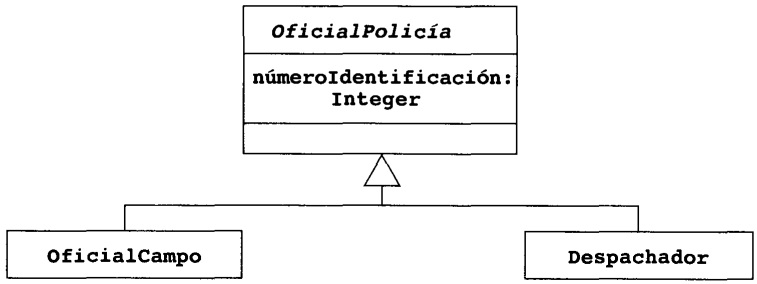


Figura 5-20 Un ejemplo de relación de herencia (diagrama de clase UML).

5.4.9 Revisión del modelo de análisis

El modelo de análisis se construye en forma incremental e iterativa. El modelo de análisis rara vez es correcto o completo en el primer paso. Se necesitan varias iteraciones con el cliente y el usuario antes de que el modelo de análisis converja hacia una especificación correcta que puedan utilizar los desarrolladores para continuar con el diseño y la implementación. Por ejemplo, una omisión descubierta durante el análisis conducirá a la adición o extensión de un caso de uso en la especificación del sistema, la cual puede conducir a la obtención de más información del usuario.

Una vez que el modelo de análisis llega a ser estable (es decir, cuando la cantidad de cambios al modelo es mínima y está localizado el alcance de los cambios) se revisa el modelo de análisis, primero entre los desarrolladores (es decir, revisiones internas) y luego en forma conjunta los desarrolladores con el cliente. El objetivo de la revisión es asegurarse que la especificación del sistema sea correcta, completa, consistente y realista. Observe que los desarrolladores deben estar preparados para descubrir errores más adelante y hacer cambios a la especificación. Sin embargo, vale la pena la inversión para atrapar al inicio la mayor cantidad de errores de requerimientos. La revisión puede facilitarse con una lista de revisión o una lista de preguntas. A continuación se tienen preguntas de ejemplo adaptadas de [Jacobson *et al.*, 1999] y [Rumbaugh *et al.*, 1991].

Deberán hacerse las siguientes preguntas para asegurarse que el modelo sea *correcto*:

- ¿El glosario de objetos de entidad es comprensible para el usuario?
- ¿Las clases abstractas corresponden a conceptos en el nivel de usuario?
- ¿Todas las descripciones están de acuerdo con las definiciones de los usuarios?
- ¿Todos los objetos de entidad y frontera tienen como nombre frases nominativas significativas?
- ¿Todos los casos de uso y objetos de control tienen como nombre frases verbales significativas?
- ¿Todos los casos de error están descritos y manejados?
- ¿Están descritas las fases de inicio y terminación del sistema?
- ¿Están descritas las funciones de administración del sistema?

Deberán hacerse las siguientes preguntas para asegurarse que el modelo esté *completo*:

- Para cada objeto: ¿Lo necesita algún caso de uso? ¿En qué caso de uso se crea? ¿En cuál se modifica? ¿En cuál se destruye? ¿Se puede tener acceso a él desde un objeto de frontera?

- Para cada atributo: ¿Cuándo se asigna? ¿Cuál es su tipo? ¿Debe ser un calificador?
- Para cada asociación: ¿Cuándo se le recorre? ¿Por qué se escogió esa multiplicidad específica? ¿Pueden calificarse las asociaciones con multiplicidades de uno a muchos o de muchos a muchos?
- Para cada objeto de control: ¿Tiene las asociaciones necesarias para tener acceso a los objetos que participan en su caso de uso correspondiente?

Deberán hacerse las siguientes preguntas para asegurarse que el modelo sea *consistente*:

- ¿Hay varias clases o casos de uso con el mismo nombre?
- ¿Las entidades (por ejemplo, casos de uso, clases, atributos) con nombres similares indican fenómenos similares?
- ¿Están descritas todas las entidades con el mismo nivel de detalle?
- ¿Hay objetos con atributos y asociaciones similares que no están en la misma jerarquía de generalización?

Deberán hacerse las siguientes preguntas para asegurarse que el sistema descrito por el modelo de análisis sea *realista*:

- ¿Hay alguna característica novedosa en el sistema? ¿En dónde se han realizado estudios o prototipos para asegurar su factibilidad?
- ¿Pueden satisfacerse los requerimientos de desempeño y confiabilidad? ¿Se verificaron esos requerimientos mediante algún prototipo ejecutado en el hardware seleccionado?

5.4.10 Resumen del análisis

La actividad de requerimientos es sumamente iterativa e incremental. Se bosquejan y proponen bloques de funcionalidad a los usuarios y clientes. El cliente añade requerimientos adicionales, critica la funcionalidad existente y modifica los requerimientos existentes. Los desarrolladores investigan los requerimientos no funcionales mediante la elaboración de prototipos y estudios de tecnología, y ponen a prueba cada requerimiento propuesto. Al principio, la obtención de requerimientos se asemeja a una actividad de lluvia de ideas. Conforme crece la descripción del sistema y los requerimientos se hacen más concretos, los desarrolladores necesitan extender y modificar el modelo de análisis de forma más ordenada para administrar la complejidad de la información.

La figura 5-21 muestra una secuencia típica de las actividades de análisis que describimos en este capítulo. Los usuarios, desarrolladores y clientes están involucrados en la *definición de casos de uso* y desarrollan un modelo de caso de uso inicial. Identifican varios conceptos y construyen un glosario de objetos participantes. Luego los desarrolladores clasifican estos objetos en objetos de entidad, frontera y control (en *definir objetos de entidad*, sección 5.4.1, *definir objetos de frontera*, sección 5.4.2, y *definir objetos de control*, sección 5.4.3). Estas actividades suceden en un ciclo apretado hasta que la mayor parte de la funcionalidad del sistema ha sido identificada como casos de uso con nombres y descripciones breves. Luego los desarrolladores construyen diagramas de secuencia para identificar cualquier objeto faltante (*definir interacciones*, sección 5.4.4). Una vez que han sido nombrados y descritos en forma breve todos los objetos de identidad, el modelo de análisis debe permanecer bastante estable conforme se le refina.

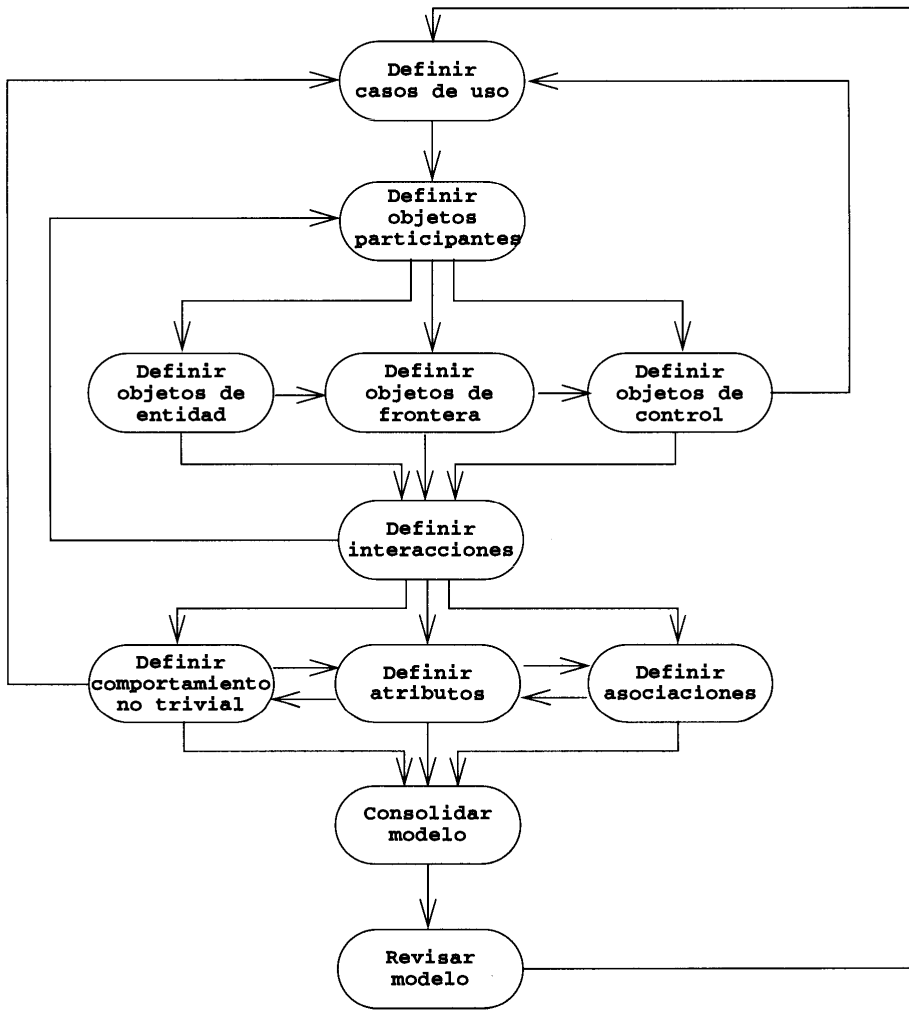


Figura 5-21 Actividades de análisis (diagrama de actividades UML).

Definir comportamiento interesante (sección 5.4.7), *definir atributos* (sección 5.4.6) y *definir asociaciones* (sección 5.4.5) constituyen el refinamiento del modelo de análisis. Estas tres actividades representan un ciclo apretado durante el cual se extraen y se detallan el estado de los objetos y sus asociaciones de los diagramas de secuencia. Luego se modifican los casos de uso para que tomen en cuenta cualquier cambio en la funcionalidad. Esta fase puede conducir a la identificación de bloques adicionales de funcionalidad en forma de casos de uso adicionales. Luego se repite el proceso general en forma incremental para estos nuevos casos de uso.

Durante *consolidar modelo* (sección 5.4.8), los desarrolladores solidifican el modelo introduciendo calificadores, relaciones de generalización y suprimiendo redundancias. Durante *revisar modelo* (sección 5.4.9), el cliente, usuarios y desarrolladores examinan el modelo para que haya en él corrección, consistencia, suficiencia y realismo. La calendarización del proyecto deberá planear varias revisiones para asegurar la alta calidad de los requerimientos y permitir espacio para el aprendizaje de la actividad de requerimientos. Sin embargo, una vez que el modelo llega al punto en donde la mayoría de las modificaciones son cosméticas, debe continuarse con el diseño del sistema. Habrá un punto durante los requerimientos en donde ya no se puedan anticipar más problemas sin información adicional de los prototipos, estudios de utilidad, investigaciones de tecnología o diseño de sistema. Hacer que todos los detalles estén correctos llega a ser un ejercicio inútil: algunos de esos detalles serán irrelevantes con el siguiente cambio. La administración debe reconocer este punto e iniciar la siguiente fase del proyecto.

5.5 Administración del análisis

En esta sección tratamos asuntos relacionados con la administración de las actividades de análisis en un proyecto de desarrollo de varios equipos. El reto principal de la administración de los requerimientos en estos proyectos es mantener la consistencia mientras se usan tantos recursos. Al final, el documento de análisis de requerimientos debe describir un solo sistema coherente que sea comprensible para una sola persona.

Primero describimos una plantilla de documento que puede usarse para documentar los resultados del análisis (sección 5.5.1). Luego describimos la asignación de papeles para el análisis (sección 5.5.2). Luego tratamos los asuntos de comunicación durante el análisis. A continuación tratamos los asuntos de administración relacionados con la naturaleza iterativa e incremental de los requerimientos (sección 5.5.4).

5.5.1 Documentación del análisis

Como vimos en el capítulo anterior, las actividades de obtención de requerimientos y análisis se documentan en el documento de análisis de requerimientos (RAD, por sus siglas en inglés). Las secciones 1 a 3.5.2 ya se han escrito durante la obtención de requerimientos. Durante el análisis revisamos estas secciones conforme se descubren ambigüedades y nueva funcionalidad. Sin embargo, el esfuerzo principal se enfoca en la escritura de las secciones que documentan el modelo de objetos de análisis (3.5.3 y 3.5.4).

La sección 3.5.3, Modelo de objetos, documenta con detalle todos los objetos que identificamos, sus atributos y, en caso que usemos diagramas de secuencia, sus operaciones. Conforme se describe cada objeto con definiciones textuales, las relaciones entre objetos se ilustran con diagramas de clase.

La sección 3.5.4, Modelos dinámicos, documenta el comportamiento del modelo de objeto desde el punto de vista de diagramas de gráfica de estado y diagramas de secuencia. Aunque esta información es redundante con el modelo de casos de uso, los modelos dinámicos nos permiten representar comportamientos complejos con mayor precisión, incluyendo casos de uso que involucran a varios actores.

El RAD, una vez terminado y publicado, será considerado como una línea base y se pondrá bajo la administración de la configuración. La sección de historia de revisiones del RAD proporcionará una historia de los cambios en forma de una lista del autor responsable del cambio, la fecha del cambio y una breve descripción del cambio.

Documento de análisis de requerimientos

1. Introducción
2. Sistema actual
3. Sistema propuesto
   - 3.1 Panorama
   - 3.2 Requerimientos funcionales
   - 3.3 Requerimientos no funcionales
   - 3.4 Seudorrequerimientos
   - 3.5 Modelos del sistema
     - 3.5.1 Escenarios
     - 3.5.2 Modelo de casos de uso
     - 3.5.3 Modelo de objetos
       - 3.5.3.1 Diccionario de datos
       - 3.5.3.2 Diagramas de clase
     - 3.5.4 Modelos dinámicos
     - 3.5.5 Interfaz de usuario: rutas de navegación y maquetas de pantallas
4. Glosario

5.5.2 Asignación de responsabilidades

El análisis requiere la participación de un amplio rango de individuos. El usuario de destino proporciona conocimiento sobre el dominio de aplicación. El cliente proporciona los fondos para el proyecto y coordina el esfuerzo del lado del usuario. El analista obtiene conocimiento del dominio de aplicación y lo formaliza. Los desarrolladores proporcionan retroalimentación sobre factibilidad y costo. El gerente del proyecto coordina el esfuerzo del lado del desarrollo. En grandes sistemas pueden estar involucrados muchos usuarios, analistas y desarrolladores, introduciendo retos adicionales para los requerimientos de integración y comunicación del proyecto. Estos retos pueden satisfacerse asignando papeles y alcances bien definidos a los individuos. Hay tres tipos principales de papeles: generación de información, integración y revisión.

- El **usuario** es el experto en el dominio de aplicación, quien genera información acerca del sistema actual, el ambiente del sistema futuro y las tareas que debe soportar. Cada usuario corresponde a uno o más actores y ayuda a identificar sus casos de uso asociados.
- El **cliente**, un papel de integración, define el alcance del sistema con base en los requerimientos del usuario. Diferentes usuarios pueden tener diferentes vistas del sistema, ya sea debido a que se beneficiarán de diferentes partes del sistema (por ejemplo, un despachador frente a un oficial de campo) o debido a que los usuarios tienen opiniones o expectativas diferentes acerca del sistema futuro. El cliente sirve como integrador de la información del dominio de aplicación y resuelve inconsistencias en las expectativas del usuario.
- El **analista** es el experto del dominio de desarrollo, quien modela el sistema actual y genera información acerca del sistema futuro. Al inicio cada analista es responsable de detallar uno o más casos de uso. Para un conjunto de casos de uso, el análisis identificará una

cantidad de objetos, sus asociaciones y sus atributos usando las técnicas que se indicaron en la sección 5.4.

- El **arquitecto**, un papel de integración, unifica los modelos de caso de uso y de objetos desde un punto de vista del sistema. Diferentes analistas pueden tener diferentes estilos de modelado y diferentes vistas de partes del sistema de las cuales no son responsables. Aunque los analistas trabajan juntos y es muy probable que resuelvan las diferencias conforme avanzan en el análisis, el papel del arquitecto es necesario para proporcionar una filosofía del sistema y para identificar omisiones en los requerimientos.
- El **editor de documentos** es responsable de la integración de bajo nivel del documento. El editor de documentos es responsable del formato general del documento y de su índice.
- El **gerente de configuración** es responsable del mantenimiento de la historia de revisiones del documento, así como de la información de rastreabilidad que relaciona al RAD con otros documentos (como el documento de diseño del sistema, vea el capítulo 6, *Diseño del sistema).*
- El **revisor** valida el RAD para que sea correcto, completo, consistente, realista, verificable y rastreable. Los usuarios, clientes, desarrolladores u otros individuos pueden llegar a ser revisores durante la validación de requerimientos. Los individuos que todavía no han estado involucrados en el desarrollo representan revisores excelentes, debido a que son más capaces de identificar ambigüedades y áreas que necesitan aclaración.

El tamaño del sistema determina la cantidad de usuarios y analistas diferentes que se necesitan para obtener y modelar los requerimientos. En todos los casos deberá haber un papel integrador del lado del cliente y otro del lado del desarrollo. Al final, sin importar qué tan grande sea el sistema, los requerimientos deberán ser comprensibles para un solo individuo que tenga conocimiento del dominio de aplicación.

5.5.3 Comunicación acerca del análisis

La tarea de comunicar información es la más desafiante durante la obtención de requerimientos y el análisis. Los factores que contribuyen comprenden:

- *Diferentes conocimientos de los participantes.* Los usuarios, clientes y desarrolladores tienen diferentes dominios de experiencia y usan vocabularios diferentes para describir los mismos conceptos.
- *Diferentes expectativas de los participantes.* Los usuarios, clientes y administradores tienen objetivos diferentes cuando definen el sistema. Los usuarios quieren un sistema que soporte sus procesos de trabajo actual sin interferir o amenazar su posición actual (por ejemplo, un sistema mejorado con frecuencia se traduce en la eliminación de posiciones actuales). El cliente quiere maximizar el rendimiento de su inversión. La administración quiere entregar el sistema a tiempo. Diferentes expectativas y diferentes personas involucradas en el proyecto pueden conducir a una resistencia a compartir información y a reportar problemas en forma ordenada.

- *Nuevos equipos.* La obtención de requerimientos y el análisis a menudo marcan el inicio de un nuevo proyecto. Esto se traduce en nuevos participantes y nuevas asignaciones de equipos y, por lo tanto, en un periodo cuesta arriba durante el cual los miembros del equipo deben aprender a trabajar juntos.
- *Sistema en evolución.* Cuando se desarrolla un nuevo sistema a partir de cero, los términos y conceptos relacionados con el nuevo sistema están fluyendo durante la mayor parte del análisis y el diseño del sistema. Un término puede tener hoy un significado que cambiará mañana.

Ningún método de requerimientos o mecanismo de comunicación puede resolver los problemas relacionados con las políticas internas y la ocultación de información. Los objetivos en conflicto y la competencia siempre serán parte de los proyectos de desarrollo grandes. Sin embargo, unos cuantos lineamientos pueden ayudar a manejar la complejidad de vistas conflictivas del sistema:

- *Definir territorios claros.* La definición de papeles, como se describe en la sección 5.5.2, es parte de esta actividad. Esto también incluye la definición de foros de discusión privados y públicos. Por ejemplo, cada equipo puede tener una base de datos de discusión como se describe en el capítulo 3, *Comunicación de proyectos*, y la discusión con el cliente se realiza en una base de datos para clientes separada. El cliente no debe tener acceso a la base de datos interna. Del mismo modo, los desarrolladores no deben interferir con las políticas internas de clientes y usuarios.
- *Definir objetivos claros y criterios de éxito.* La definición conjunta por parte del cliente y los desarrolladores de objetivos claros, mensurables y verificables, y de criterios de éxito, facilita la resolución de conflictos. Observe que la definición de un objetivo claro y verificable no es una tarea trivial, tomando en cuenta que es más fácil dejar abiertos los objetivos. Los objetivos y el criterio de éxito del proyecto deben estar documentados en la sección 1.3 del RAD.
- *Lluvia de ideas.* Poner a todos los interesados juntos en el mismo salón y hacer que generen con rapidez soluciones y definiciones puede eliminar muchas barreras en la comunicación. La realización de entrevistas como actividad recíproca (es decir, la revisión por parte del cliente y los desarrolladores, durante la misma sesión, de los productos a entregar) tiene un efecto similar.

La lluvia de ideas y, en términos más generales, el desarrollo cooperativo de los requerimientos puede conducir a la definición de notaciones compartidas *ad hoc* para soportar la comunicación. Los guiones sinópticos, los bosquejos de interfaz de usuario y los diagramas de flujo de datos de alto nivel con frecuencia aparecen de modo espontáneo. Conforme aumenta la información acerca del dominio de aplicación y el nuevo sistema, es crítico que se utilice una notación precisa y estructurada. En UML los desarrolladores emplean casos de uso y escenarios para comunicarse con el cliente y los usuarios, y usan diagramas de objetos, diagramas de secuencia y gráficas de estado para comunicarse con los demás desarrolladores (vea las secciones 4.4 y 5.4). Además, la versión más reciente de los requerimientos deberá estar disponible para todos los participantes. El mantenimiento en línea de una versión viva del documento de análisis de requerimientos, con una historia de cambios actualizada, facilita la propagación a tiempo de los cambios a lo largo de todo el proyecto.

5.5.4 Iteración por el modelo de análisis

El análisis sucede en forma iterativa e incremental, con frecuencia en forma paralela a las demás actividades del desarrollo, como el diseño del sistema y la implementación. Sin embargo, observe que la modificación y extensión irrestricta del modelo de análisis sólo puede producir caos, en especial cuando está involucrada una gran cantidad de participantes. Las iteraciones e incrementos necesitan administrarse con cuidado, y se debe dar seguimiento a las peticiones de cambio una vez que se han definido los requerimientos. La actividad de requerimientos puede verse como una serie de pasos (lluvia de ideas, solidificación, madurez) que convergen hacia un modelo estable.

Lluvia de ideas

Antes de que se inicie cualquier otra actividad de desarrollo, los requerimientos son un proceso de lluvia de ideas. Todo cambia: los conceptos y los términos usados para referirse a ellos. El objetivo de un proceso de lluvia de ideas es generar la mayor cantidad posible de ideas sin tener necesariamente que organizarlas. Durante esta etapa las iteraciones son rápidas y de largo alcance.

Solidificación

Una vez que el cliente y los desarrolladores convergen en una idea común, definen las fronteras del sistema y se ponen de acuerdo en un conjunto de términos estándar, comienza la solidificación. La funcionalidad se inicia en grupos de caso de uso con sus interfaces correspondientes. Los grupos de funcionalidad se asignan a diferentes equipos que son responsables de detallar sus casos de uso correspondientes. Durante esta etapa las iteraciones son rápidas pero localizadas.

Madurez

Todavía son posibles los cambios de alto nivel, pero más difíciles, y por lo tanto se realizan con mayor cuidado. Cada equipo es responsable de los casos de uso y modelos de objetos relacionados con la funcionalidad que se les ha asignado. Un equipo de funcionalidad cruzada, el equipo de arquitectura, compuesto por representantes de cada equipo, es responsable de asegurar la integración de los requerimientos (por ejemplo, denominación).

Una vez que el cliente acepta los requerimientos, las modificaciones al modelo de análisis deben ser sobre las omisiones y errores. Los desarrolladores, en particular el equipo de arquitectura, necesitan asegurarse que no se comprometa la consistencia del modelo. El modelo de requerimientos está bajo la administración de configuración y los cambios deben propagarse a los modelos de diseño existentes. Las iteraciones son rápidas pero se localizan.

Siempre se incrementará con el tiempo la cantidad de características y funciones de un sistema. Sin embargo, cada cambio puede amenazar la integridad del sistema. El riesgo de introducir más problemas con cambios tardíos puede dar como resultado la pérdida de información en el proyecto. Las dependencias entre funciones no se han capturado del todo y muchas suposiciones pueden estar implícitas y olvidadas para cuando se hace el cambio. A menudo el cambio responde a problemas y, en ese caso, hay mucha presión para implementarlo, dando como resultado sólo un examen superficial de las consecuencias del cambio. Cuando se añaden características y funciones nuevas al sistema deberán ponerse en tela de juicio con las siguientes preguntas: ¿Las solicitó el cliente? ¿Son necesarias o son adornos? ¿Deben ser parte de un programa de utilidad separado y enfocado en vez de ser parte del sistema básico? ¿Cuál es el impacto de los cambios en las funcio-

nes existentes desde el punto de vista de la consistencia, la interfaz y la confiabilidad? ¿Los cambios son requerimientos medulares o características opcionales?

Cuando los cambios son necesarios, el cliente y el desarrollador definen el alcance del cambio y su resultado deseado, y cambian el modelo de análisis. Tomando en cuenta que ya existe un modelo de análisis completo para el sistema, es fácil la especificación de nueva funcionalidad (aunque su implementación es más difícil).

5.5.5 Aprobación del cliente

La aprobación del cliente representa la aceptación del modelo de análisis (como está documentada en el documento de análisis de requerimientos) por parte del cliente. El cliente y los desarrolladores convergen en una sola idea y se ponen de acuerdo acerca de las funciones y características que tendrá el sistema. Además se ponen de acuerdo en:

- Una lista de prioridades
- Un proceso de revisión
- Una lista de criterios que se usarán para aceptar o rechazar el sistema
- Una calendarización y un presupuesto

El establecimiento de la prioridad de las funciones del sistema permite que los desarrolladores comprendan mejor las expectativas del cliente. En su forma más simple permite que los desarrolladores separen los adornos de las características esenciales del sistema. En general, permite que los desarrolladores entreguen el sistema en fragmentos incrementales: primero se entregan las funciones esenciales, y los fragmentos adicionales se entregan dependiendo de la evaluación del primer fragmento. Aunque el sistema vaya a entregarse como un solo paquete completo, el establecimiento de la prioridad de las funciones permite que el cliente comunique con claridad lo que es importante para él y dónde deberá ponerse el énfasis del desarrollo. La figura 5-22 proporciona un ejemplo de un esquema de prioridades.

A cada función se le debe asignar alguna de las siguientes prioridades

- **Prioridad alta**—Una característica de prioridad alta debe demostrarse en forma satisfactoria durante la aceptación del cliente.
- **Prioridad media**—Una característica de prioridad media debe tomarse en cuenta en el diseño del sistema y en el diseño de objetos. Se instrumentará y demostrará en la segunda iteración del desarrollo del sistema.
- **Prioridad baja**—Una característica de prioridad baja ilustra la manera en que podría extenderse el sistema a largo plazo.

Figura 5-22 Un ejemplo de un esquema de prioridad para los requerimientos.

Un proceso de revisión permite que el cliente y el desarrollador definan la manera en que se van a manejar los cambios a los requerimientos después de la aprobación. Los requerimientos cambiarán, ya sea a causa de errores, omisiones, cambios en el ambiente operativo, cambios en el dominio de aplicación o cambios en la tecnología. La definición desde el principio de un pro-

ceso de revisión fomenta que los cambios se comuniquen a lo largo del proyecto y reduce la cantidad de sorpresas a largo plazo. Observe que un proceso de cambio no necesita ser burocrático o requerir una sobrecarga excesiva. Puede ser tan simple como designar a una persona como responsable para la recepción de solicitudes de cambio, aprobar los cambios y llevar cuenta de su implementación. La figura 5-23 muestra un ejemplo más complejo en donde los cambios son diseñados y revisados por el cliente antes de que se implementen en el sistema. En todos los casos, el reconocimiento de que no se pueden congelar los requerimientos (sino sólo considerarlos como una línea base) beneficiará al proyecto.

La lista de criterios de aceptación se revisa antes de la aprobación. La actividad de obtención de requerimientos y análisis aclara muchos aspectos del sistema, incluyendo los requerimientos no funcionales a los cuales debe apegarse el sistema y la importancia relativa de cada función. Al

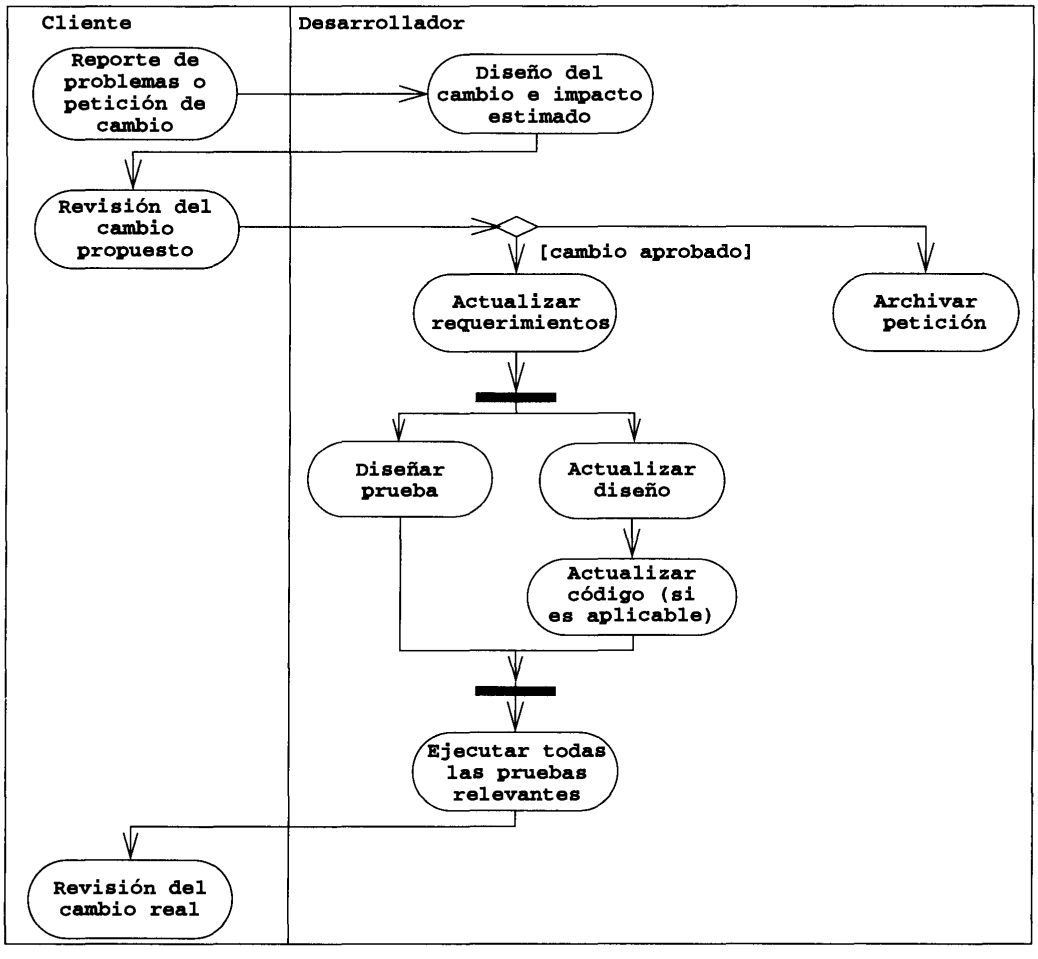


Figura 5-23 Un ejemplo de un proceso de revisión (diagrama de actividad UML).

volver a plantear los criterios de aceptación en la aprobación, el cliente asegura que los desarrolladores estén actualizados acerca de cualquier cambio en sus expectativas.

El presupuesto y la calendarización se revisan después de que el modelo de análisis se vuelve estable. En el capítulo 11, *Administración del proyecto*, describimos asuntos relacionados con la estimación de costos.

La aprobación del cliente es un hito importante en el proyecto, ya sea que se trate de un nuevo acuerdo contractual o bien que el proyecto esté regido por un contrato establecido con anterioridad. La aprobación del cliente representa la convergencia de éste y el desarrollador en un solo conjunto de definiciones funcionales del sistema y un solo conjunto de expectativas. La aceptación del documento de análisis de requerimientos es más crítica que la de cualquier otro documento, en vista de que muchas actividades dependen del modelo de análisis.

5.6 Ejercicios

1. Considere un sistema de archivo con una interfaz gráfica de usuario, como el Finder de Macintosh, el Explorador de Windows de Microsoft o el KDE de Linux. Los siguientes objetos se identificaron en un caso de uso que describe la manera de copiar un archivo desde un disco flexible hacia un disco duro: `Archivo`, `Icono`, `Papelera de reciclaje`, `Carpeta`, `Disco`, `Apuntador`. Especifique cuáles son objetos de entidad, cuáles son de frontera y cuáles son de control.
2. Suponiendo el mismo sistema de archivo anterior, considere un escenario que consista en seleccionar un archivo en un disco flexible, arrastrarlo hacia una carpeta y soltarlo. Identifique y defina al menos un objeto de control asociado con este escenario.
3. Acomode en forma horizontal los objetos que se listan en los ejercicios 1 y 2 en un diagrama de secuencia, poniendo a la izquierda los objetos de frontera, luego los objetos de control y por último los objetos de entidad. Trace la secuencia de interacciones resultantes de soltar el archivo en la carpeta. Por ahora ignore los casos excepcionales.
4. Examine el diagrama de secuencia que produjo en el ejercicio 4 e identifique las asociaciones entre esos objetos.
5. Identifique los atributos de cada objeto que sean relevantes para este escenario (mover un archivo desde un disco flexible hacia un disco duro). También considere los casos de excepción "Ya hay un archivo con ese nombre en la carpeta" y "Ya no hay espacio en el disco".
6. Considere el modelo de objetos de la figura 5-24 (adaptado de [Jackson, 1995]): dado su conocimiento del calendario gregoriano, liste todos los problemas que tiene este modelo. Modifíquelo para corregirlos.
7. Considere el modelo de objetos de la figura 5-24. Usando solamente la multiplicidad de asociación, ¿puede modificar el modelo para que alguien que no esté familiarizado con el calendario gregoriano pueda deducir la cantidad de días de cada mes? Identifique clases adicionales si es necesario.
8. Considere un sistema de semáforos en un cruce de caminos de cuatro vías (por ejemplo, dos caminos que se cruzan en ángulo recto). Suponga el algoritmo más simple para hacer ciclo entre las luces (por ejemplo, se permite que pase todo el tráfico de una vía mientras se detiene al tráfico de la otra). Identifique los estados de este sistema y trace una gráfica de estado que los describa. Recuerde que cada semáforo tiene tres estados (es decir, verde, ámbar y rojo).

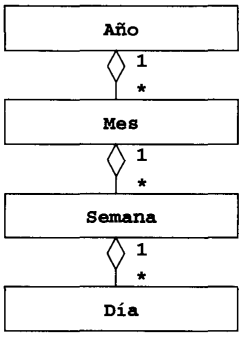


Figura 5-24 Un modelo sencillo del calendario gregoriano (diagrama de clase UML).

Referencias

[Abbott, 1983] R. Abbott, "Program design by informal English descriptions", *Communications of the ACM*, 1983, vol. 26, núm. 11.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* 2a. ed. Benjamin/ Cummings, Redwood City, CA, 1994.

[Booch *et al.*, 1998] G. Booch, J. Rumbaugh e I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, MA, 1998.

[Bruegge *et al.*, 1994] B. Bruegge, K. O'Toole y D. Rothenberger, "Design considerations for an accident management system", en M. Brodie, M. Jarke y M. Papazoglou (eds.), *Proceedings of the Second International Conference on Cooperative Information Systems.* University of Toronto, Canadá, mayo de 1994, págs. 90–100.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification.* Yourdon, Nueva York, 1978.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Reading, MA, 1997.

[FRIEND, 1994] *FRIEND Project Documentation.* School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1994.

[Harel, 1987] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 1987, págs. 231–274.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[Jacobson *et al.*, 1999] Jacobson, G. Booch y J. Rumbaugh, *The Unified Software Development Process.* Addison-Wesley, Reading, MA, 1999.

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, Reading, MA, 1995.

[Larman, 1998] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, Upper Saddle River, NJ, 1998.

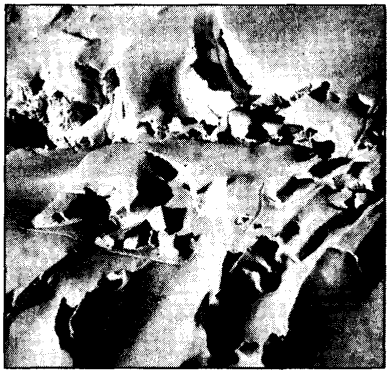
[Meyer, 1997] Bertrand Meyer, *Object-Oriented Software Construction*, 2a. ed. Prentice Hall, Upper Saddle River, NJ, 1997.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Wirfs-Brock *et al.*, 1990] R. Wirfs-Brock, B. Wilkerson y Lauren Wiener, *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ, 1990.

6

6



Diseño del sistema

Hay dos formas de construir un diseño de software: una es hacerlo tan simple que obviamente no haya deficiencias, y la otra es hacerlo tan complicado que no haya deficiencias obvias.

—C. A. R. Hoare

El diseño de sistemas es la transformación del modelo de análisis en un modelo de diseño del sistema. Durante el diseño del sistema los desarrolladores definen los objetivos de diseño del proyecto y descomponen el sistema en subsistemas más pequeños que pueden ser realizados por equipos individuales. Los desarrolladores también seleccionan estrategias para la construcción del sistema, como la plataforma de hardware y software en la que se ejecutará el sistema, la estrategia de almacenamiento de datos persistentes, el flujo de control global, la política de control de acceso y el manejo de condiciones de frontera. El resultado del diseño del sistema es un modelo que incluye una descripción clara de cada una de estas estrategias, una descomposición en subsistemas y un diagrama de organización UML que representa la correspondencia entre el hardware y el software del sistema.

El diseño de sistemas no es algorítmico. Sin embargo, los profesionales y académicos han desarrollado soluciones patrón para problemas comunes, y han definido notaciones para la representación de arquitecturas de software. En este capítulo presentamos primero esos bloques de construcción y luego tratamos las actividades de diseño que tienen impacto sobre esos bloques de construcción. En particular, el diseño de sistemas incluye:

- La definición de los objetivos de diseño
- La descomposición del sistema en subsistemas
- La selección de componentes hechos y heredados
- La correspondencia entre los subsistemas y el hardware
- La selección de la infraestructura de administración de datos persistentes
- La selección de una política de control de acceso
- La selección de un mecanismo de flujo de control global
- El manejo de condiciones de frontera

Concluimos este capítulo describiendo asuntos administrativos que se relacionan con el diseño de sistemas.

6.1 Introducción: un ejemplo del plano de un piso

El diseño del sistema, el diseño de objetos y la implementación constituyen la construcción del sistema. Durante estas tres actividades los desarrolladores llenan el hueco entre la especificación del sistema producida durante la obtención de requerimientos y el análisis y el sistema que se entregará a los usuarios. El diseño del sistema es el primer paso de este proceso y se enfoca en la descomposición del sistema en partes manejables. Durante la obtención de requerimientos y el análisis nos concentramos en el propósito y la funcionalidad del sistema. Durante el diseño del sistema nos enfocamos en los procesos, estructuras de datos y componentes de software y hardware necesarios para implementarlo. El reto del diseño del sistema es que hay que satisfacer muchos criterios y restricciones conflictivos cuando se descompone el sistema.

Considere, por ejemplo, la tarea de diseñar una casa residencial. Después de ponerse de acuerdo con el cliente sobre la cantidad de cuartos y pisos, el tamaño del área de estar y la ubicación de la casa, el arquitecto debe diseñar el plano de la casa, esto es, dónde deberán ir las paredes, puertas y ventanas. Debe hacerlo de acuerdo con varios requerimientos funcionales: la cocina debe estar cerca del comedor y la cochera, el baño debe estar cerca de las recámaras, etc. El arquitecto también debe apoyarse en varios estándares cuando establece las dimensiones de cada cuarto y la posición de la puerta: los anaqueles de cocina vienen en incrementos fijos y las camas tienen tamaños estándar. Sin embargo, observe que el arquitecto no tiene que conocer el contenido exacto de cada cuarto ni la disposición del mobiliario, sino que, por el contrario, estas decisiones deben posponerse y dejárselas al cliente.

La figura 6-1 muestra tres revisiones sucesivas de un plano de piso para una casa residencial. Necesitamos satisfacer las siguientes restricciones:

1. Esta casa debe tener dos recámaras, un estudio, una cocina y un área de estar.
2. Se debe minimizar la distancia general que recorren diario los ocupantes.
3. Se debe maximizar la utilización de la luz de día.

Para satisfacer las restricciones anteriores suponemos que la mayoría de los recorridos se harán entre la puerta de entrada y la cocina cuando los alimentos se descargan del coche, y entre la cocina y el área de estar cuando se transporta la vajilla antes y después de las comidas. El siguiente recorrido a minimizar es entre las recámaras y el baño. Suponemos que los ocupantes de la casa pasarán la mayor parte del tiempo en el área de estar y en la recámara principal.

En la primera versión del plano (en la parte superior de la figura 6-1) encontramos que el área de estar está demasiado lejos de la cocina. Para resolver este problema intercambiamos la recámara 2 (vea las flechas grises de la figura 6-1). Esto también tiene la ventaja de mover el área de estar hacia la pared sur de la casa. En la segunda revisión encontramos que la cocina y las escaleras están muy lejos de la puerta de entrada. Para resolver este problema movemos la puerta de entrada a la pared norte. Esto nos permite reacomodar la recámara 2 y mover el baño más cerca de las recámaras. El área de estar se incrementa y satisfacemos todas las restricciones que se impusieron al principio.

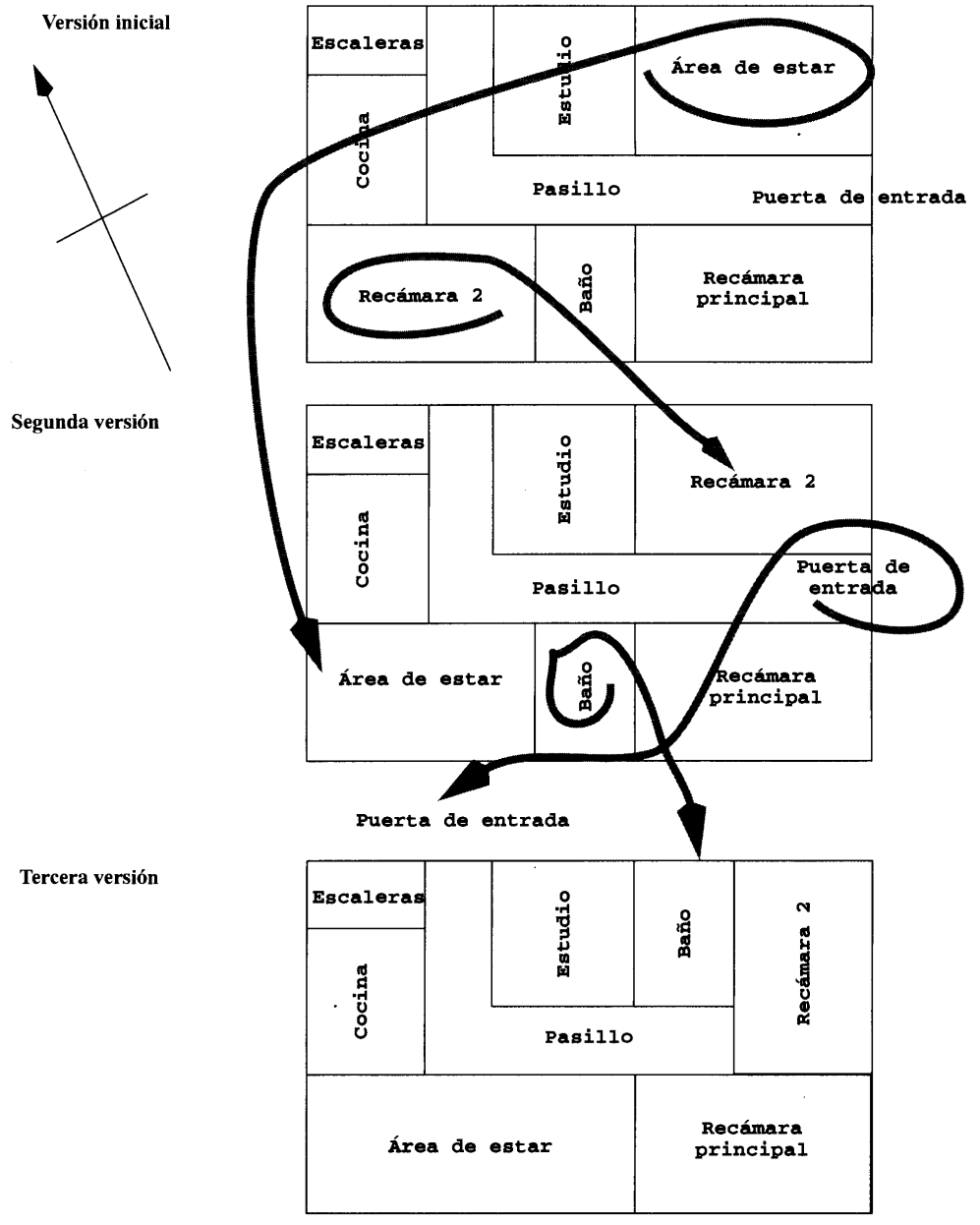


Figura 6-1 Ejemplo de un diseño de plano de un piso. Tres versiones sucesivas muestran cómo minimizamos la distancia a recorrer y aprovechamos la luz de día.

En este momento podemos colocar las puertas y ventanas de cada cuarto para precisar y detallar los requerimientos. Después que hayamos hecho esto habremos terminado el diseño del plano sin un conocimiento detallado de la disposición de cada cuarto individual. Se puede continuar con los planos para las instalaciones hidráulica, eléctrica y de calefacción.

El diseño de un plano en arquitectura es similar al diseño en la ingeniería de software. El conjunto se divide en componentes más simples (es decir, cuartos, subsistemas) e interfaces (es decir, puertas, servicios) tomando en cuenta los requerimientos no funcionales (es decir, área de estar, tiempo de respuesta) y funcionales (es decir, cantidad de recámaras, casos de uso). El diseño del sistema tiene un impacto en las actividades de implementación (es decir, la disposición de la cocina, la complejidad de la codificación de subsistemas individuales), y si se cambia después provoca una repetición costosa del trabajo (es decir, movimiento de paredes, cambio de interfaces de subsistemas). El diseño de los componentes individuales se deja para después.

En la sección 6.2 proporcionamos una vista a ojo de pájaro del diseño del sistema y su relación con el análisis. En la sección 6.3 describimos el concepto de subsistemas y la descomposición en subsistemas. En la sección 6.4 describimos las actividades del diseño del sistema y usamos un ejemplo para ilustrar la manera en que se pueden usar juntos estos bloques de construcción. En la sección 6.5 describimos asuntos administrativos relacionados con el diseño de sistemas.

6.2 Un panorama del diseño de sistemas

El análisis da como resultado el modelo de requerimientos descrito por los siguientes productos:

- Un conjunto de *requerimientos no funcionales* y *restricciones*, como tiempo de respuesta máximo, producción mínima, confiabilidad, plataforma de sistema operativo, etcétera.
- Un *modelo de caso de uso* que describe la funcionalidad del sistema desde el punto de vista de los actores.
- Un *modelo de objetos* que describe las entidades manipuladas por el sistema.
- Un *diagrama de secuencia* para cada caso de uso que muestra la secuencia de interacciones entre los objetos que participan en el caso de uso.

El modelo de análisis describe el sistema por completo desde el punto de vista de los actores y sirve como la base de comunicación entre el cliente y los desarrolladores. Sin embargo, el modelo de análisis no contiene información acerca de la estructura interna del sistema, su configuración de hardware o, en términos más generales, la manera en que se debe realizar el sistema. El diseño del sistema es el primer paso en esta dirección. El diseño del sistema da como resultado los siguientes productos:

- Una lista de *objetivos de diseño* que describe las cualidades del sistema que deben optimizar los desarrolladores.
- Una *arquitectura de software* que describe la descomposición en subsistemas desde el punto de vista de responsabilidades del subsistema, dependencias entre subsistemas, correspondencia de los subsistemas con el hardware y decisiones de política principales, como el flujo de control, control de acceso y almacenamiento de datos.

Los objetivos de diseño se derivan de los requerimientos no funcionales. Los objetivos de diseño guían las decisiones que deben tomar los desarrolladores, en especial cuando hay compromisos.

La descomposición en subsistemas constituye la mayor parte del diseño del sistema. Los desarrolladores dividen el sistema en partes manejables para tratar la complejidad: cada subsistema se asigna a un equipo y se realiza en forma independiente. Sin embargo, para que esto sea posible, los desarrolladores necesitan atacar asuntos en el nivel de sistema cuando descomponen el sistema. En particular necesitan atacar los siguientes asuntos:

- *Correspondencia entre hardware y software:* ¿Cuál es la configuración de hardware del sistema? ¿Cuál nodo es responsable de cuál funcionalidad? ¿Cómo se realiza la comunicación entre nodos? ¿Cuáles servicios se realizan usando componentes de software existentes? ¿Cómo se encapsulan estos componentes? El establecimiento de la correspondencia entre hardware y software conduce, con frecuencia, a la definición de *subsistemas adicionales* dedicados a mover datos de un nodo hacia otro, manejando los asuntos de concurrencia y confiabilidad. Los componentes ya hechos y adquiridos a terceros ("tomados del mostrador") permiten que los desarrolladores realicen servicios complejos en forma más económica. Los paquetes de interfaz de usuario y los sistemas de administración de bases de datos son muy buenos ejemplos de componentes ya hechos. Sin embargo, los componentes deben encapsularse para minimizar la dependencia de un componente particular: puede llegar un vendedor con un mejor componente.
- *Administración de datos:* ¿Cuáles datos necesitan ser persistentes? ¿Dónde deben guardarse los datos persistentes? ¿Cómo se tiene acceso a ellos? Los datos persistentes representan un cuello de botella en muchos frentes diferentes del sistema: la mayor parte de la funcionalidad del sistema se relaciona con la creación o manipulación de datos persistentes. Por esta razón, el acceso a los datos debe ser rápido y confiable. Si la recuperación de los datos es lenta, el sistema completo será lento. Si es probable la corrupción de los datos, también es probable la falla completa del sistema. Estos asuntos necesitan manejarse en forma consistente en el nivel de sistema. Con frecuencia, esto conduce a la selección de un sistema de administración de base de datos y a un *subsistema adicional* dedicado a la administración de datos persistentes.
- *Control de acceso:* ¿Quién puede tener acceso a cuáles datos? ¿Puede cambiar de manera dinámica el control de acceso? ¿Cómo se especifica y realiza el control de acceso? El control de acceso y la seguridad son asuntos en el nivel de sistema. El control de acceso debe ser consistente por todo el sistema. En otras palabras, la política usada para especificar quién puede y quién no puede tener acceso a determinados datos debe ser la misma en *todos los subsistemas*.
- *Flujo de control:* ¿Cómo pone el sistema en secuencia a las operaciones? ¿El sistema es manejado por eventos? ¿Puede manejar más de una interacción de usuario a la vez? La selección del control de flujo tiene impacto en las interfaces de los subsistemas. Si se selecciona un control de flujo manejado por eventos, los subsistemas proporcionarán manejadores de eventos. Si se seleccionan hilos de proceso, los subsistemas deben garantizar la exclusión mutua en secciones críticas.
- *Condiciones de frontera:* ¿Cómo se inicia el sistema? ¿Cómo se le apaga? ¿Cómo se detectan y manejan los casos excepcionales? La iniciación y apagado del sistema representan, a menudo, la parte más grande de la complejidad del sistema, en especial en un ambiente distribuido. La iniciación, el apagado y el manejo de excepciones tienen impacto en la interfaz de *todos los subsistemas*.

La figura 6-2 muestra las actividades del diseño de sistemas. Cada actividad aborda cada uno de los asuntos que describimos antes. El ataque a cualquiera de esos asuntos puede dar lugar a cambios en la descomposición de los subsistemas y presentar nuevos asuntos. Como verá cuando describamos cada una de estas actividades, el diseño de sistemas es una actividad muy iterativa, que conduce en forma continua a la identificación de nuevos subsistemas, a la modificación de los subsistemas existentes y a revisiones en el nivel del sistema que tienen un impacto en todos los subsistemas. Pero primero describamos el concepto de subsistema con mayor detalle.

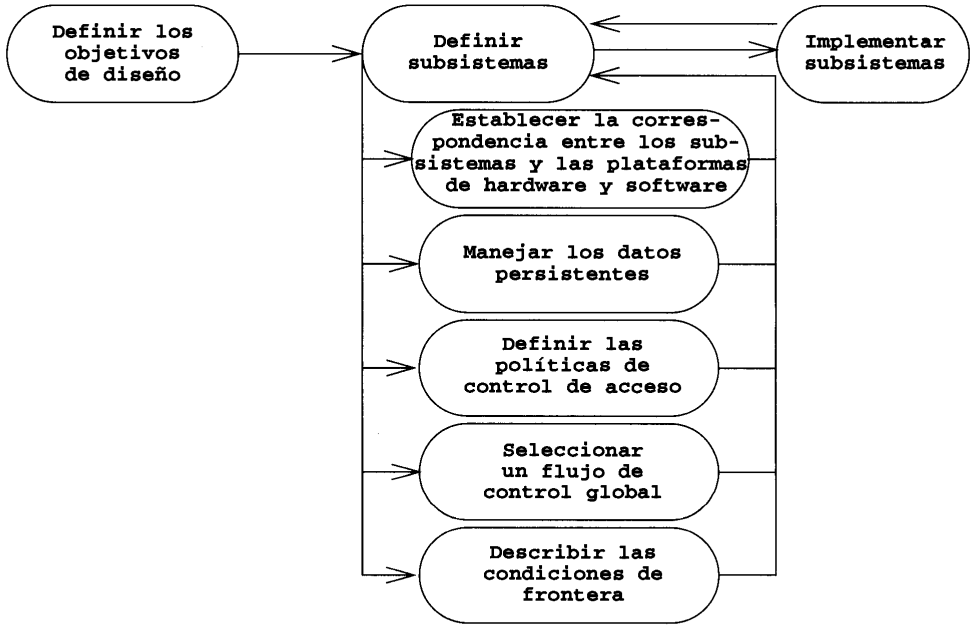


Figura 6-2 Las actividades del diseño de sistemas (diagrama de actividad UML).

6.3 Conceptos del diseño de sistemas

En esta sección describimos con mayor detalle la descomposición en subsistemas y sus propiedades. Primero definimos el concepto de **subsistema** y su relación con las clases (sección 6.3.1). Luego vemos la interfaz de los subsistemas (sección 6.3.2): los subsistemas proporcionan **servicios** a otros subsistemas. Un servicio es un conjunto de operaciones relacionadas que comparten un propósito común. Durante el diseño del sistema definimos los subsistemas desde el punto de vista de los servicios que proporcionan. Más adelante, durante el diseño de objetos, definimos la interfaz de los subsistemas desde el punto de vista de las operaciones que proporcionan. Luego vemos dos propiedades de los subsistemas: **acoplamiento** y **coherencia** (sección 6.3.3). El acoplamiento mide la dependencia entre dos subsistemas y la coherencia mide las dependencias entre clases dentro de un subsistema. La descomposición ideal en subsistemas debe minimizar el acoplamiento y maximizar la coherencia. Luego vemos la **subdivisión en capas** y la **partición**, dos técnicas para

relacionar subsistemas (sección 6.3.4). La subdivisión en capas permite que se organice un sistema como una jerarquía de subsistemas, proporcionando cada uno servicios de alto nivel a los subsistemas que tiene encima usando servicios de nivel más bajo de los subsistemas que están debajo de él. La partición organiza los subsistemas como partes iguales que se proporcionan mutuamente diferentes servicios. En la sección 6.3.5 describimos varias arquitecturas de software típicas que se encuentran en la práctica. Por último, en la sección 6.3.6 describimos los diagramas de organización UML que usamos para representar la correspondencia entre los subsistemas de software y los componentes de hardware.

6.3.1 Subsistemas y clases

En el capítulo 2, *Modelado con UML*, presentamos la distinción entre dominio de aplicación y dominio de solución. Para reducir la complejidad del dominio de aplicación identificamos partes más pequeñas, llamadas clases, y las organizamos en paquetes. En forma similar, para reducir la complejidad del dominio de solución descomponemos un sistema en partes más simples, llamadas subsistemas, que están compuestas de varias clases del dominio de solución. En el caso de subsistemas complejos aplicamos en forma reiterada este principio y descomponemos un subsistema en subsistemas más simples (vea la figura 6-3).

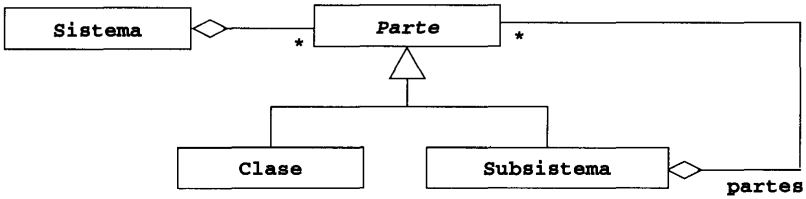


Figura 6-3 Descomposición de subsistemas (diagrama de clase UML).

Por ejemplo, el sistema de administración de accidentes que describimos antes puede descomponerse en un subsistema `InterfazDespachador` que implementa la interfaz de usuario para el `Despachador`, un subsistema `InterfazOficialCampo` que implementa la interfaz de usuario para el `OficialCampo`, un subsistema `AdministrarIncidente` que implementa la creación, modificación y almacenamiento de `Incidente` y un subsistema `Notificación` que implementa la comunicación entre las terminales `OficialCampo` y las estaciones `Despachador`. Esta descomposición en subsistemas se muestra en la figura 6-4 usando paquetes UML.

Varios lenguajes de programación (por ejemplo, Java y Modula-2) proporcionan construcciones para el modelado de subsistemas (paquetes en Java, módulos en Modula-2). En otros lenguajes, como C o C++, los subsistemas no están modelados en forma explícita y, en ese caso, los desarrolladores usan convenciones para el agrupamiento de clases (por ejemplo, un subsistema puede representarse como un directorio que contiene todos los archivos que implementan el subsistema). Sin importar si los subsistemas están representados en forma explícita en el lenguaje de programación o no, los desarrolladores necesitan documentar de manera minuciosa la descomposición en subsistemas, ya que los subsistemas son realizados, por lo general, por equipos diferentes.

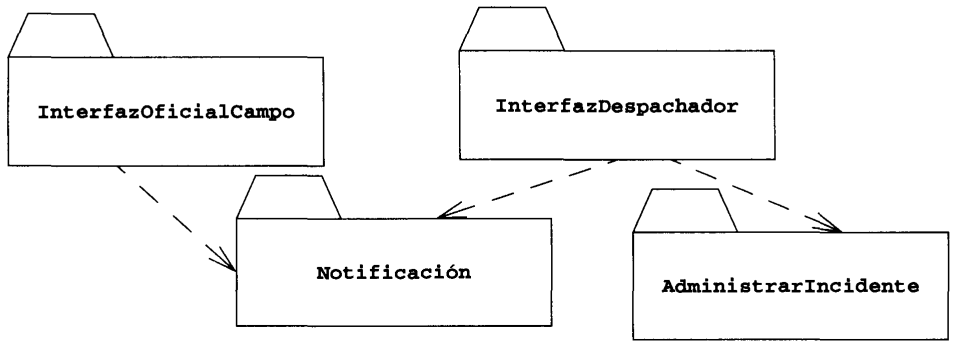


Figura 6-4 Descomposición en subsistemas para un sistema de administración de accidentes (diagrama de clase UML, vista colapsada). Los subsistemas se muestran como paquetes UML. Las flechas de guiones indican dependencias entre subsistemas.

6.3.2 Servicios e interfaces de subsistema

Un subsistema se caracteriza por los **servicios** que proporciona a otros subsistemas. Un servicio es un conjunto de operaciones relacionadas que comparten un propósito común. Un subsistema que proporciona un servicio de notificación, por ejemplo, define operaciones para enviar noticias, buscar canales de notificación y suscribirse y anular la suscripción a un canal.

El conjunto de operaciones de un subsistema que está disponible para otros subsistemas forma la **interfaz del subsistema**. Ésta, a la que también se le menciona como interfaz de programación de aplicaciones (API, por sus siglas en inglés), incluye el nombre de las operaciones, sus parámetros, sus tipos y sus valores de retorno. El diseño de sistemas se enfoca en la definición de los servicios proporcionados por cada subsistema, esto es, en la enumeración de sus operaciones, sus parámetros y su comportamiento de alto nivel. El diseño de objetos se enfocará en la definición de las interfaces de los subsistemas, es decir, el tipo de los parámetros y el valor de retorno de cada operación.

La definición de un subsistema desde el punto de vista de los servicios que proporciona nos ayuda a enfocarnos en su interfaz, en vez de en su implementación. Una buena interfaz de subsistema debe proporcionar la menor cantidad de información acerca de su implementación. Esto nos permite minimizar el impacto de los cambios cuando revisamos la implementación de un subsistema. En términos generales, queremos minimizar el impacto del cambio minimizando las dependencias entre subsistemas.

6.3.3 Acoplamiento y coherencia

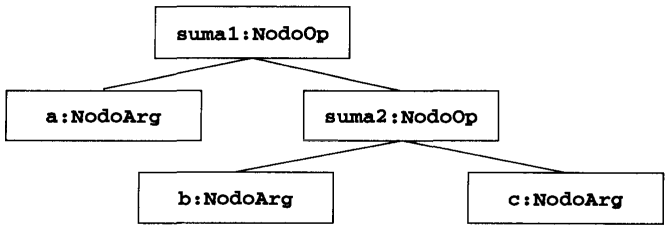
El **acoplamiento** es la fuerza de las dependencias entre dos subsistemas. Si dos subsistemas están poco acoplados son relativamente independientes y, por lo tanto, las modificaciones a uno de los subsistemas tendrá poco impacto en el otro. Si dos subsistemas están muy acoplados es probable que las modificaciones a un subsistema tengan impacto en el otro. Una propiedad deseable de una descomposición en subsistemas es que los subsistemas estén lo menos acoplados posible. Esto minimiza el impacto que tengan los errores o cambios futuros en la operación correcta del sistema.

Considere, por ejemplo, un compilador en el que un árbol de análisis es producido por el subsistema de análisis sintáctico y pasado al subsistema de análisis semántico. Ambos subsistemas acceden al árbol de análisis y lo modifican. Una forma eficiente para compartir grandes cantidades de datos es permitir que ambos subsistemas tengan acceso a los nodos del árbol de análisis por medio de atributos. Sin embargo, esto introduce un acoplamiento fuerte: ambos subsistemas necesitan conocer la estructura exacta del árbol de análisis y sus invariantes. Las figuras 6-5 y 6-6 muestran el efecto de cambiar la estructura de datos del árbol de análisis para dos casos: las columnas de la izquierda muestran las interfaces de clases cuando se usan atributos para compartir datos, y las columnas de la derecha muestran las interfaces de clases cuando se usan operaciones. Debido a que el analizador sintáctico y el analizador semántico dependen de esas clases, ambos subsistemas deberían ser modificados y volver a probarse en el caso que muestra la columna izquierda. En general, compartir datos por medio de atributos incrementa el acoplamiento y debe evitarse.

La **coherencia** es la fuerza de las dependencias dentro de un subsistema. Si un subsistema contiene muchos objetos que están relacionados entre sí y realizan tareas similares, su coherencia es alta. Si un subsistema contiene varios objetos que no están relacionados, su coherencia es baja. Una propiedad deseable de una descomposición en subsistemas es que conduzca a subsistemas con coherencia alta.

Por ejemplo, considere un sistema de seguimiento de decisiones para el registro de problemas de diseño, discusiones, evaluaciones alternas, decisiones y su implementación desde el punto de vista de tareas (vea la figura 6-7).

Representación de árbol binario



Compartiendo mediante atributos

```
class NodoOp {
    NodoArg izquierdo;
    NodoArg derecho;
    String nombre;
}
class NodoArg {
    String nombre;
}
```

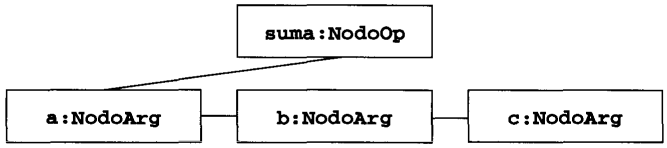
Compartiendo mediante operaciones

```
class NodoOp {
    Enumeration obtenerArgumentos();
    String obtenerNombre();
}

class NodoArg {
    String obtenerNombre();
}
```

Figura 6-5 Ejemplo de una reducción de acoplamiento (diagrama de objetos UML y declaraciones Java). Esta figura muestra un árbol de análisis para la expresión "a + b + c". La columna izquierda muestra la interfaz de la clase `NodoOp` compartiéndola mediante atributos. La columna derecha muestra la interfaz de `NodoOp` compartiéndola mediante operaciones. La figura 6-6 muestra los cambios para cada caso cuando, en vez de esto, se selecciona una lista vinculada.

Representación de lista vinculada



Compartiendo mediante atributos

```
class NodoOp {
    NodoArg primero;
    NodoArg izquierdo;
    NodoArg derecho;
    String nombre;
}
class NodoArg {
    String nombre;
    NodoArg siguiente;
}
```

Compartiendo mediante operaciones

```
class OpNode {
    Enumeration obtenerArgumentos();
    String obtenerNombre();
}

class NodoArg {
    String obtenerNombre();
}
```

Figura 6-6 Ejemplo de una reducción de acoplamiento (diagrama de objetos UML y declaraciones Java). Esta figura muestra el impacto de cambiar la representación de árbol de análisis de la figura 6-5 hacia una lista vinculada. En la columna izquierda, donde se comparte mediante atributos, es necesario cambiar cuatro atributos (los cambios se indican en cursivas). En la columna derecha, donde se comparte mediante operaciones, la interfaz permanece sin cambios.

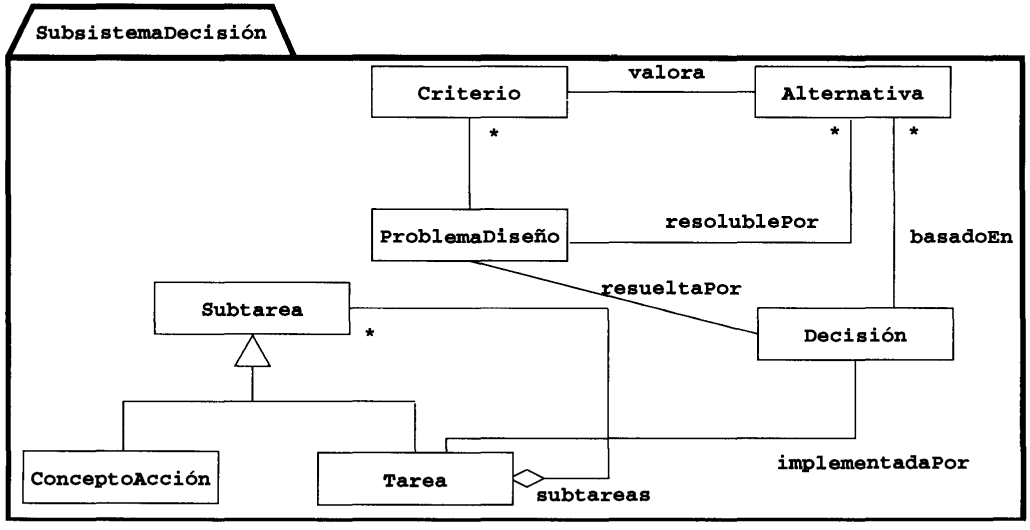


Figura 6-7 Sistema de seguimiento de decisiones (diagrama de clase UML). El `SubsistemaDecisión` tiene una coherencia baja: las clases `Criterio`, `Alternativa` y `ProblemaDiseño` no tienen relación con `Subtarea`, `ConceptoAcción` y `Tarea`.

ProblemaDiseño y Alternativa representan la exploración del espacio de diseño: formulamos el sistema desde el punto de vista de varios ProblemaDiseño y documentamos cada Alternativa que se explora. La clase Criterio representa las cualidades en las que estamos interesados. Una vez que valoramos las Alternativa exploradas contra los Criterio deseados, tomamos Decisión y las implementamos desde el punto de vista de Tarea. Las Tarea se descomponen en forma reiterada en Subtarea lo bastante pequeñas para que se asignen a desarrolladores individuales. A las tareas atómicas las llamamos ConceptoAcción.

El sistema de seguimiento de decisiones es tan pequeño que podríamos agrupar todas estas clases en un solo subsistema llamado SubsistemaDecisión (vea la figura 6-7). Sin embargo, observamos que el modelo de la clase puede partirse en dos subgráficas. Una, llamada SubsistemaRacional, contiene las clases ProblemaDiseño, Alternativa, Criterio y Decisión. La otra, llamada SubsistemaPlaneación, contiene Tarea, Subtarea y ConceptoAcción (vea la figura 6-8). Ambos subsistemas tienen mejor coherencia que el SubsistemaDiseño original. Además, los subsistemas resultantes son más pequeños que el subsistema original:

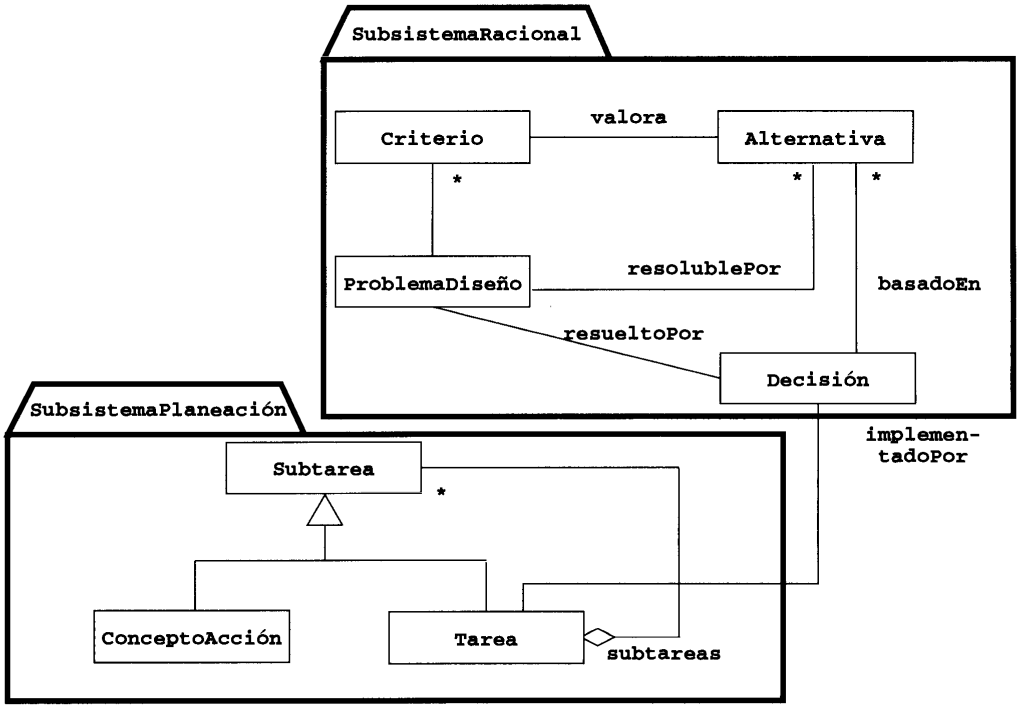


Figura 6-8 Descomposición alterna de subsistemas para el sistema de seguimiento de decisiones de la figura 6-7 (diagrama de clase UML). La coherencia del SubsistemaRacional y del SubsistemaPlaneación es más alta que la coherencia del SubsistemaDecisión original. Observe también que hemos reducido la complejidad descomponiendo el sistema en subsistemas más pequeños.

redujimos complejidad. El acoplamiento entre los nuevos subsistemas es relativamente bajo, ya que sólo hay una asociación entre los dos subsistemas.

En general, hay un compromiso entre la coherencia y el acoplamiento. Siempre podemos incrementar la coherencia descomponiendo el sistema en subsistemas más pequeños. Sin embargo, esto incrementa el acoplamiento conforme se incrementa la cantidad de interfaces. Una buena heurística es que los desarrolladores pueden manejar 7 ± 2 conceptos en cualquier nivel de abstracción. Si hay más de nueve subsistemas en cualquier nivel de abstracción dado, o si hay un subsistema que proporciona más de nueve servicios, se debe considerar una revisión de la descomposición. Siguiendo la misma regla, la cantidad de capas no debe ser mayor a 7 ± 2. De hecho, muchos buenos diseños de sistemas pueden realizarse con sólo tres capas.

6.3.4 Capas y particiones

El objetivo del diseño del sistema es manejar la complejidad dividiendo el sistema en partes más manejables. Esto puede lograrse mediante un enfoque de dividir y conquistar, en donde dividimos las partes en forma reiterada hasta que son lo bastante simples para ser manejadas por una persona o un equipo. La aplicación sistemática de este enfoque conduce a una descomposición jerárquica en la cual cada subsistema, o **capa**, proporciona servicios de nivel más alto usando servicios proporcionados por subsistemas de nivel inferior (vea la figura 6-9). Cada capa puede depender sólo de las capas de nivel inferior y no tiene conocimiento de las capas que están encima de ella. En una **arquitectura cerrada** cada capa sólo puede depender de las capas que están inmediatamente debajo de ella. En una **arquitectura abierta** una capa también puede tener acceso a capas que están en niveles más profundos.

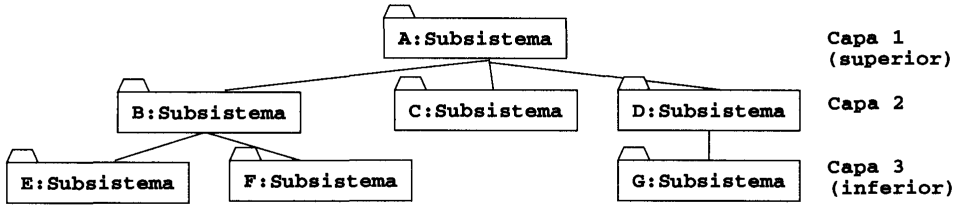


Figura 6-9 Descomposición de un sistema en tres capas de subsistemas (diagrama de objetos UML). A un subconjunto de una descomposición en capas que incluye al menos un subsistema de cada capa se le llama rebanada vertical. Por ejemplo, los subsistemas A, B y E constituyen una rebanada vertical, mientras que no es así con los subsistemas D y G.

Un ejemplo de arquitectura cerrada es el Modelo de Referencia de la Interconexión de Sistemas Abiertos (abreviado, el modelo OSI, por sus siglas en inglés), que está compuesto por siete capas [Day y Zimmermann, 1983]. Cada capa es responsable de la realización de una función bien definida. Además, cada capa proporciona sus servicios usando servicios de la capa inferior (vea la figura 6-10).

La capa `Física` representa la interfaz de hardware hacia la red. Es responsable de la transmisión de bits por los canales de comunicación. La capa `VínculoDeDatos` es responsable de la transmisión de marcos de datos sin error usando los servicios de la capa `Física`. La capa `Red` es

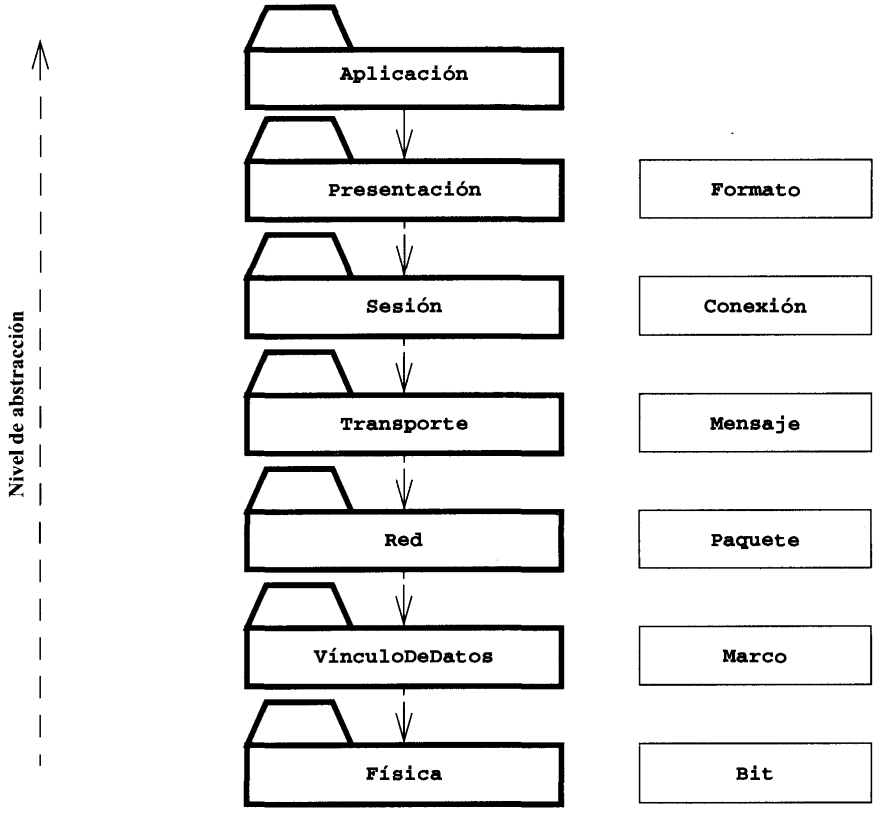


Figura 6-10 Un ejemplo de arquitectura cerrada: el modelo OSI (diagrama de clase UML). El modelo OSI descompone los servicios de red en siete capas, donde cada una es responsable de un nivel de abstracción diferente.

responsable de la transmisión y el enrutamiento de paquetes dentro de una red. La capa `Transporte` es responsable de asegurarse que los datos se transmitan en forma confiable de extremo a extremo. La capa `Transporte` es la interfaz que ven los programadores Unix cuando transmiten información a través de sockets TCP/IP entre dos procesos. La capa `Sesión` es responsable de la iniciación de una conexión, incluyendo la autentificación. La capa `Presentación` realiza los servicios de transformación de datos, como el intercambio de bytes o el cifrado. La capa `Aplicación` es el sistema que uno está diseñando (a menos que esté construyendo un sistema operativo o pila de protocolo). La capa `Aplicación` también puede consistir de subsistemas en capas.

Hasta hace poco sólo estaban estandarizadas las cuatro capas inferiores del modelo OSI. Unix y muchos sistemas operativos de escritorio, por ejemplo, proporcionan interfaces para TCP/IP que implementan las capas `Transporte`, `Red` y `VínculoDeDatos`. El desarrollador de aplicaciones todavía necesita llenar el hueco entre las capas `Transporte` y `Aplicación`. Con el creciente número de aplicaciones distribuidas este hueco motivó el desarrollo de middleware como CORBA [OMG,

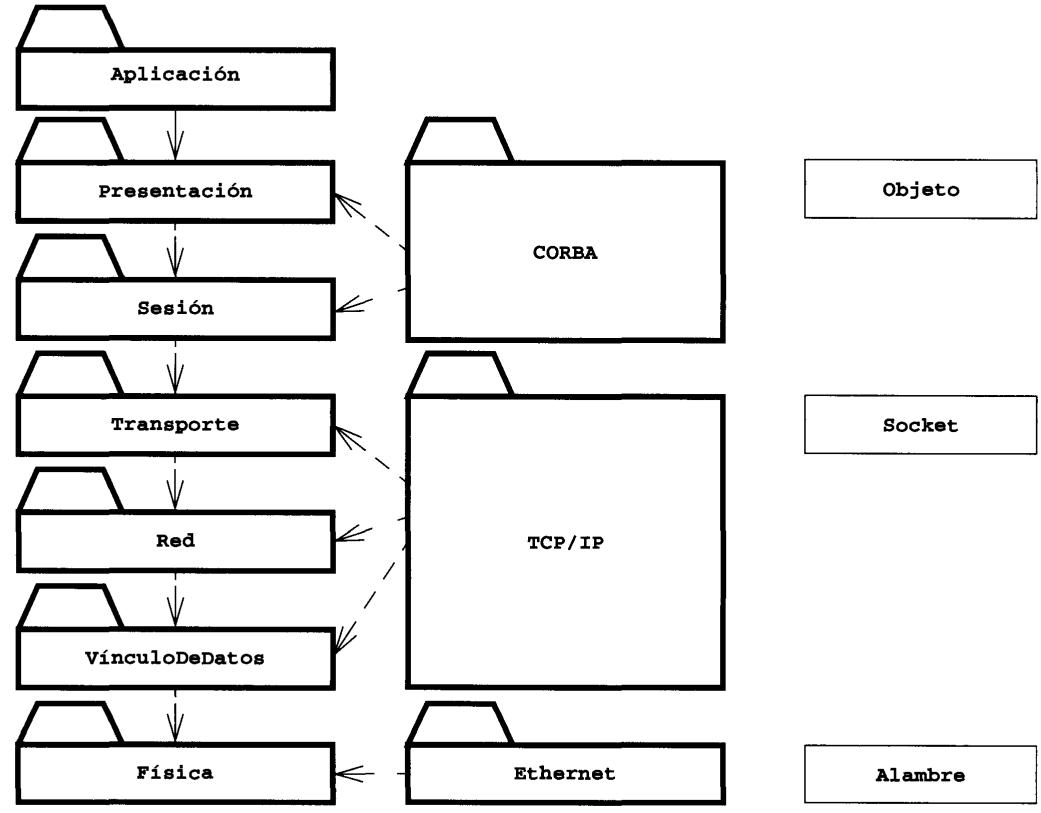


Figura 6-11 Un ejemplo de arquitectura cerrada (diagrama de clase UML). CORBA permite el acceso a objetos implementados en diferentes lenguajes en anfitriones diferentes. CORBA implementa en forma efectiva las capas `Presentación` y `Sesión` de la pila OSI.

1995] y Java RMI [RMI, 1998]. CORBA y Java RMI nos permiten tener acceso a objetos remotos en forma transparente, enviándoles mensajes en forma similar a como se envían a los objetos locales, implementando en forma efectiva las capas de `Presentación` y `Sesión` (vea la figura 6-11).

Un ejemplo de arquitectura abierta es el juego de herramientas de interfaz de usuario Motif para X11 [Nye y O'Reilly, 1992]. La capa inferior, `Xlib`, proporciona facilidades de trazado básicas y define el concepto de ventana. `Xt` proporciona herramientas para la manipulación de objetos de interfaz de usuario, llamados aparatos mecánicos, usando servicios de `Xlib`. `Motif` es una biblioteca de aparatos mecánicos que proporciona un amplio rango de facilidades, desde botones hasta manejo de la geometría. `Motif` está construido sobre `Xt`, pero también tiene acceso directo a `Xlib`. Por último, una aplicación que usa `Motif`, como un administrador de ventanas, puede tener acceso a las tres capas. `Motif` no tiene conocimiento del administrador de ventanas y `Xt` no tiene conocimiento de `Motif` o de la aplicación. Muchos otros juegos de herramientas de interfaz de usuario para X11 tienen arquitecturas abiertas. La apertura de la arquitectura permite que los desarrolladores hagan a un lado las capas superiores cuando se presentan cuellos de botella (figura 6-12).

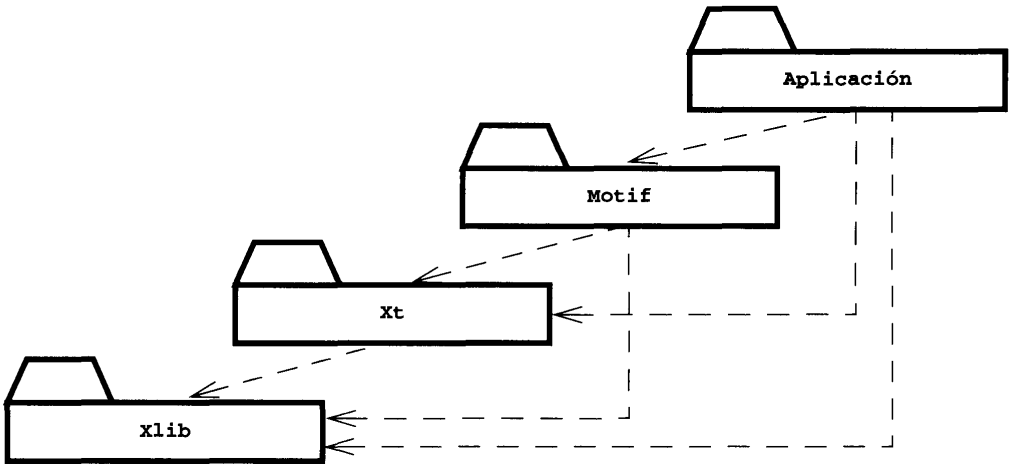


Figura 6-12 Un ejemplo de arquitectura abierta: la biblioteca OSF/Motif (diagrama de clase UML, paquetes colapsados). `Xlib` proporciona facilidades de trazado de bajo nivel. `Xt` proporciona el manejo de aparatos mecánicos de interfaz de usuario básicos. `Motif` proporciona una gran cantidad de aparatos mecánicos sofisticados. La `Aplicación` puede tener acceso a cada una de estas capas en forma independiente.

Las arquitecturas de capas cerradas tienen propiedades deseables: conducen a un bajo acoplamiento entre subsistemas y los subsistemas pueden integrarse y probarse en forma incremental. Sin embargo, cada nivel introduce una sobrecarga de velocidad y almacenamiento que puede dificultar la satisfacción de requerimientos no funcionales. Además, puede ser difícil la adición de funcionalidad al sistema en revisiones posteriores, en especial cuando no se anticiparon las adiciones. En la práctica, rara vez se descompone un sistema en más de tres a cinco capas.

Otro enfoque para el manejo de la complejidad es hacer una **partición** del sistema en subsistemas de igual rango, donde cada uno es responsable de una clase de servicios diferente. Por ejemplo, un sistema a bordo para un automóvil puede descomponerse en un servicio de viaje que dé direcciones en tiempo real al conductor, un servicio de preferencias individuales que recuerde la posición del asiento del conductor y las estaciones de radio favoritas y un servicio del vehículo que lleve cuenta del consumo de gasolina del automóvil, las reparaciones y el mantenimiento preventivo. Cada subsistema depende vagamente de los demás y con frecuencia puede operar aislado.

En general, la descomposición en subsistemas es el resultado de la partición y de la separación en capas. Primero partimos el sistema en subsistemas de alto nivel que son responsables de la funcionalidad específica o que se ejecutan en un nodo de hardware específico. Si la complejidad lo justifica, cada uno de los subsistemas resultantes se descompone en capas de nivel más inferior hasta que es lo suficientemente simple para que lo implemente un solo desarrollador. Cada subsistema añade una determinada sobrecarga de procesamiento debido a su interfaz con los demás sistemas. La partición o las capas excesivas pueden conducir a una complejidad mayor.

6.3.5 Arquitectura de software

Conforme se incrementa la complejidad de los sistemas se hace crítica la especificación de la descomposición del sistema. Es difícil modificar o corregir una descomposición débil una vez que ha comenzado el desarrollo conforme tienen que cambiarse más interfaces de subsistema. En reconocimiento a la importancia de este problema ha surgido el concepto de **arquitectura de software**. Una arquitectura de software incluye la descomposición del sistema, el flujo de control global, las políticas de manejo de errores y los protocolos de comunicación entre subsistemas [Shaw y Garlan, 1996].

En esta sección describimos unas cuantas arquitecturas de ejemplo que pueden usarse para diferentes sistemas. Ésta no es, por ningún medio, una exposición sistemática o exhaustiva del tema. En vez de ello, pretendemos proporcionar unos cuantos ejemplos representativos y referir al lector a la literatura para mayores detalles.

Arquitectura de depósito

En la arquitectura de depósito (vea la figura 6-13) los subsistemas acceden y modifican datos de una sola estructura de datos llamada **depósito** central. Los subsistemas son relativamente independientes e interactúan tan sólo por medio de la estructura de datos central. El flujo de control puede estar dictado por el depósito central (por ejemplo, activadores en los datos llaman a los sistemas periféricos) o por los subsistemas (por ejemplo, flujo de control independiente y sincronización mediante candados en el depósito).

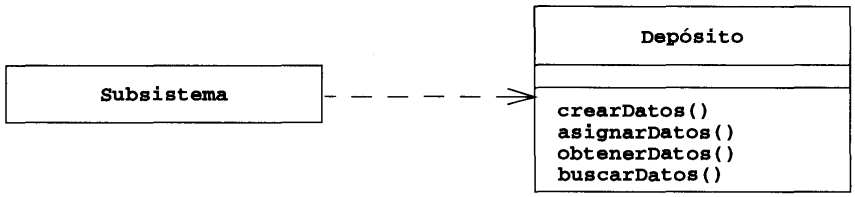


Figura 6-13 Arquitectura de depósito (diagrama de clase UML). Cada uno de los subsistemas depende sólo de una estructura de datos central llamada depósito. El depósito, a su vez, no tiene conocimiento de los demás subsistemas.

La arquitectura de depósito es típica de los sistemas de administración de bases de datos, como los sistemas de nómina o bancarios. La ubicación central de los datos facilita el manejo de los asuntos de concurrencia e integridad entre subsistemas. Los compiladores modernos y los ambientes de desarrollo de software también siguen una arquitectura de depósito (vea la figura 6-14). Los diferentes subsistemas de un compilador tienen acceso a un árbol de análisis y una tabla de símbolos centrales y los actualizan. Los depuradores y editores de sintaxis también tienen acceso a la tabla de símbolos.

El subsistema de depósito también puede usarse para la implementación del flujo de control global. En el ejemplo del compilador de la figura 6-14, cada herramienta individual (por ejemplo, el compilador, el depurador y el editor) es llamada por el usuario. El depósito sólo asegura que sean seriados los accesos concurrentes. En forma alterna, el depósito puede usarse para llamar a los subsistemas con base en el estado de la estructura de datos central. A estos sistemas se les llama sistemas de pizarrón. El sistema de comprensión del habla HEARSAY II [Erman *et al.*, 1980], uno de los primeros sistemas de pizarrón, selecciona las herramientas a llamar con base en el estado actual del pizarrón.

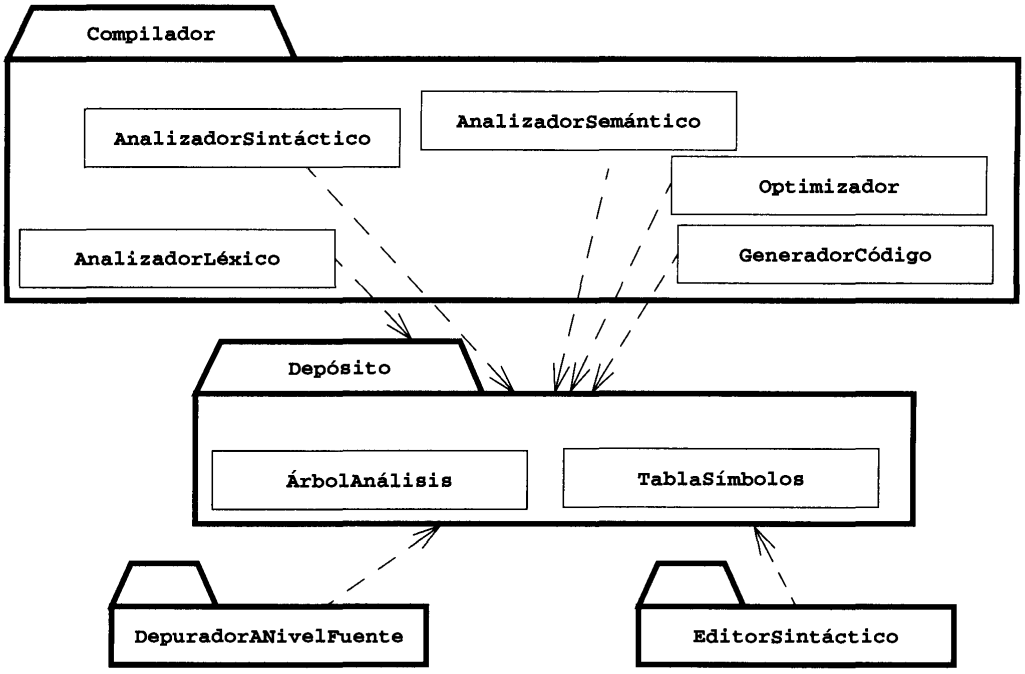


Figura 6-14 Una instancia de la arquitectura de depósito (diagrama de clase UML). Un compilador moderno genera en forma incremental un árbol de análisis y una tabla de símbolos que pueden ser usados después por los depuradores y los editores de sintaxis.

Las arquitecturas de depósito son bastante adecuadas para aplicaciones con tareas de procesamiento de datos complejos que cambian en forma constante. Una vez que está bien definido el depósito central podemos añadir con facilidad nuevos servicios en forma de subsistemas adicionales. La desventaja principal de los sistemas de depósito es que el depósito central puede convertirse rápido en un cuello de botella, tanto en el aspecto de desempeño como en el de modificabilidad. El acoplamiento entre cada subsistema y el depósito es alto y, por lo tanto, dificulta cambiar el depósito sin que tenga un impacto sobre todos los subsistemas.

Modelo/Vista/Controlador

En la arquitectura Modelo/Vista/Controlador (MVC) (vea la figura 6-15) se clasifica a los sistemas en tres tipos diferentes: los **subsistemas modelo** son responsables del mantenimiento del conocimiento del dominio; los **subsistemas vista** son responsables del despliegue ante el usuario, y los **subsistemas controlador** son responsables del manejo de la secuencia de interacciones con el usuario. Los subsistemas modelo se desarrollan de tal forma que no dependan de ningún subsistema vista o controlador. Los cambios en su estado son propagados a los subsistemas vista mediante un protocolo de suscripción/notificación. La arquitectura MVC es un caso especial de la arquitectura de depósito en donde el Modelo implementa la estructura de datos central y los objetos de control dictan el flujo de control.

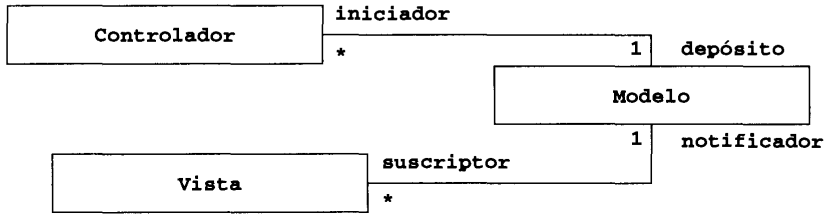


Figura 6-15 Arquitectura Modelo/Vista/Controlador (diagrama de clase UML). El `Controlador` recopila la entrada de los usuarios y envía mensajes al `Modelo`. El `Modelo` mantiene la estructura de datos central. La `Vista` despliega el `Modelo` y es notificada (mediante un protocolo de suscripción/notificación) cuando éste cambia.

Por ejemplo, las figuras 6-16 y 6-17 ilustran la secuencia de eventos que suceden en una arquitectura MVC. La figura 6-17 despliega dos vistas del sistema de archivos. La ventana inferior lista el contenido de la carpeta `Comp-Based Software Engineering` que incluye al archivo `9DesignPatterns2.ppt`. La ventana superior despliega información acerca de ese archivo. El nombre del archivo, `9DesignPatterns2.ppt`, aparece en tres lugares: en ambas ventanas y en el título de la ventana superior.

Supongamos ahora que cambiamos el nombre del archivo a `9DesignPatterns.ppt`. La figura 6-16 muestra la secuencia de eventos:

1. `VistaInfo` y `VistaCarpeta` se suscriben ante cambios de los modelos Archivo que despliegan (cuando se crean).
2. El usuario teclea el nuevo nombre del archivo.
3. El `Controlador`, el objeto responsable de la interacción con el usuario durante los cambios de nombre de archivo, envía una petición al Modelo.
4. El `Modelo` cambia el nombre del archivo y notifica el cambio a todos los suscriptores.
5. Se actualizan `VistaInfo` y `VistaCarpeta` para que los usuarios vean un cambio consistente.

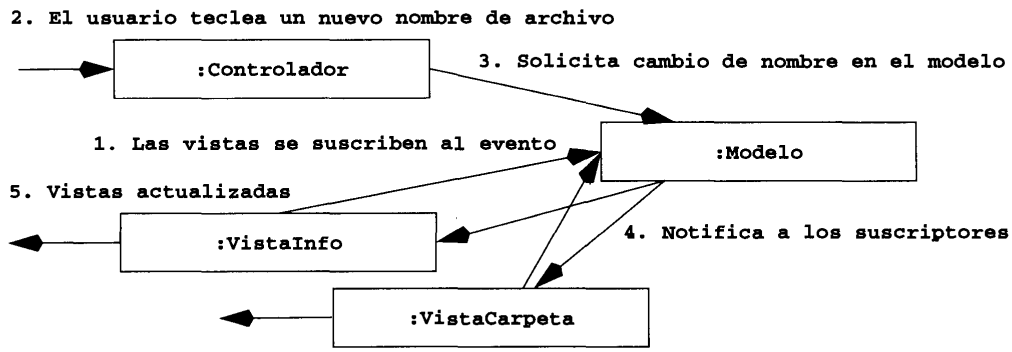


Figura 6-16 Secuencia de eventos en la arquitectura Modelo/Vista/Controlador (diagrama de colaboración UML).

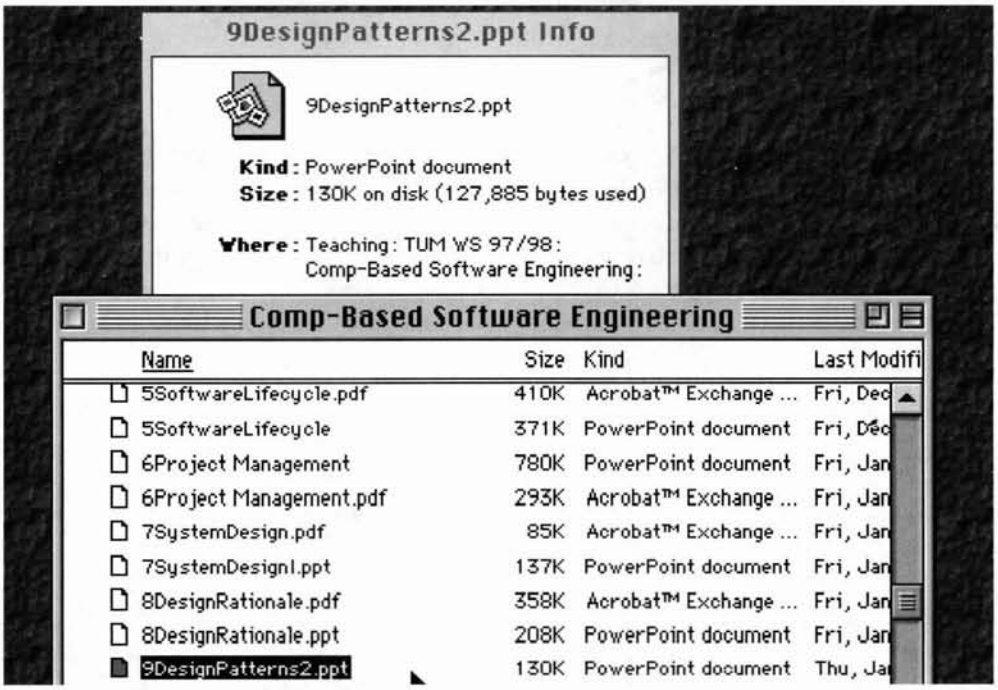


Figura 6-17 Un ejemplo de arquitectura MVC. El "modelo" es el nombre de archivo `9DesignPatterns2.ppt`. Una "vista" es una ventana titulada `Comp-Based Software Engineering` que despliega el contenido de una carpeta que contiene el archivo `9DesignPatterns2.ppt`. La otra "vista" es una ventana llamada `9DesignPatterns2.ppt Info` que despliega información relacionada con el archivo. Si se cambia el nombre del archivo ambas vistas son actualizadas por el "controlador".

La funcionalidad de suscripción y notificación asociada con esta secuencia de eventos se realiza, por lo general, con un patrón Observador (vea la sección A.7). El **patrón Observador** permite que los objetos Modelo y Vista se desacoplen aún más eliminando de ellos las dependencias directas. Para mayores detalles el lector debe consultar [Gamma *et al.*, 1994] y la sección A.7.

Las razones que hay para la separación de Modelo, Vista y Controlador es que las interfaces de usuario, o sea, la Vista y el Controlador, están sujetas a cambios más frecuentes que el dominio de conocimiento, esto es, el Modelo. Además, al eliminar cualquier dependencia del Modelo en la Vista con el protocolo suscripción/notificación, los cambios en las vistas (interfaces de usuario) no tienen ningún efecto en los subsistemas del modelo. En el ejemplo de la figura 6-17 podríamos añadir una vista del sistema de archivos como shell estilo Unix sin tener que modificar el sistema de archivos. En el capítulo 5, *Análisis*, describimos una descomposición similar cuando identificamos los objetos de entidad, frontera y control. Esta descomposición también está motivada por las mismas consideraciones acerca de los cambios.

Las arquitecturas MVC son bastante adecuadas para sistemas interactivos, en especial cuando se necesitan varias vistas del mismo modelo. La MVC puede usarse para mantener la

consistencia a través de datos distribuidos, pero introduce los mismos cuellos de botella de desempeño que otras arquitecturas de depósito.

Arquitectura cliente/servidor

En la arquitectura cliente/servidor (figura 6-18), un subsistema, el servidor, proporciona servicios a instancias de los demás subsistemas llamados los **clientes**, que son responsables de la interacción con el usuario. La petición de un servicio se realiza, por lo general, mediante un mecanismo de llamada a procedimiento remoto o a un agente ("broker") de objetos común (por ejemplo, CORBA o Java RMI). El flujo de control en los clientes y el servidor es independiente, a excepción de la sincronización para administrar las peticiones o para recibir los resultados.

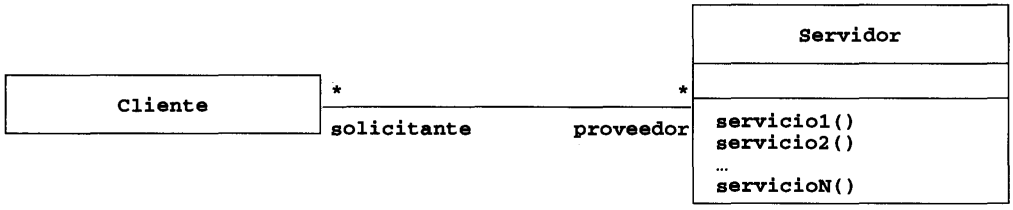


Figura 6-18 Arquitectura cliente/servidor (diagrama de clase UML). Los clientes solicitan servicios de uno o más servidores. Los servidores no tienen conocimiento del cliente. La arquitectura cliente/servidor es una generalización de la arquitectura de depósito.

Un sistema de información con una base de datos central es un ejemplo de la arquitectura cliente/servidor. Los clientes son responsables de la recepción de entrada del usuario, de la realización de revisiones de rango y de la iniciación de transacciones de base de datos una vez que se han recolectado todos los datos necesarios. Luego el servidor es responsable de la realización de las transacciones y de garantizar la integridad de los datos. En este caso, una arquitectura cliente/servidor es un caso especial de la arquitectura de depósito en donde la estructura de datos central es manejada por un proceso. Sin embargo, los sistemas cliente/servidor no están restringidos a un solo servidor. En la World Wide Web un solo cliente puede tener acceso con facilidad a datos de miles de servidores diferentes (figura 6-19).

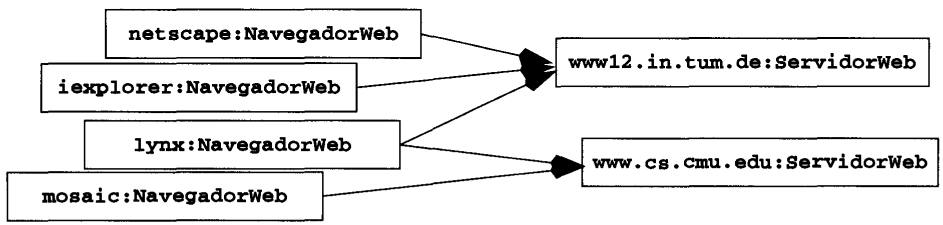


Figura 6-19 La World Wide Web como una instancia de la arquitectura cliente/servidor (diagrama de objetos UML).

Las arquitecturas cliente/servidor son muy adecuadas para los sistemas distribuidos que manejan grandes cantidades de datos.

Arquitectura par a par

Una arquitectura par a par (vea la figura 6-20) es una generalización de la arquitectura cliente/servidor en donde los subsistemas pueden actuar como clientes y como servidores, en el sentido de que cada subsistema puede solicitar y proporcionar servicios. El flujo de control dentro de cada subsistema es independiente de los demás, a excepción de la sincronización de las peticiones.

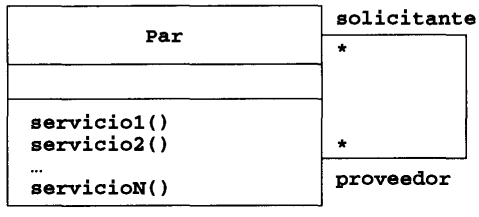


Figura 6-20 Arquitectura par a par (diagrama de clase UML). Los pares pueden solicitar y proporcionar servicios a los demás pares.

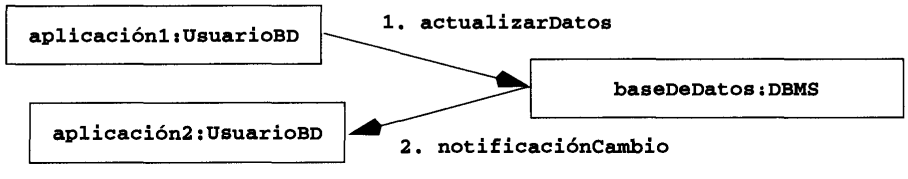


Figura 6-21 Un ejemplo de arquitectura par a par (diagrama de colaboración UML). El servidor de base de datos puede procesar peticiones de las aplicaciones y enviar notificaciones a éstas.

Un ejemplo de una arquitectura par a par es una base de datos en donde, por un lado, acepta peticiones de la aplicación y, por el otro, envía notificaciones a la aplicación cada vez que cambian ciertos datos (figura 6-21). Los sistemas par a par son más difíciles de diseñar que los sistemas cliente/servidor. Introducen la posibilidad de estancamientos y complican el flujo de control.

Arquitectura de tubo y filtro

En la arquitectura de tubo y filtro (vea la figura 6-22) los subsistemas procesan datos recibidos de un conjunto de entradas y envían resultados a otros subsistemas por medio de un conjunto de salidas. A los subsistemas se les llama **filtros** y a las asociaciones entre los subsistemas se les llama **tubos**. Cada filtro sólo sabe el contenido y el formato de los datos recibidos en los tubos de entrada y no el de los filtros que los producen. Cada filtro ejecuta en forma concurrente y la sincronización se realiza por medio de los tubos. La arquitectura de tubo y filtro es modificable: los filtros pueden sustituirse por otros o volver a configurarse para lograr un propósito diferente.

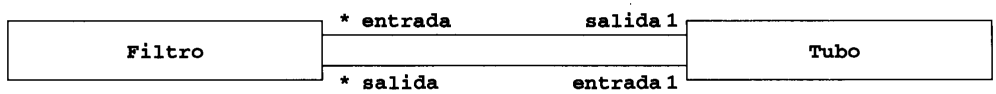


Figura 6-22 Arquitectura de tubo y filtro (diagrama de clase UML). Un `Filtro` puede tener muchas entradas y salidas. Un `Tubo` conecta una de las salidas de un `Filtro` a una de las entradas de otro `Filtro`.

El ejemplo más conocido de la arquitectura de tubo y filtro es el shell Unix. La mayoría de los filtros están escritos de tal forma que leen su entrada y escriben sus resultados en tubos estándar. Esto permite que un usuario de Unix los combine en muchas formas diferentes. La figura 6-23 muestra un ejemplo compuesto por cuatro filtros. La salida de `ps` (estado del proceso) se alimenta a `grep` (búsqueda de patrón) para eliminar todos los procesos que no pertenecen a un usuario específico. La salida de `grep` (es decir, los procesos que posee el usuario) es ordenada luego mediante `sort` y luego enviada a `more`, que es un filtro que despliega su salida en una terminal, pantalla por pantalla.

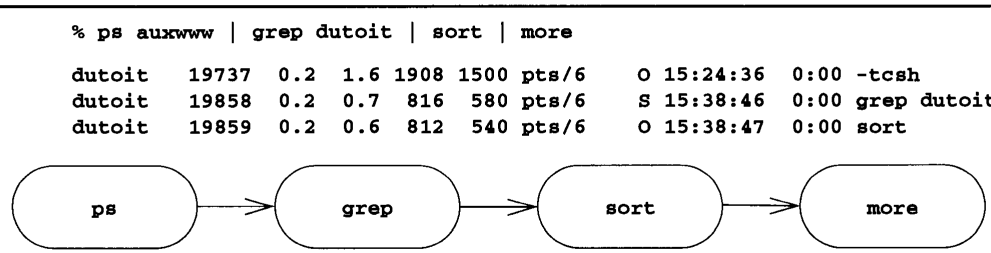


Figura 6-23 Una instancia de la arquitectura de tubo y filtro (comandos Unix y diagrama de actividad UML).

Las arquitecturas de tubo y filtro son adecuadas para sistemas que aplican transformaciones a flujos de datos sin intervención de los usuarios. No son adecuadas para sistemas que requieren interacciones más complejas entre componentes, como un sistema de administración de información o un sistema interactivo.

6.3.6 Diagramas de despliegue UML

Los **diagramas de despliegue UML** se usan para mostrar la relación entre componentes del tiempo de ejecución y los nodos de hardware. Los componentes son entidades autocontenidas que proporcionan servicios a otros componentes o actores. Un servidor Web, por ejemplo, es un componente que proporciona servicios a navegadores Web. Un navegador Web, como Netscape, es un componente que proporciona servicios a los usuarios. Un sistema distribuido puede estar compuesto por muchos componentes interactuantes del tiempo de ejecución.

En los diagramas de despliegue UML los nodos se representan con cuadros que contienen iconos de componentes. Las dependencias entre los componentes se representan con flechas de guiones. La figura 6-24 muestra un ejemplo de un diagrama de despliegue que ilustra dos

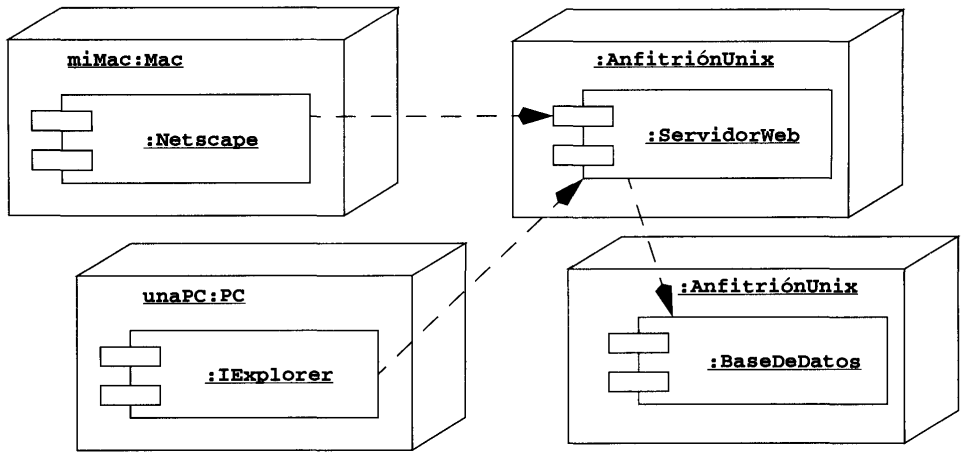


Figura 6-24 Un diagrama de organización UML que representa la asignación de componentes a diferentes nodos y las dependencias entre componentes. Los navegadores Web de la `PC` y la `Mac` pueden acceder a `ServidorWeb`, que proporciona información de una `BaseDeDatos`.

navegadores Web accediendo a un servidor Web. El servidor Web a su vez accede a datos de un servidor de base de datos. En la gráfica de dependencia podemos ver que los navegadores Web no acceden en forma directa a la base de datos en ningún momento.

El diagrama de organización de la figura 6-24 se enfoca en la asignación de componentes a nodos y proporciona una vista de alto nivel de cada componente. Los componentes pueden refinarse para que incluyan información acerca de las interfaces que proporcionan y las clases que contienen. La figura 6-25 ilustra las interfaces `OBTENER` y `ENVIAR` del componente `ServidorWeb` y las clases que contiene.

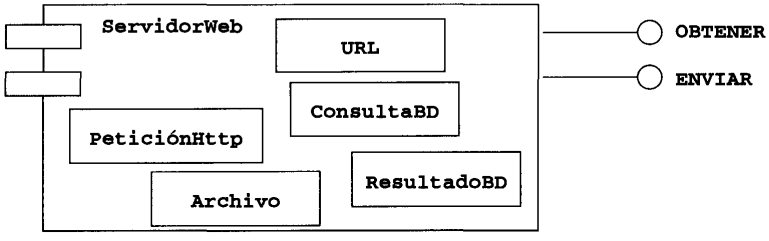


Figura 6-25 Vista refinada del componente `ServidorWeb` (diagrama de despliegue UML). El componente `ServidorWeb` proporciona dos interfaces a los navegadores: un navegador puede solicitar el contenido de un archivo al que hace referencia mediante un URL (`OBTENER`) o enviar el contenido de un formulario (`ENVIAR`). El componente `ServidorWeb` contiene cinco clases: URL, `PeticiónHttp`, `ConsultaBD`, `Archivo` y `ResultadoBD`.

6.4 Actividades del diseño de sistemas: desde los objetos hasta los subsistemas

El diseño de sistemas consiste en la transformación del modelo de análisis en el modelo de diseño, que toma en cuenta los requerimientos no funcionales y las restricciones descritos en el enunciado del problema y el documento de análisis de requerimientos. En la sección 6.3 nos enfocamos en la descomposición en subsistemas y sus propiedades. En esta sección describimos las actividades necesarias para asegurarse que una descomposición en subsistemas tome en cuenta todos los requerimientos no funcionales y prepararse para tomar en cuenta cualquier restricción durante la fase de implementación. Ilustramos estas actividades con un ejemplo, MiViaje, un sistema para la planeación de rutas para los automovilistas. Esto proporcionará un conocimiento más concreto de los conceptos de diseño de sistemas.

Comenzamos con el modelo de análisis de MiViaje. Luego:

- Identificamos los objetivos de diseño para los requerimientos no funcionales (sección 6.4.2)
- Diseñamos una descomposición inicial en subsistemas (sección 6.4.3)
- Establecemos la correspondencia entre los subsistemas y los procesadores y componentes (sección 6.4.4)
- Decidimos el almacenamiento (sección 6.4.5)
- Definimos las políticas de control de acceso (sección 6.4.6)
- Seleccionamos un mecanismo de flujo de control (sección 6.4.7)
- Identificamos condiciones de frontera (sección 6.4.8)

En la sección 6.4.9 examinamos asuntos relacionados con la estabilización del diseño del sistema y, al mismo tiempo, anticipamos los cambios. Por último, en la sección 6.4.10 describimos la manera en que se revisa el modelo de diseño del sistema. Pero primero describimos el modelo de análisis que usamos como punto de partida para el diseño del sistema de MiViaje.

6.4.1 Punto de partida: el modelo de análisis para un sistema de planeación de rutas

Usando MiViaje, un automovilista puede planear un viaje desde una computadora casera poniéndose en contacto con un servicio de planeación de viajes en la Web (`PlanViaje` en la figura 6-26). El viaje se guarda en el servidor para consultas futuras. El servicio de planeación de viajes debe soportar a más de un automovilista.

Nombre del caso de uso	`PlanViaje`
Condición inicial	1. El `Automovilista` activa su computadora casera y se registra en el servicio Web de planeación de viajes.
Flujo de eventos	2. Después de registrarse en forma satisfactoria, el `Automovilista` proporciona las restricciones para el viaje como una secuencia de destinos.
	3. Basándose en una base de datos de mapas, el servicio de planeación calcula la ruta más corta visitando los destinos en el orden especificado. El resultado es una secuencia de segmentos que enlazan una serie de cruces y una lista de indicaciones.
	4. El `Automovilista` puede revisar el viaje añadiendo o eliminando destinos.
Condición final	5. El `Automovilista` guarda el viaje planeado con un nombre en la base de datos del servicio de planeación para recuperarlo más adelante.

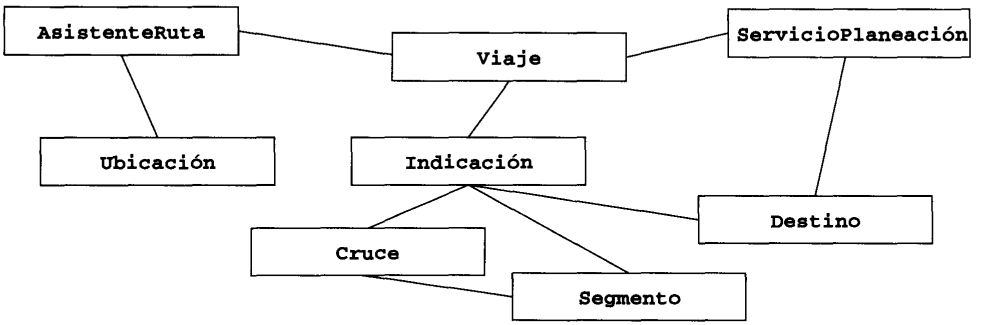
Figura 6-26 El caso de uso `PlanViaje` del sistema MiViaje.

Luego el automovilista va al automóvil e inicia el viaje, mientras la computadora de a bordo le da indicaciones, basada en la información del viaje del servicio de planeación y su ubicación actual indicada por un sistema GPS a bordo (`EjecutarViaje` en la figura 6-27).

Nombre del caso de uso	`EjecutarViaje`
Condición inicial	1. El `Automovilista` arranca su automóvil y se registra en el asistente de ruta a bordo.
Flujo de eventos	2. Después de registrarse en forma satisfactoria, el `Automovilista` especifica el servicio de planeación y el nombre del viaje a ejecutar.
	3. El asistente de ruta a bordo obtiene la lista de destinos, indicaciones, segmentos y cruces desde el servicio de planeación.
	4. Tomando en cuenta la posición actual, el asistente de ruta proporciona al `Automovilista` el siguiente conjunto de indicaciones.
Condición final	5. El `Automovilista` llega a su destino y apaga al asistente de ruta.

Figura 6-27 El caso de uso `EjecutarViaje` del sistema MiViaje.

Realizamos el análisis del sistema MiViaje siguiendo la técnica indicada en el capítulo 5, *Análisis*, y obtenemos el modelo de la figura 6-28.



Cruce	Un `Cruce` es un punto geográfico en donde el automovilista puede escoger entre varios `Segmento`.
Destino	Un `Destino` representa una ubicación a donde quiere ir el automovilista.
Indicación	Teniendo un `Cruce` y un `Segmento` adyacente dados, una `Indicación` describe en lenguaje natural la manera de conducir el automóvil hacia el `Segmento` dado.
Ubicación	Una `Ubicación` es la posición del automóvil como es conocida por el sistema GPS a bordo o por la cantidad de giros de las ruedas.
ServicioPlaneación	Un `ServicioPlaneación` es un servidor Web que puede proporcionar un viaje, vinculando varios destinos en forma de una secuencia de cruces y segmentos.
AsistenteRuta	Un `AsistenteRuta` da `Indicación` al automovilista tomando en cuenta la `Ubicación` actual y el siguiente `Cruce`.
Segmento	Un `Segmento` representa el camino que hay entre dos `Cruce`.
Viaje	Un `Viaje` es una secuencia de `Indicación` entre dos `Destino`.

Figura 6-28 Modelo de análisis para la planeación y ejecución de la planeación de ruta de MiViaje.

Además, durante la obtención de requerimientos nuestro cliente especificó los siguientes requerimientos no funcionales para MiViaje.

Requerimientos no funcionales para MiViaje

1. MiViaje está en contacto con el `ServicioPlaneación` mediante un módem inalámbrico. Puede suponerse que el módem inalámbrico funciona en forma adecuada en su destino inicial.
2. Una vez que ha comenzado el viaje, MiViaje debe dar indicaciones correctas aunque el módem falle y no pueda mantener una conexión con el `ServicioPlaneación`.
3. MiViaje debe minimizar el tiempo de conexión para reducir costos de operación.
4. Sólo es posible la replaneación si se puede hacer la conexión con el `ServicioPlaneación`.
5. El `ServicioPlaneación` puede soportar, al menos, 50 automovilistas diferentes y 1000 viajes.

6.4.2 Identificación de los objetivos de diseño

La definición de los objetivos de diseño es el primer paso del diseño del sistema. Identifica las cualidades en las que debe enfocarse el sistema. Muchos objetivos de diseño pueden inferirse a partir de los requerimientos no funcionales o del dominio de aplicación. Otros habrá que obtenerlos del cliente. Sin embargo, es necesario especificarlos de manera explícita para que cada una de las decisiones de diseño importantes pueda tomarse en forma consistente siguiendo el mismo conjunto de criterios.

Por ejemplo, a la luz de los requerimientos no funcionales de MiViaje, descritos en la sección 6.4.1, identificamos la *confiabilidad* y la *tolerancia a fallas de conectividad* como objetivos de diseño. Luego identificamos la *seguridad* como objetivo de diseño, ya que muchos automovilistas tendrán acceso al mismo servidor de planeación de viajes. Añadimos la *modificabilidad* como objetivo de diseño, ya que queremos proporcionar la capacidad de que los automovilistas seleccionen el servicio de planificación de viajes que deseen. El siguiente cuadro resume los objetivos de diseño que identificamos.

Objetivos de diseño para MiViaje

- **Confiabilidad:** MiViaje debe ser confiable [generalización del requerimiento no funcional 2].
- **Tolerancia a fallas:** MiViaje debe tolerar la falla de pérdida de conectividad con el servicio de rutas [el requerimiento no funcional 2 en otras palabras].
- **Seguridad:** MiViaje debe ser seguro, es decir, no debe permitir que otros automovilistas o usuarios no autorizados tengan acceso a los viajes de otro automovilista [se deduce del dominio de aplicación].
- **Modificabilidad:** MiViaje debe ser modificable para que use servicios de enrutamiento diferentes [anticipación de cambios hechos por los desarrolladores].

En general, podemos seleccionar los objetivos de diseño a partir de una larga lista de cualidades muy deseables. Las tablas 6-1 a 6-5 listan varios criterios de diseño posibles. Estos criterios están organizados en cinco grupos: *desempeño*, *solidez*, *costo*, *mantenimiento* y *criterios de usuario final*. El desempeño, la seguridad y los criterios de usuario final se especifican, por lo general, en los requerimientos o se deducen del dominio de aplicación. Los criterios de costo y mantenimiento son dictados por el cliente y el proveedor.

Los **criterios de desempeño** (tabla 6-1) incluyen los requerimientos de velocidad y espacio impuestos por el sistema. ¿Deberá ser sensible el sistema o deberá realizar un cantidad máxima de tareas? ¿Se dispone de espacio de memoria para optimizaciones de velocidad o se deberá usar la memoria con mesura?

Tabla 6-1 Criterios de desempeño.

Criterio de diseño	Definición
Tiempo de respuesta	¿Qué tan rápido debe atenderse una petición de usuario después de haberla enviado?
Producción	¿Qué tantas tareas puede realizar el sistema en un periodo de tiempo fijo?
Memoria	¿Qué tanto espacio se requiere para que se ejecute el sistema?

Tabla 6-2 Criterios de solidez.

Criterio de diseño	Definición
Robustez	Capacidad para sobrevivir ante datos inválidos del usuario
Confiabilidad	Diferencia entre el comportamiento especificado y el observado
Disponibilidad	Porcentaje del tiempo del sistema en que puede usarse para realizar las tareas normales
Tolerancia a fallas	Capacidad para operar bajo condiciones erróneas
Seguridad	Capacidad para resistir ataques maliciosos
Inocuidad	Capacidad para no poner en riesgo la vida humana, aun en presencia de errores y fallas

Los **criterios de solidez** (tabla 6-2) determinan qué tanto esfuerzo debe ponerse en la minimización de fallas del sistema y sus consecuencias. ¿Con cuánta frecuencia puede fallar el sistema? ¿Qué tan disponible debe estar el sistema ante el usuario? ¿Hay asuntos de seguridad asociados con las fallas del sistema? ¿Hay riesgos de seguridad asociados con el ambiente del sistema?

Los **criterios de costo** (tabla 6-3) incluyen el costo para desarrollar el sistema, entregarlo y administrarlo. Observe que los criterios de costo no sólo incluyen consideraciones de diseño sino también las administrativas. Cuando el sistema está reemplazando a otro antiguo, también tiene que tomarse en cuenta el costo de asegurar la compatibilidad retrospectiva o la transición hacia el nuevo sistema. También hay compromisos entre diferentes tipos de costos, como el costo del desarrollo, el costo del entrenamiento a los usuarios finales, los costos de transición y los costos de mantenimiento. El mantenimiento de la compatibilidad retrospectiva con un sistema anterior puede sumarse al costo de desarrollo, reduciendo el costo de transición.

Tabla 6-3 Criterios de costo.

Criterio de diseño	Definición
Costo de desarrollo	Costo del desarrollo del sistema inicial
Costo de entrega	Costo de la instalación del sistema y del entrenamiento a los usuarios
Costo de actualización	Costo de trasladar los datos del sistema anterior. Este criterio es resultado de los requerimientos de compatibilidad retrospectiva
Costo de mantenimiento	Costo requerido para la corrección de errores y para las mejoras al sistema
Costo de administración	Dinero requerido para la administración del sistema

Los **criterios de mantenimiento** (tabla 6-4) determinan qué tan difícil es cambiar el sistema después de la entrega. ¿Con cuánta facilidad puede añadirse nueva funcionalidad? ¿Con cuánta facilidad pueden revisarse las funciones existentes? ¿Puede adaptarse el sistema a un dominio de aplicación diferente? ¿Qué tanto esfuerzo se requiere para transportar el sistema a una plataforma diferente? Estos criterios son difíciles de optimizar y planear, ya que rara vez se tiene en claro qué tanto tiempo será operativo el sistema y qué tan exitoso será el proyecto.

Tabla 6-4 Criterios de mantenimiento.

Criterio de diseño	Definición
Extensibilidad	¿Qué tan fácil es agregar funcionalidad o nuevas clases al sistema?
Modificabilidad	¿Qué tan fácil es cambiar la funcionalidad del sistema?
Adaptabilidad	¿Qué tan fácil es transportar el sistema a diferentes dominios de aplicación?
Portabilidad	¿Qué tan fácil es transportar el sistema a diferentes plataformas?
Legibilidad	¿Qué tan fácil es comprender el sistema leyendo el código?
Rastreabilidad de requerimientos	¿Qué tan fácil es establecer la correspondencia entre el código y los requerimientos específicos?

Los **criterios del usuario final** (tabla 6-5) incluyen cualidades que son deseables desde el punto de vista de los usuarios que todavía no han sido cubiertas bajo los criterios de desempeño y solidez. Estos incluyen la utilidad (¿qué tan difícil es usar y aprender el software?) y el valor práctico (¿qué tan bien apoya el sistema el trabajo del usuario?). Con frecuencia estos criterios no reciben mucha atención, en especial cuando el cliente que contrata el sistema es diferente a los usuarios del sistema.

Tabla 6-5 Criterios de usuario final.

Criterio de diseño	Definición
Valor práctico	¿Qué tan bien soporta el sistema el trabajo del usuario?
Utilidad	¿Qué tan fácil es para el usuario la utilización del sistema?

Cuando se definen los objetivos de diseño sólo se toman en cuenta en forma simultánea un pequeño subconjunto de estos criterios. Por ejemplo, no es realista desarrollar software que sea inocuo, seguro y barato. Por lo general, los desarrolladores necesitan establecer la prioridad de los objetivos de diseño e intercambiarlos entre sí, así como con los objetivos administrativos conforme el proyecto se atrasa o rebasa el presupuesto. La tabla 6-6 lista varios intercambios posibles.

Tabla 6-6 Ejemplos de intercambios de objetivos de diseño.

Intercambio	Razones
Espacio frente a velocidad	Si el software no satisface los requerimientos de tiempo de respuesta o productividad se puede gastar en más memoria para agilizar el software (por ejemplo, cacheo, más redundancia, etcétera). Si el software no satisface las restricciones de espacio de memoria se pueden comprimir los datos a costa de la velocidad.
Tiempo de entrega frente a funcionalidad	Si el desarrollo se atrasa con respecto a lo planeado, un gerente de proyecto puede entregar una funcionalidad menor a la esperada y entregar a tiempo o entregar toda la funcionalidad en un momento posterior. Los contratos de software, por lo general, ponen más énfasis en la funcionalidad, mientras que los proyectos de software para la venta en paquete ponen más énfasis en la fecha de entrega.
Tiempo de entrega frente a calidad	Si las pruebas se atrasan con respecto a lo planeado, un gerente de proyecto puede entregar el software a tiempo con errores conocidos (y tal vez proporcionar más adelante un parche para corregir cualquier error serio) o entregar el software en un momento posterior con más errores corregidos.
Tiempo de entrega frente a contratación de personal	Si el desarrollo se atrasa con respecto a lo planeado, un gerente de proyecto puede añadir recursos al proyecto para incrementar la productividad. En la mayoría de los casos, esta opción sólo está disponible al inicio del proyecto: la adición de recursos, por lo general, disminuye la productividad mientras se entrena al nuevo personal o se le actualiza. Tome en cuenta que la adición de recursos también incrementará el costo del desarrollo.

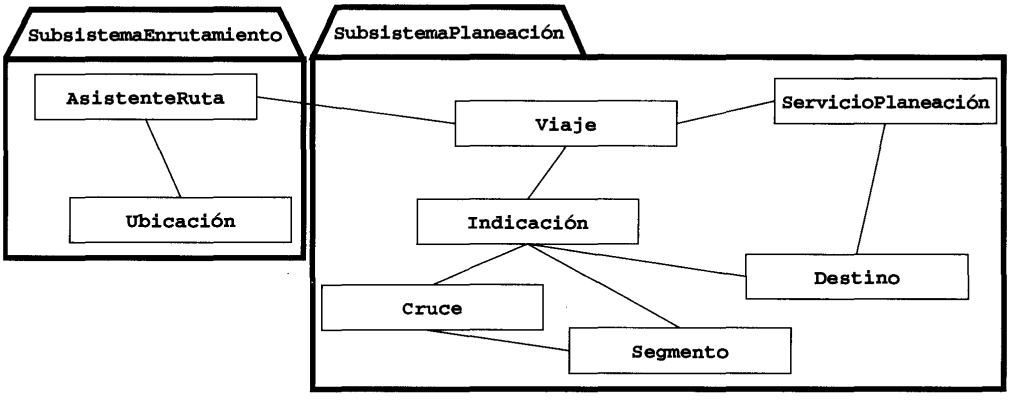
Los objetivos administrativos pueden entrar en conflicto con los objetivos técnicos (por ejemplo, tiempo de entrega frente a funcionalidad). Una vez que se tiene una idea clara de los objetivos de diseño se puede continuar para diseñar una descomposición inicial en subsistemas.

6.4.3 Identificación de subsistemas

Encontrar subsistemas durante el diseño del sistema tiene muchas similitudes con encontrar objetos durante el análisis: es una actividad volátil manejada por heurística. En consecuencia, las técnicas para la identificación de objetos que describimos en el capítulo 5, *Análisis*, como las reglas léxicas de Abbott, son aplicables a la identificación de subsistemas. Es más, la descomposición en subsistemas se revisa en forma constante siempre que se tratan temas nuevos: los subsistemas se combinan en un subsistema, un subsistema complejo se divide en partes y se añaden algunos subsistemas para que se encarguen de nueva funcionalidad. Las primeras iteraciones sobre la descomposición en subsistemas pueden introducir cambios drásticos en el modelo de diseño del sistema. Con frecuencia se manejan mejor mediante lluvia de ideas.

La descomposición en subsistemas inicial debe derivarse de los requerimientos funcionales. Por ejemplo, en el sistema MiViaje identificamos dos grupos principales de objetos: los que están involucrados en los casos de uso `PlanearViaje` y los que están involucrados en el caso de

uso EjecutarViaje. Las clases Viaje, Indicación, Cruce, Segmento y Destino se comparten entre ambos casos de uso. Este conjunto de clases está muy acoplado, ya que se usa como un conjunto para representar un Viaje. Decidimos asignarlas, junto con ServicioPlaneación, al SubsistemaPlaneación, y el resto de las clases se asignan al SubsistemaEnrutamiento (figura 6-29). Esto conduce a una sola asociación que cruza fronteras de subsistema. Observe que esta descomposición en subsistemas sigue una arquitectura de depósito en donde el SubsistemaPlaneación es responsable de la estructura de datos central.



SubsistemaPlaneación	El SubsistemaPlaneación es responsable de la construcción de un Viaje que conecta una secuencia de Destino. El SubsistemaPlaneación también es responsable de responder a peticiones de replanificación de los SubsistemasEnrutamiento.
SubsistemaEnrutamiento	El SubsistemaEnrutamiento es responsable de la descarga de un Viaje desde el ServicioPlaneación y de ejecutarlo dando Indicación al automovilista con base en su Ubicación.

Figura 6-29 Descomposición en subsistemas inicial para MiViaje (diagrama de clase UML).

Otra heurística para la identificación de subsistemas es mantener juntos los objetos relacionados desde el punto de vista funcional. Un punto inicial es tomar los casos de uso y asignar a los subsistemas los objetos participantes que hayan sido identificados en cada uno de ellos. Algunos grupos de objetos, como el grupo Viaje en MiViaje, se comparten y usan para comunicar información entre subsistemas. Podemos crear un nuevo subsistema para alojarlos o asignarlos al subsistema que crea esos objetos.

Heurística para el agrupamiento de objetos en subsistemas

- Asigne los objetos identificados en un caso de uso al mismo subsistema.
- Cree un subsistema dedicado para los objetos que se usan para mover datos entre subsistemas.
- Minimice la cantidad de asociaciones que cruzan fronteras de subsistemas.
- Todos los objetos que están en el mismo subsistema deben estar relacionados desde el punto de vista funcional.

Encapsulado de subsistemas

La descomposición en subsistemas reduce la complejidad del dominio de solución minimizando las dependencias entre clases. El **patrón Fachada** [Gamma *et al.*, 1994] nos permite reducir aún más las dependencias entre clases encapsulando un subsistema con una interfaz unificada simple. Por ejemplo, en la figura 6-30 la clase `Compilador` es una `Fachada` que oculta las clases `GeneradorCódigo`, `Optimizador`, `NodoAnálisis`, `Analizador` y `Léxico`. La `Fachada` proporciona el acceso sólo a los servicios públicos proporcionados por el subsistema y oculta todos los demás detalles, reduciendo en forma efectiva el acoplamiento entre subsistemas.

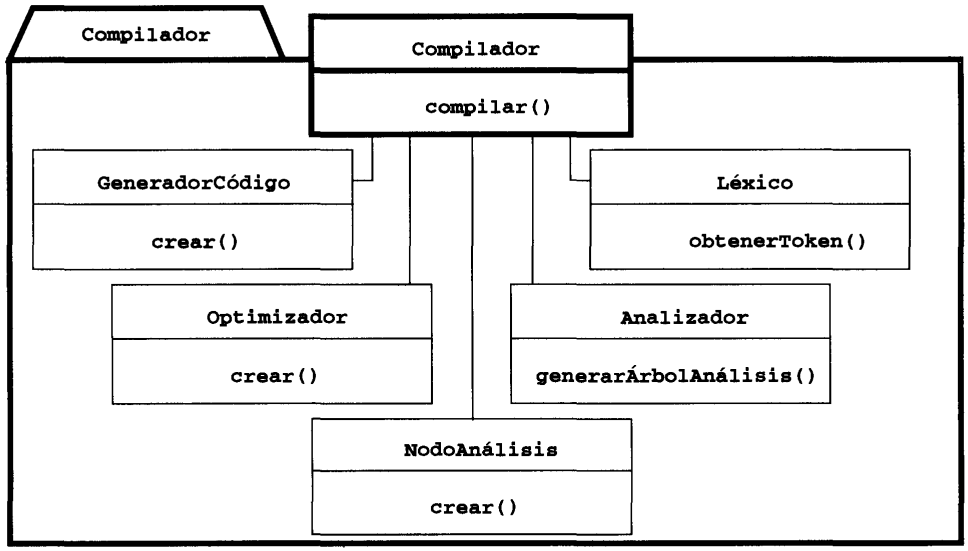


Figura 6-30 Un ejemplo del patrón `Fachada` (diagrama de clase UML).

Los subsistemas que se identifican durante la descomposición inicial en subsistemas resultan, con frecuencia, del agrupamiento de varias clases relacionadas desde el punto de vista funcional. Estos subsistemas son buenos candidatos para el patrón `Fachada` y deben encapsularse bajo una clase.

6.4.4 Correspondencia de subsistemas a procesadores y componentes

Selección de una configuración de hardware y una plataforma

Muchos sistemas ejecutan en más de una computadora y dependen del acceso a una red interna o a Internet. El uso de varias computadoras puede resolver las necesidades de alto desempeño o la interconexión de varios usuarios distribuidos. En consecuencia, necesitamos examinar en forma minuciosa la asignación de subsistemas a computadoras y el diseño de una infraestructura para soportar la comunicación entre subsistemas. Estas computadoras se modelan como nodos en los diagramas de organización UML. Los nodos pueden representar instancias específicas (por

ejemplo, MiMac) o una clase de computadoras (por ejemplo, ServidorWeb).[1] Debido a que la actividad de correspondencia con el hardware tiene un impacto significativo en el desempeño y complejidad del sistema, la realizamos al inicio del diseño del sistema.

La selección de una configuración de hardware también incluye la selección de una máquina virtual sobre la que se construirá el sistema. La máquina virtual incluye el sistema operativo y cualquier componente de software necesario, como un sistema de administración de base de datos o un paquete de comunicaciones. La selección de una máquina virtual reduce la distancia entre el sistema y la plataforma de hardware sobre la que se ejecutará. Entre más funcionalidad proporcionen los componentes habrá menos trabajo involucrado. Sin embargo, la selección de una máquina virtual puede estar restringida por el cliente que adquiere el hardware antes del inicio del proyecto. La selección de una máquina virtual también puede estar restringida por consideraciones de costo: en algunos casos es difícil estimar si la construcción de un componente cuesta más que comprarlo.

En MiViaje deducimos de los requerimientos que el SubsistemaPlaneación y el SubsistemaEnrutamiento se ejecutan en dos nodos diferentes: el primero es un servicio basado en Web en un anfitrión Internet, mientras que el segundo ejecuta en una computadora a bordo. La figura 6-31 ilustra la ubicación del hardware para MiViaje con dos nodos llamados :ComputadoraABordo y :ServidorWeb.

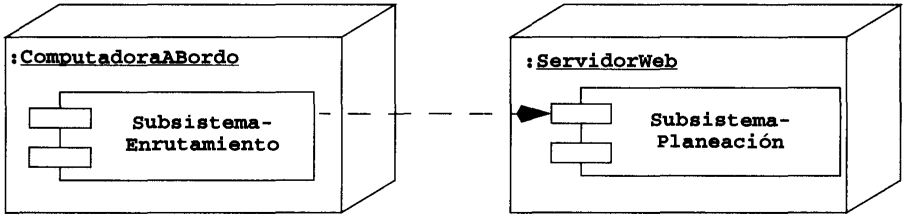


Figura 6-31 Asignación de los subsistemas de MiViaje al hardware (diagrama de organización UML). SubsistemaEnrutamiento ejecuta en :ComputadoraABordo y SubsistemaPlaneación ejecuta en :ServidorWeb.

Seleccionamos una máquina Unix como máquina virtual para el :ServidorWeb, y los navegadores Web Netscape e Internet Explorer como máquinas virtuales para la :ComputadoraABordo.

Asignación de objetos y subsistemas a nodos

Una vez que se ha definido la configuración de hardware y se han seleccionado las máquinas virtuales, se asignan los objetos y subsistemas a los nodos. Esto activa, a veces, la identificación de nuevos objetos y subsistemas para el transporte de datos entre los nodos.

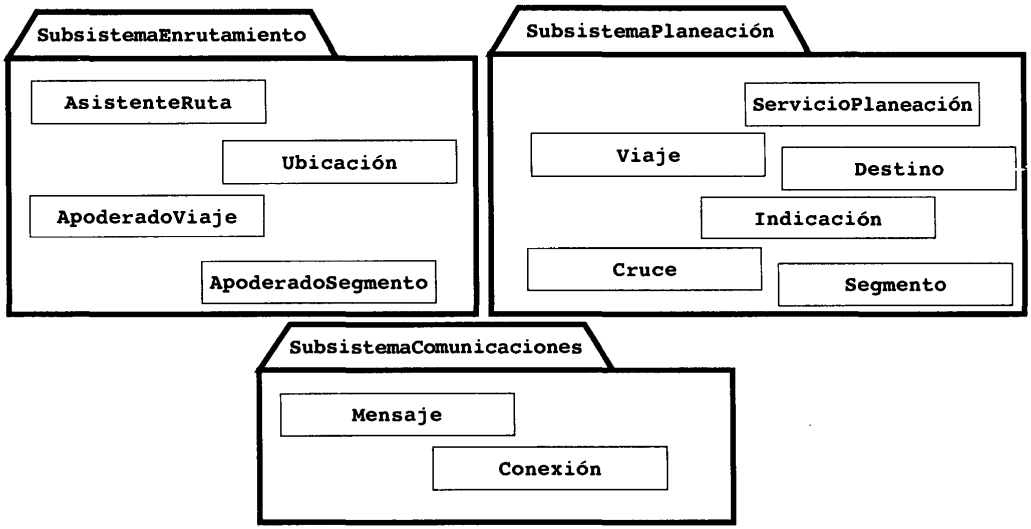
En el sistema MiViaje, SubsistemaEnrutamiento y SubsistemaPlaneación comparten los objetos Viaje, Destino, Cruce, Segmento e Indicación. Las instancias de estas clases necesitan comunicarse mediante módems inalámbricos usando algún protocolo de comunicaciones. Creamos

1. Estos dos casos se distinguen con la convención de denominación usual UML: nombres subrayados para instancias y nombres sin subrayado para clases.

un nuevo subsistema para soportar esta comunicación: SubsistemaComunicaciones, un subsistema que se encuentra en ambos nodos para administrar la comunicación entre los dos.

También observamos que en SubsistemaEnrutamiento sólo se guardan los segmentos que constituyen el viaje planeado. Los segmentos adyacentes que no son parte del viaje se guardan sólo en el SubsistemaPlaneación. Para tomar esto en cuenta necesitamos objetos en el SubsistemaEnrutamiento que puedan actuar como sustitutos de Segmento y Viaje del SubsistemaPlaneación. A un objeto que actúa a nombre de otro se le llama apoderado ("Proxy"). Por lo tanto, creamos dos nuevas clases llamadas ApoderadoSegmento y ApoderadoViaje y las hacemos parte del SubsistemaEnrutamiento. Estos apoderados son ejemplos del patrón de diseño **Apoderado** [Gamma *et al.*, 1994].

En caso de una replaneación por parte del automovilista, esta clase solicitará en forma transparente al SubsistemaComunicaciones que recupere la información asociada con sus Segmento correspondientes del SubsistemaPlaneación. Por último, el SubsistemaComunicaciones se usa para transferir un viaje completo desde SubsistemaPlaneación hacia AsistenteRuta. El modelo de diseño revisado y las descripciones de las clases adicionales se muestran en la figura 6-32.



SubsistemaComunicaciones	El SubsistemaComunicaciones es responsable del transporte de objetos del SubsistemaPlaneación al SubsistemaEnrutamiento.
Conexión	Una Conexión representa un vínculo activo entre el SubsistemaPlaneación y el SubsistemaEnrutamiento. Un objeto Conexión maneja los casos excepcionales asociados con la pérdida de los servicios de red.
Mensaje	Un Mensaje representa a un Viaje y sus Destino, Segmento, Cruce e Indicación relacionados codificados para el transporte.

Figura 6-32 Modelo de diseño revisado para MiViaje (diagrama de clase UML, se omiten las asociaciones por claridad).

En general, la asignación de subsistemas a nodos de hardware nos permite distribuir la funcionalidad y la potencia de procesamiento a donde más se necesita. Por desgracia, también introduce problemas relacionados con almacenamiento, transferencia, réplica y sincronización de datos entre subsistemas. Por esta razón, los desarrolladores también seleccionan los componentes que usarán para desarrollar el sistema.

Encapsulado de componentes

Conforme se incrementa la complejidad de los sistemas y se acorta el tiempo para lanzarlo al mercado, los desarrolladores tienen fuertes incentivos para reutilizar código y apoyarse en componentes proporcionados por vendedores. Los sistemas interactivos, por ejemplo, ahora rara vez se construyen a partir de cero, sino que se desarrollan con juegos de herramientas de interfaz de usuario que proporcionan un amplio rango de diálogos, ventanas, botones u otros objetos estándar de interfaz. Otros proyectos se enfocan en volver a hacer sólo partes de un sistema existente. Por ejemplo, los sistemas de información corporativa, que son costosos de diseñar y construir, necesitan ser actualizados para un nuevo hardware del cliente. Con frecuencia sólo se mejora el lado cliente del sistema para la nueva tecnología y se deja intacto el otro extremo del sistema.[2] Cuando se manejan componentes hechos o código heredado, los desarrolladores tienen que manejar código existente que no pueden modificar y que no ha sido diseñado para ser integrado en sus sistemas.

Podemos manejar los componentes existentes, como el código, encapsulándolos. Este enfoque tiene la ventaja de desacoplar al sistema con respecto al código encapsulado, minimizando, por lo tanto, el impacto del software existente en el diseño. Cuando el código encapsulado está escrito en el mismo lenguaje que el nuevo sistema esto puede realizarse usando un patrón Adaptador.

El **patrón Adaptador** (figura 6-33) se usa para convertir la interfaz de un fragmento de código existente en una interfaz, llamada `NuevaInterfaz`, de acuerdo a lo que espera el subsistema que lo llama. Una clase `Adaptador`, llamada también una envoltura, se introduce para proporcionar el pegamento entre *`NuevaInterfaz`* y `SistemaHeredado`. Por ejemplo, supongamos

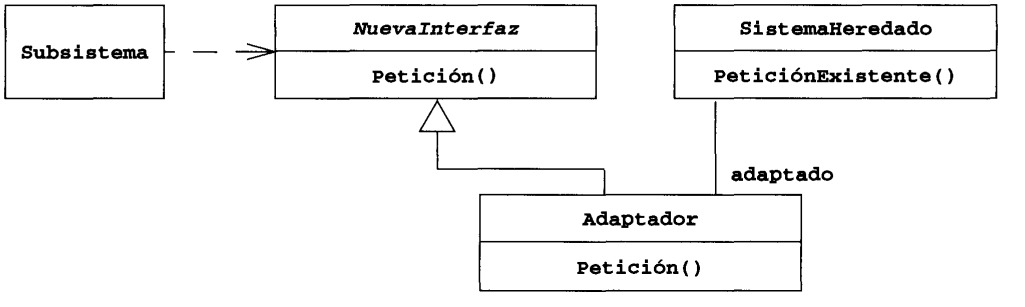


Figura 6-33 Patrón `Adaptador` (diagrama de clase UML). El patrón `Adaptador` se usa para proporcionar una interfaz diferente (*`NuevaInterfaz`*) a un componente existente (*`SistemaHeredado`*).

2. Estos dos casos se distinguen con la convención de denominación usual UML: nombres subrayados para instancias y nombres sin subrayado para clases.

```
/* Interfaz de destino existente */
interface Comparator {
    int compare(Object o1, Object o2);
    /* ... */
}

// Cliente existente
class Array {
    static void sort(Object [] a, Comparator c);
    /* ...*/
}

/* Clase adaptadora existente */
class MyString extends String {
    boolean equals(Object o);
    boolean greaterThan(MyString s);
    /* ... */

}

/* Clase adaptadora nueva */
class MyStringComparator implements Comparator {
    /* ... */
    int compare(Object o1, Object o2) {
        int result;
        if (o1.greaterThan(o2)) {
            result = 1
        } else if (o1.equals(o2)) {
            result = 0;
        } else {
            result = -1;
        }
        return result;
    }
}
```

Figura 6-34 Ejemplo de patrón `Adaptador` (Java). El método estático `sort()` en `Array` toma dos argumentos: un arreglo de `Objects` a ser ordenado y un `Comparator` que define el orden relativo de los elementos. Para ordenar un arreglo de `MyStrings` necesitamos definir un comparador llamado `MyStringsComparator` con la interfaz adecuada. `MyStringsComparator` es un `Adaptador`.

que el cliente es el método `sort()` estático de la clase `Array` de Java (figura 6-34). Este método espera dos argumentos `a`, un `Array` de objetos y `c`, un objeto `Comparator` que proporciona un método `compare()` que define el orden relativo entre los elementos. Supongamos que estamos interesados en el ordenamiento de cadenas de la clase `MyString`, la cual define a los métodos `greaterThan()` y `equals()`. Para ordenar un `Array` de `MyString` necesitamos definir un nuevo comparador, `MyStringComparator`, que proporciona un método `compare()` usando `greaterThan()` y `equals()`. `MyStringComparator` es una clase `Adaptador`.[3]

3. A estos proyectos se les llama proyectos de ingeniería de interfaz (vea el capítulo 4, *Obtención de requerimientos*).

Cuando se encapsula código heredado que está escrito en un lenguaje diferente al del sistema que se está desarrollando, necesitamos manejar las diferencias entre los lenguajes. Aunque se puede realizar la integración de código de dos lenguajes compilados diferentes, puede presentar grandes problemas, en especial cuando uno o ambos lenguajes están orientados a objetos e implementan diferentes semánticas para el envío de mensajes. Esto motivó a estándares como CORBA, el cual define protocolos para permitir la interoperabilidad de objetos distribuidos escritos en lenguajes diferentes. En el caso de arquitecturas cliente/servidor, otras soluciones incluyen el desarrollo de envolturas alrededor de los protocolos de comunicaciones entre procesos.

Los protocolos para la comunicación entre procesos (por ejemplo, tubos y sockets) son proporcionados, por lo general, por el sistema operativo y, por lo tanto, son independientes con relación al lenguaje. En el caso de CORBA y la comunicación entre procesos, el costo del llamado del servicio es mucho más alto que el costo del envío de mensajes entre objetos en el mismo proceso. Cuando se han seleccionado el tiempo de respuesta y otros objetivos de diseño que tienen relación con el desempeño, es necesario evaluar con cuidado el impacto en el desempeño de las envolturas alrededor del código heredado.

Las decisiones sobre tecnología se hacen obsoletas con rapidez. Es probable que el sistema que se está construyendo sobreviva a muchas plataformas, y será transportado y mejorado varias veces durante el mantenimiento. Estas tareas, cuando se realizan durante el mantenimiento, por lo general son costosas, debido a que se ha perdido gran cantidad de la información de diseño. ¿Cuál fue la configuración original del hardware? ¿En cuáles características del sistema de administración de base de datos se apoya este sistema? Los desarrolladores pueden conservar esta información documentando las razones de diseño del sistema, incluyendo las decisiones sobre hardware y componentes. En el capítulo 8, *Administración de la fundamentación*, describimos técnicas para hacerlo.

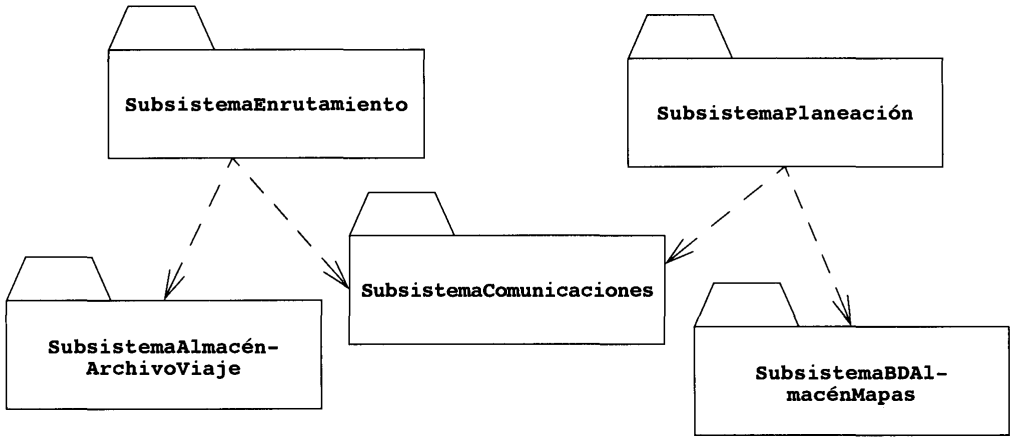
6.4.5 Definición de los almacenes de datos persistentes

Los datos persistentes sobreviven a una sola ejecución del sistema. Por ejemplo, un autor puede guardar su trabajo en un archivo cuando usa un procesador de palabras. Luego puede volver a abrir el archivo unos cuantos días o semanas después. No es necesario que esté ejecutando el procesador de palabras para que exista el archivo. En forma similar, la información relacionada con los empleados, su estado de empleo y sus cheques de pago viven en un sistema de administración de base de datos. Esto permite que todos los programas que operan sobre datos de empleados lo hagan en forma consistente. Además, al guardar los datos en una base de datos se permite que el sistema realice consultas complejas en un gran conjunto de datos (por ejemplo, los registros de varios miles de empleados).

El lugar y la manera en que se guardan los datos tienen un impacto en la descomposición del sistema. En algunos casos, por ejemplo, en una arquitectura de depósito (vea la sección 6.3.5), un subsistema puede estar dedicado por completo al almacenamiento de los datos. La selección de un sistema de administración de base de datos específico también puede tener implicaciones en la estrategia de control general y la administración de la concurrencia.

Por ejemplo, en MiViaje decidimos almacenar el `Viaje` actual en un archivo en un disco removible pequeño para permitir la recuperación del `Viaje` en caso de que el automovilista detenga el automóvil antes de llegar al `Destino` final. El uso de un archivo es la solución más simple y eficiente en este caso, tomando en cuenta que el `SubsistemaEnrutamiento` sólo guardará viajes completos en

el archivo antes de apagarse y cargará el archivo al arrancar. Sin embargo, en el `SubsistemaPlaneación`, los viajes se guardarán en una base de datos. Luego se podrá usar este subsistema para manejar todos los `Viaje` de muchos automovilistas, así como los mapas necesarios para generar los viajes. El uso de una base de datos para este subsistema nos permite realizar consultas complejas sobre esos datos. Añadimos los subsistemas `SubsistemaAlmacénArchivoViaje` y `SubsistemaBDAlmacénMapas` a MiViaje para reflejar esas decisiones, como se muestra en la figura 6-35.



SubsistemaAlmacén-ArchivoViaje	El `SubsistemaAlmacénArchivoViaje` es responsable del almacenamiento de viajes en archivos en la computadora a bordo. Debido a que esta funcionalidad se usa sólo para el almacenamiento de viajes cuando se detiene el vehículo, este subsistema sólo soporta el almacenamiento y la carga rápidos de viajes completos.
SubsistemaBDAlmacén-Mapas	El `SubsistemaBDAlmacénMapas` es responsable del almacenamiento de mapas y viajes en una base de datos para el `SubsistemaPlaneación`. Este subsistema soporta varios automovilistas y agentes de planeación concurrentes.

Figura 6-35 Descomposición en subsistemas de MiViaje después de decidir sobre los asuntos de almacenamiento de datos (diagrama de clase UML, paquetes colapsados por claridad).

En general, primero necesitamos identificar cuáles objetos necesitan ser persistentes. La persistencia de objetos se infiere en forma directa del dominio de aplicación. En MiViaje sólo necesitan guardarse los `Viaje` y sus clases relacionadas. La ubicación del automóvil, por ejemplo, no necesita ser persistente debido a que es necesario recalcularla de manera constante. Luego necesitamos decidir la manera en que deben guardarse estos objetos (por ejemplo, archivo, base de datos relacional o base de datos de objetos). La decisión sobre la administración del almacenamiento es más compleja y, por lo general, está dictada por requerimientos no funcionales: ¿Deben recuperarse rápido los objetos? ¿Se necesitan consultas complejas? ¿Ocupan mucho espacio los objetos (por ejemplo, se guardarán imágenes)? Los sistemas de administración de base de datos proporcionan mecanismos para el control de la concurrencia y consultas eficientes sobre conjuntos de datos grandes.

En la actualidad hay tres opciones realistas para la administración del almacenamiento:

- **Archivos planos.** Los archivos son la abstracción de almacenamiento proporcionada por los sistemas operativos. La aplicación guarda sus datos como una secuencia de bytes y define la manera y el momento en que deben recuperarse los datos. La abstracción de archivo es relativamente de bajo nivel y permite que la aplicación realice una variedad de optimizaciones de tamaño y velocidad. Sin embargo, los archivos requieren que la aplicación se encargue de muchos asuntos, como el acceso concurrente y la pérdida de datos en caso de falla del sistema.
- **Base de datos relacional.** Una base de datos relacional proporciona una abstracción de datos que es más elevada que la de los archivos planos. Los datos se guardan en tablas que se apegan a un tipo predefinido llamado *esquema*. Cada columna de la tabla representa un atributo. Cada renglón representa un concepto de dato como un tuplo de valores de atributo. Varios tuplos de diferentes tablas se usan para representar los atributos de un objeto individual. Las bases de datos relacionales ya se han usado bastante tiempo y son una tecnología madura. El uso de una base de datos relacional introduce un alto costo y, con frecuencia, un cuello de botella.
- **Base de datos orientada a objetos.** Una base de datos orientada a objetos proporciona servicios similares a los de una base de datos relacional. A diferencia de una base de datos relacional, guarda los datos como objetos y asociaciones. Además de proporcionar un nivel de abstracción más alto (reduciendo, por lo tanto, la necesidad de traducir entre objetos y entidades de almacenamiento), las bases de datos orientadas a objetos proporcionan a los desarrolladores herencia y tipos de datos abstractos. Las bases de datos orientadas a objetos son, por lo general, más lentas que las bases de datos relacionales para consultas típicas.

El siguiente cuadro resume los intercambios cuando se seleccionan sistemas de administración de almacenamiento.

Intercambios entre archivos y bases de datos

¿Cuándo se debe escoger un archivo?

- Datos voluminosos (por ejemplo, imágenes).
- Datos temporales (por ejemplo, archivo de imagen de memoria).
- Baja densidad de información (por ejemplo, archivos para archivado, bitácoras de historia).

¿Cuándo se debe escoger una base de datos?

- Accesos concurrentes.
- Accesos a niveles de detalle finos.
- Plataformas múltiples.
- Aplicaciones múltiples sobre los mismos datos.

¿Cuándo se debe escoger una base de datos relacional?

- Consultas complejas sobre los atributos.
- Conjunto grande de datos.

¿Cuándo se debe escoger una base de datos orientada a objetos?

- Amplio uso de asociaciones para recuperar datos.
- Conjunto de datos de tamaño medio.
- Asociaciones irregulares entre objetos.

Encapsulado de almacenes de datos

Una vez que hemos seleccionado un mecanismo de almacenamiento (digamos, una base de datos relacional), podemos encapsularlo en un subsistema y definir una interfaz de alto nivel que sea independiente del vendedor. Por ejemplo, el **patrón Puente** (vea la figura 6-36 y [Gamma *et al.*, 1994]) permite que se desacoplen la interfaz y la implementación de una clase. Esto permite la sustitución de diferentes implementaciones de una clase dada, a veces hasta en el tiempo de ejecución. La clase `Abstracción` define la interfaz visible ante el cliente. La `Implementadora` es una clase abstracta que define los métodos de nivel más bajo disponibles en `Abstracción`. Una instancia de `Abstracción` mantiene una referencia hacia su instancia de `Implementadora` correspondiente. `Abstracción` e `Implementadora` se pueden refinar en forma independiente.

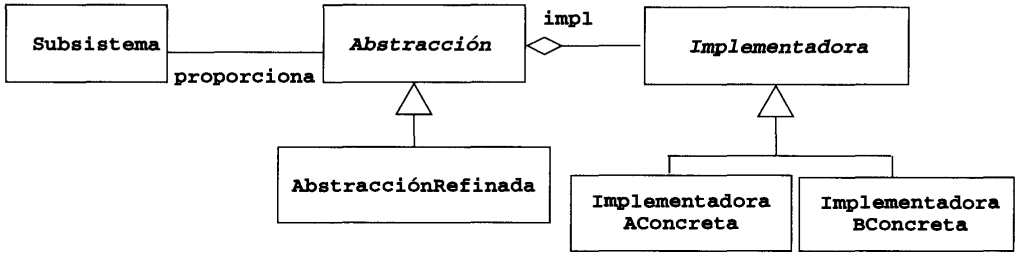


Figura 6-36 Patrón `Puente` (diagrama de clase UML).

Los estándares de conectividad de bases de datos, como ODBC [Microsoft, 1995] y JDBC [JDBC, 1998], proporcionan tales abstracciones para las bases de datos relacionales (vea patrón `Puente` ODBC en la figura 6-37). Sin embargo, observe que, aunque la mayoría de las bases de datos relacionales proporcionan servicios similares, la provisión de una abstracción como ésta reduce el desempeño. Los objetivos de diseño que definimos al inicio de la fase de diseño del sistema nos ayudan a resolver los compromisos de desempeño y modificabilidad.

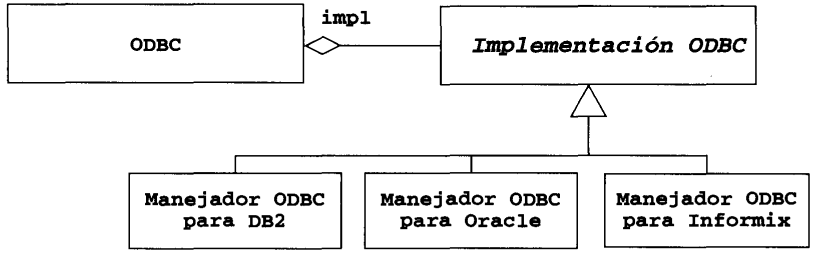


Figura 6-37 Patrón `Puente` para abstraer a los vendedores de bases de datos (diagrama de clase UML). La eliminación de la dependencia de los sistemas con relación a los vendedores de bases de datos proporciona mayor flexibilidad.

6.4.6 Definición del control de acceso

En los sistemas multiusuario, actores diferentes tienen acceso a funcionalidades y datos diferentes. Por ejemplo, un actor de trabajo diario puede acceder sólo a los datos que él crea, mientras que un actor administrador del sistema puede tener acceso ilimitado a los datos del sistema y a los datos de los demás usuarios. Durante el análisis modelamos estas distinciones asociando diferentes casos de uso a diferentes actores. Durante el diseño del sistema modelamos el acceso examinando el modelo de objetos y determinando cuáles objetos están compartidos entre actores y definiendo la manera en que los actores pueden controlar el acceso. Dependiendo de los requerimientos de seguridad del sistema, también definimos la manera en que los actores se autentifican ante el sistema (es decir, cómo prueban los actores quiénes son ante el sistema) y la manera en que deben cifrarse datos seleccionados en el sistema.

Tabla 6-7 Revisión del modelo de diseño procedente de la decisión de autentificar a los `Automovilista` y cifrar el tráfico de comunicaciones. El texto añadido al modelo está en *cursivas*.

SubsistemaComunicaciones	El `SubsistemaComunicaciones` es responsable del transporte de objetos desde el `SubsistemaPlaneación` hacia el `SubsistemaEnrutamiento`. *El SubsistemaComunicaciones usa al Automovilista asociado con el Viaje que se está transportando para seleccionar una clave y cifrar el tráfico de comunicaciones.*
SubsistemaPlaneación	El `SubsistemaPlaneación` es responsable de la construcción de un `Viaje` que conecta una secuencia de `Destino`. El `SubsistemaPlaneación` también es responsable de responder a peticiones de replanificación de los `SubsistemaEnrutamiento`. *Antes de procesar cualquier petición, el SubsistemaPlaneación autentifica al Automovilista del SubsistemaEnrutamiento. El Automovilista autentificado se usa para determinar cuáles Viaje pueden enviarse al SubsistemaEnrutamiento correspondiente.*
Automovilista	*Un Automovilista representa a un usuario autentificado. Lo usan el SubsistemaComunicaciones para recordar las claves asociadas con un usuario y el SubsistemaPlaneación para asociar Viaje con los usuarios.*

Por ejemplo, en MiViaje el almacenamiento de mapas y `Viaje` para muchos automovilistas en la misma base de datos introduce asuntos de seguridad. Debemos asegurarnos que los `Viaje` se envíen sólo al automovilista que los creó. Esto también es consistente con el objetivo de diseño de seguridad que definimos en la sección 6.4.2 para MiViaje. En consecuencia, modelamos a un automovilista con la clase `Automovilista` y lo asociamos con la clase `Viaje`. El `SubsistemaPlaneación` también llega a ser responsable de la autentificación de `Automovilista` antes de enviar `Viaje`. Por último, decidimos cifrar el tráfico de comunicaciones entre el `SubsistemaEnrutamiento` y el `SubsistemaPlaneación`. Esto lo realizará el `SubsistemaComunicaciones`. En la tabla 6-7 se muestran la descripción de la clase `Automovilista` y las descripciones revisadas para el `SubsistemaPlaneación` y el `SubsistemaComunicaciones`. Las revisiones al modelo de diseño se muestran en cursivas.

La definición del control de acceso para un sistema multiusuario, por lo general, es más compleja que en MiViaje. En general, para cada actor necesitamos definir las operaciones a las cuales

puede tener acceso en cada objeto compartido. Por ejemplo, en un sistema de información bancario un cajero puede abonar o cargar dinero a cuentas locales hasta una cantidad predefinida. Si la transacción excede a la cantidad predefinida, un gerente necesita aprobar la transacción. Además, los gerentes y cajeros sólo pueden acceder a cuentas de su propia sucursal, esto es, no pueden acceder a las cuentas de otras sucursales. Los analistas, por otro lado, pueden acceder a información de todas las sucursales de la corporación, pero no pueden hacer transacciones en cuentas individuales.

Modelamos el acceso a las clases con una matriz de acceso. Los renglones de la matriz representan a los actores del sistema. Las columnas representan a las clases cuyo acceso controlamos. A una entrada `(clase, actor)` de la matriz de acceso se le llama un **derecho de acceso**, y lista las operaciones (por ejemplo, `aplicarCargoPequeño()`, `aplicarCargoGrande()`, `examinarSaldo()`, `obtenerDirecciónCliente()`) que puede ejecutar el `actor` en instancias de la `clase`. La tabla 6-8 es un ejemplo de una matriz de acceso para el sistema de información bancario.

Tabla 6-8 Matriz de acceso para un sistema bancario. Los `Cajero` sólo pueden ver cuentas locales, realizar pequeñas transacciones sobre las cuentas y solicitar saldos. Los `Gerente` pueden realizar transacciones más grandes y tener acceso a la historia de la cuenta, además de las operaciones accesibles a los cajeros. Los `Analista` pueden tener acceso a estadísticas de todas las sucursales pero no realizan operaciones en el nivel de cuenta.

Objetos Actores	Corporación	SucursalLocal	Cuenta

`Cajero`		`BuscarCuentaLocal()`	`aplicarCargoPequeño()`
			`aplicarAbonoPequeño()`
			`buscarSaldo()`

`Gerente`		`BuscarCuentaLocal()`	`aplicarCargoPequeño()`
			`aplicarAbonoPequeño()`
			`aplicarCargoGrande()`
			`aplicarAbonoGrande()`
			`examinarSaldo()`
			`examinarHistoria()`

`Analista`	`examinarCargosGlobales()` `examinarAbonosGlobales()`	`examinarCargosLocales()` `examinarAbonosLocales()`	

Podemos representar la matriz de acceso usando uno de tres diferentes enfoques: tabla de acceso global, lista de control de acceso y capacidades:

- Una **tabla de acceso global** representa en forma explícita a cada celda de la matriz como un tuplo (`actor, clase, operación`). La determinación de si un actor tiene acceso a un objeto específico requiere que se busque el tuplo correspondiente. Si no se encuentra el tuplo se niega el acceso.
- Una **lista de control de acceso** asocia una lista de pares (`actor, operación`) con cada `clase` a acceder. Se descartan las celdas vacías. Cada vez que se accede a un objeto se revisa la lista de acceso para buscar el actor y la operación correspondientes. Un ejemplo

de una lista de control de acceso es la lista de invitados a una fiesta. Un mayordomo revisa a los invitados que llegan comparando sus nombres contra los nombres de la lista de invitados. Si hay una concordancia el invitado puede entrar y si no, se le niega la entrada.

- Una **capacidad** asocia un par (`clase, operación`) con un `actor`. Una capacidad proporciona a un actor la obtención de acceso a un objeto de la clase descrita en la capacidad. La negación de la capacidad es equivalente a la negación del acceso. Un ejemplo de una capacidad es una invitación a una fiesta. En este caso, el mayordomo revisa si el invitado que llega tiene una invitación a la fiesta. Si la invitación es válida se admite al invitado y si no, se le niega la entrada. No se necesita ninguna revisión adicional.

La representación de la matriz de acceso también es un asunto de desempeño. Las tablas de acceso globales se usan rara vez ya que requieren mucho espacio. Las listas de control de acceso hacen que sea rápido responder la pregunta "¿Quién tiene acceso a este objeto?", mientras que las listas de capacidad hacen que sea rápido responder a la pregunta "¿A cuáles objetos tiene acceso este actor?"

Cada renglón de la matriz de acceso representa una vista de acceso diferente de las clases que están listadas en las columnas. Todas estas vistas del acceso deben ser consistentes. Sin embargo, por lo general, las vistas de acceso se implementan definiendo una subclase para cada tipo diferente de tuplo (`actor, operación`). Por ejemplo, en nuestro sistema bancario podríamos implementar unas clases `CuentaVistaPorCajero` y `CuentaVistaPorGerente` como subclases de `Cuenta`. Sólo las clases adecuadas están disponibles para el actor correspondiente. Por ejemplo, el software cliente `Analista` no incluiría una clase `Cuenta`, debido a que el `Analista` no tiene acceso a ninguna operación de esta clase. Esto reduce el riesgo de que un error en el sistema dé como resultado la posibilidad de un acceso no autorizado.

Una matriz de acceso sólo representa el **control de acceso estático**. Esto significa que los derechos de acceso pueden modelarse como atributos de los objetos del sistema. En el ejemplo del sistema de información bancario, considere a un actor corredor a quien se asigna en forma dinámica un conjunto de portafolios. Por política, un corredor no puede acceder a los portafolios manejados por otro corredor. En este caso, necesitamos modelar los derechos de acceso en forma dinámica en el sistema y, por lo tanto, a este tipo de acceso se le llama **control de acceso dinámico**. Por ejemplo, la figura 6-38 muestra la manera en que se puede implementar este acceso con un **patrón Apoderado** [Gamma *et al.*, 1994]. Para cada `Portafolio` creamos un `ApoderadoPortafolio` para proteger el `Portafolio` y revisar el acceso. Una asociación `Acceso` entre un `Corredor` legítimo y un `ApoderadoPortafolio` indica a cuáles `Portafolio` tiene acceso el `Corredor`. Para acceder a un `Portafolio` el `Corredor` envía un mensaje al `ApoderadoPortafolio` correspondiente. El `ApoderadoPortafolio` revisa primero si el `Corredor` que llama tiene la asociación correspondiente con el `ApoderadoPortafolio`. Si se otorga el acceso, el `ApoderadoPortafolio` delega el mensaje al `Portafolio`. En caso contrario la operación falla.

En ambos tipos de control de acceso suponemos que conocemos al actor: ya sea el usuario que está ante el teclado o el subsistema que hace la llamada. A este proceso de verificación de la asociación entre la identidad del usuario o subsistema y el sistema se le llama **autentificación**. Un mecanismo de autentificación muy difundido es, por ejemplo, que el usuario especifique un nombre de usuario, conocido por todos, y una contraseña correspondiente conocida sólo por el sistema y guardada en una lista de control de acceso. El sistema protege las contraseñas de los usuarios cifrándolas antes de guardarlas o transmitirlas. Si únicamente un usuario conoce esta combinación

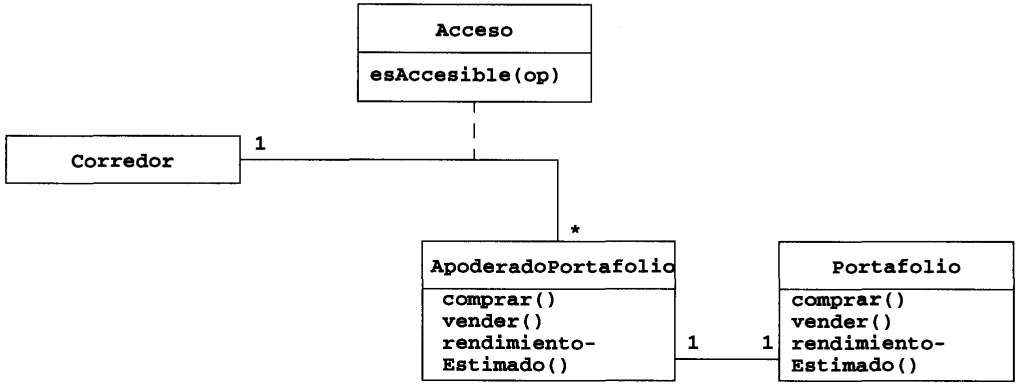


Figura 6-38 El acceso dinámico implementado con un `Apoderado` de protección. La clase de asociación `Acceso` contiene un conjunto de operaciones que puede usar `Corredor` para tener acceso a un `Portafolio`. Cada operación de `ApoderadoPortafolio` revisa primero con la operación `esAccesible()` si el `Corredor` que llama tiene acceso legítimo. Una vez que se ha otorgado el acceso, `ApoderadoPortafolio` delega la operación al objeto `Portafolio` actual. Una asociación `Acceso` puede usarse para controlar el acceso a muchos `Portafolio`.

de nombre de usuario y contraseña podemos suponer que el usuario que está ante el teclado es legítimo. Aunque la autentificación de contraseñas puede hacerse segura con la tecnología actual, tiene muchas desventajas en su utilización: los usuarios escogen contraseñas que son fáciles de recordar y, por lo tanto, fáciles de adivinar. También tienden a escribir su contraseña en notas que conservan cerca de su monitor y, por lo tanto, visibles para muchos otros usuarios no autorizados. Por suerte se dispone de otros mecanismos de autentificación más seguros. Por ejemplo, se puede usar una tarjeta inteligente junto con una contraseña: un intruso necesitaría la tarjeta inteligente y la contraseña para obtener acceso al sistema. Lo que es mejor, se puede usar un sensor biométrico para analizar patrones de vasos sanguíneos en los dedos u ojos de una persona. Un intruso necesitaría la presencia física del usuario legítimo para obtener acceso al sistema, lo cual es mucho más difícil que el simple robo de una tarjeta inteligente.

En un ambiente en donde se comparten recursos entre varios usuarios, la autentificación, por lo general, no es suficiente. En el caso de una red, por ejemplo, es relativamente fácil para un intruso encontrar herramientas para husmear el tráfico de la red, incluyendo paquetes generados por otros usuarios (vea la figura 6-39). Lo que es peor, los protocolos como TCP/IP no fueron diseñados pensando en la seguridad: un intruso puede falsificar paquetes de tal forma que aparezcan como si vinieran de usuarios legítimos.

El **cifrado** se usa para impedir tales accesos no autorizados. Utilizando un algoritmo de cifrado podemos traducir un mensaje, llamado **textollano**, hacia un mensaje cifrado, llamado **texto cifrado**, tal que aunque un intruso intercepte el mensaje no pueda comprenderlo. Sólo el receptor tiene el conocimiento suficiente para descifrar en forma correcta el mensaje, esto es, invertir el proceso original. El proceso de cifrado está parametrizado por una **clave** tal que el método de cifrado y descifrado puede cambiarse con rapidez en caso de que el intruso se las arregle para obtener el conocimiento suficiente para descifrar el mensaje.

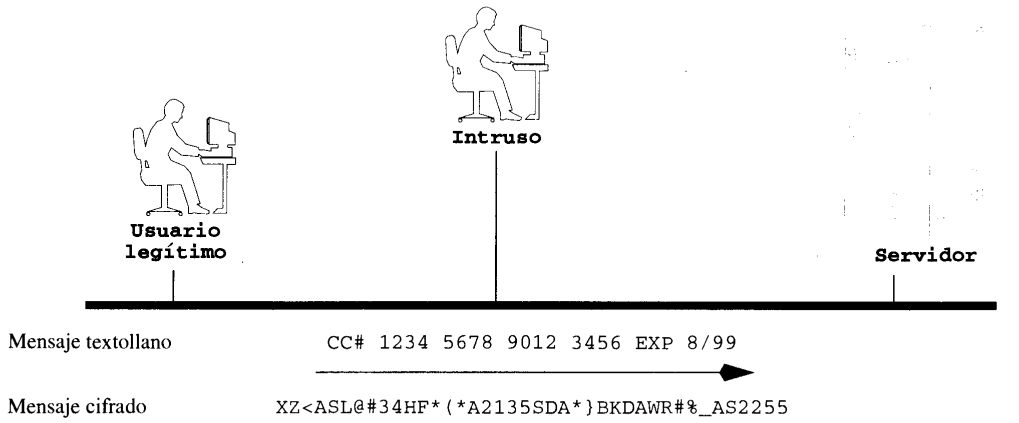


Figura 6-39 Ataque pasivo. Tomando en cuenta la tecnología actual, es relativamente fácil que un intruso pasivo escuche todo el tráfico de la red. Para impedir este tipo de ataque, el cifrado hace que la información que ve el intruso sea difícil de entender.

La autentificación segura y el cifrado son problemas fundamentalmente difíciles. Siempre se deberá seleccionar uno o más algoritmos o paquetes hechos en vez de diseñar los propios (a menos que su negocio sea construir estos paquetes). Muchos de estos paquetes están basados en estándares públicos que son revisados en forma amplia por los académicos y la industria, asegurando, por lo tanto, un nivel relativamente alto de confiabilidad y seguridad.

Encapsulado del control de acceso

El uso de software proporcionado por vendedores introduce un problema de seguridad: ¿Cómo podemos estar seguros de que el software proporcionado no incluye una puerta trasera? Además, una vez que se encuentra una vulnerabilidad en un paquete usado de manera amplia, ¿cómo protegemos el sistema hasta que se disponga de un parche? Podemos usar redundancia para atacar ambos asuntos. Por ejemplo, la Arquitectura Criptográfica Java [JCA, 1998] permite que coexistan varias implementaciones de los mismos algoritmos en el mismo sistema, reduciendo, por lo tanto, la dependencia de un vendedor específico. En términos más generales, podemos usar el **patrón Estrategia** [Gamma *et al.*, 1994] para encapsular varias implementaciones del mismo algoritmo. En este patrón (vea la figura 6-40) la clase abstracta *Estrategia* define la interfaz genérica que deben tener todas las implementaciones del algoritmo encapsulado. Las clases `EstrategiaConcreta` proporcionan implementaciones del algoritmo haciendo a *Estrategia* una subclase. Una clase `Contexto` es responsable del manejo de la estructura de datos sobre los que opera `EstrategiaConcreta`. Las clases `Contexto` y `EstrategiaConcreta` cooperan para proporcionar la funcionalidad necesaria.

Una vez que se proporcionan autentificación y cifrado, se puede implementar con más facilidad el control de acceso específico de la aplicación sobre estos bloques de construcción. En todos los casos, el tratamiento de los asuntos de seguridad es un tema difícil. Cuando se atacan estos asuntos los desarrolladores deben registrar sus suposiciones y describir los escenarios de intrusos que están

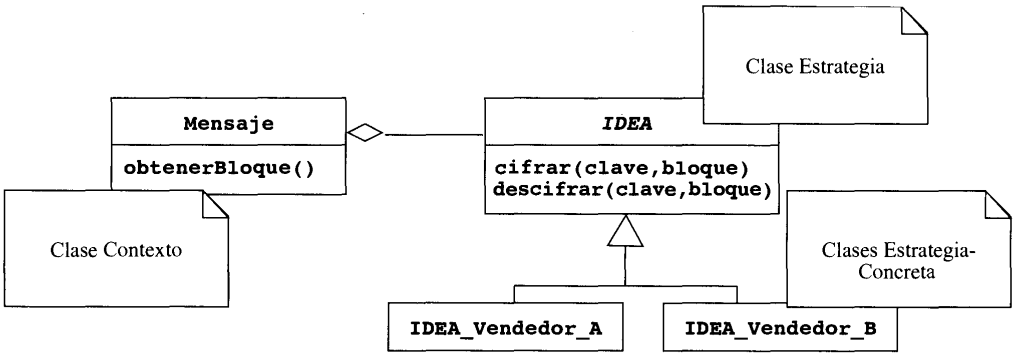


Figura 6-40 Un ejemplo de un patrón `Estrategia` encapsulando varias implementaciones del algoritmo de cifrado IDEA (diagrama de clase UML). Las clases `Mensaje` e *IDEA* cooperan para realizar el cifrado del texto llano. Puede hacerse en forma dinámica la selección de una implementación.

considerando. Cuando se exploran varias alternativas, los desarrolladores deben especificar los problemas de diseño que están tratando de resolver y registrar los resultados de la evaluación. En el siguiente capítulo describimos la manera de hacerlo en forma sistemática usando el modelado de problemas.

6.4.7 Diseño del flujo de control global

El **flujo de control** es el ordenamiento de las acciones en un sistema. En los sistemas orientados a objetos las acciones a ordenar incluyen la decisión de cuáles operaciones deben ejecutarse y en qué orden. Estas decisiones se basan en eventos externos provocados por un actor o en el paso del tiempo.

El flujo de control es un problema de diseño. Durante el análisis, el flujo de control no importa, debido a que asumimos tan sólo que todos los objetos se están ejecutando en forma simultánea, ejecutando operaciones en el momento en que necesitan hacerlo. Durante el diseño del sistema necesitamos tomar en cuenta que no todos los objetos tienen el lujo de estar ejecutando en su propio procesador. Hay tres mecanismos posibles para el flujo de control:

- **Control manejado por procedimientos.** Las operaciones esperan entrada cada vez que necesitan datos de un actor. Este tipo de flujo de control es el más usado en los sistemas heredados y en los sistemas que están escritos en lenguajes procesales. Presenta dificultades cuando se usa con lenguajes orientados a objetos. Como el ordenamiento de operaciones está distribuido entre un gran conjunto de objetos, llega a ser cada vez más difícil determinar el orden de las entradas observando el código (figura 6-41).

```
    Stream in, out;
    String userid, passwd;
/* Se omite la inicialización */
    out.println("Login:");
    in.readln(userid);
    out.println("Contraseña:");
    in.readln(passwd);
    if (!security.check(userid, passwd)) {
        out.println("Falló el Login.");
        system.exit(-1);
    }
/* ...*/
```

Figura 6-41 Un ejemplo de control manejado por procedimientos (Java). El código imprime mensajes y espera entrada del usuario.

- **Control manejado por eventos.** Un ciclo principal espera un evento externo. Cada vez que se tiene disponible un evento se le despacha al objeto adecuado con base en la información asociada con el evento. Este tipo de flujo de control tiene la ventaja de conducir hacia una estructura más simple y centralizar toda la entrada en el ciclo principal. Sin embargo, hace que las secuencias de varios pasos sean más difíciles de implementar (figura 6-42).
- **Hilos.** También se les llama hilos ligeros, para distinguirlos con respecto a los procesos que requieren más sobrecarga de cómputo, y son la variación concurrente del control manejado por procedimientos: el sistema puede crear una cantidad arbitraria de hilos y cada uno responde a un evento diferente. Si un hilo necesita datos adicionales, espera entrada de un actor específico. Es probable que este tipo de flujo de control sea el más intuitivo de los tres mecanismos. Sin embargo, la depuración de software en esta modalidad ("hilado") requiere buenas herramientas de depuración: los sistemas de ejecución de hilos por prioridad introducen indeterminismo y, por lo tanto, hacen que sea difícil encontrar casos de prueba repetibles (figura 6-43).

```
Enumeration subscribers, eventStream;
Subscriber subscriber;
Event event;
EventStream eventStream;
/* ... */
while (eventStream.hasMoreElements) {
    event = eventStream.nextElement();
    subscribers = dispatchInfo.getSubscribers(event);
    while (subscribers.hasMoreElements()) {
        subscriber = subscribers.nextElement()) {
        subscriber.process(event);
    }
}
/* ... */
```

Figura 6-42 Un ejemplo de un ciclo principal para el control manejado por eventos (Java). Un evento se toma de una cola de eventos y se envía a los objetos que están interesados en él.

```
Thread thread;
Event event;
EventHandler eventHandler;
boolean done;
/* ...*/
while (!done) {
    event = eventStream.getNextEvent();
    eventHandler = new EventHandler(event)
    thread = new Thread(eventHandler);
    thread.start();
}
/* ...*/
```

Figura 6-43 Un ejemplo de procesamiento de eventos con hilos (Java). `manejadorEventos` es un objeto dedicado al manejo de `eventos`. Implementa la operación `ejecutar()` la cual es llamada cuando se inicia el `hilo`.

El control manejado por procedimientos es útil para la prueba de subsistemas. Un manejador hace llamadas específicas a los métodos que proporciona el subsistema. Sin embargo, para el flujo de control del sistema final se debe evitar el control manejado por procedimientos.

El intercambio entre el control manejado por eventos y por hilos es más complicado. El control manejado por eventos es más maduro que los hilos. Los lenguajes modernos sólo hasta hace poco comenzaron a proporcionar soporte para la programación con hilos. Conforme se tengan disponibles más herramientas de depuración y se acumule experiencia, el desarrollo de sistemas basados en hilos se hará más fácil. Además, muchos paquetes de interfaz de usuario proporcionan la infraestructura para despachar eventos e imponen este tipo de flujo de control en el diseño. Aunque los hilos son más intuitivos, en la actualidad introducen muchos problemas durante la depuración y las pruebas. Hasta que se disponga de herramientas e infraestructuras más maduras para el desarrollo con hilos, se prefiere el flujo de control manejado por eventos.

Una vez que se selecciona un mecanismo de flujo de control podemos realizarlo con un conjunto de uno o más objetos de control. El papel de los objetos de control es registrar los eventos externos, guardar el estado temporal acerca de ellos y emitir la secuencia adecuada de llamadas a operaciones sobre los objetos de frontera y entidad asociados con el evento externo. Por otro lado, la colocación de las decisiones de flujo de control para un caso de uso en un solo objeto da como resultado código más comprensible, pero por otro lado hace al sistema más adaptable a cambios en la implementación del flujo de control.

Encapsulado del flujo de control

Un ejemplo de encapsulado de control es el **patrón Comando** [Gamma *et al.*, 1994] (vea la figura 6-44). En los sistemas interactivos a menudo es deseable ejecutar, deshacer o guardar las peticiones del usuario sin conocer el contenido de la petición. La clave para el desacoplamiento de las peticiones con respecto a su manejo es convertir las peticiones en objetos de comando, los cuales heredan de una clase abstracta `Comando`. La clase `Comando` define la manera en que el comando se ejecuta, deshace o guarda, mientras que la clase concreta implementa peticiones específicas.

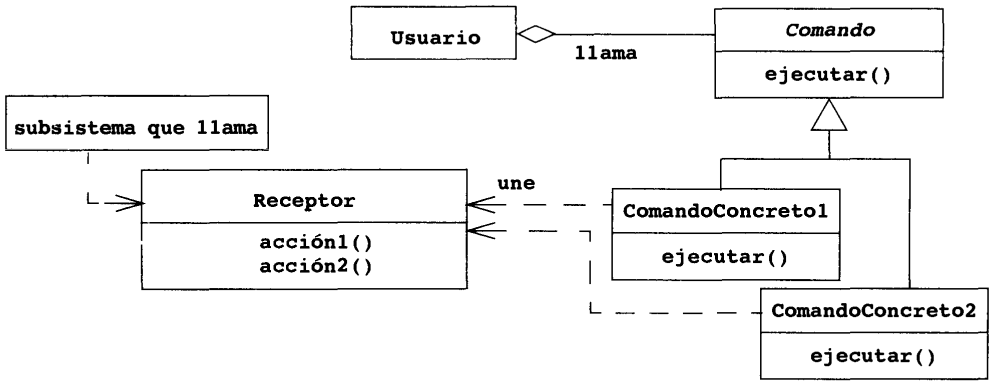


Figura 6-44 Patrón `Comando` (diagrama de clase UML). Este patrón permite el encapsulado del control en forma tal que las peticiones de usuario pueden tratarse en forma uniforme, independiente de la petición específica.

Podemos usar el patrón `Comando` para desacoplar los conceptos de menú con respecto a las acciones (vea la figura 6-45). El desacoplamiento de los conceptos de menú con respecto a las acciones tiene la ventaja de centralizar el flujo de control (por ejemplo, ordenamiento de diálogos) en objetos de control en vez de repartirlo entre objetos de frontera y de entidad. Un `Menú` compuesto de `ConceptoMenú` crea un objeto `Comando` de la clase adecuada cada vez que el usuario selecciona el `ConceptoMenú` correspondiente. La `Aplicación` llama a la operación `ejecutar()` del objeto `Comando` recién creado. Si el usuario desea deshacer la última petición se ejecuta la operación `deshacer()` del último objeto `Comando`. Diferentes objetos `Comando` implementan peticiones diferentes (por ejemplo, `ComandoCopiar` y `ComandoPegar`).

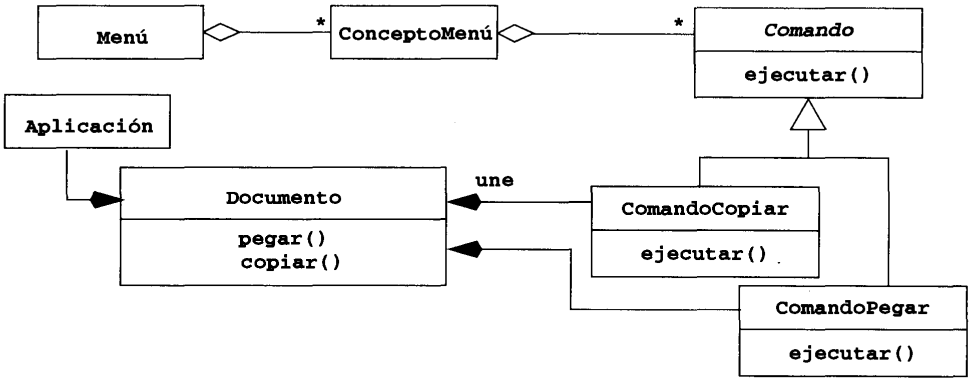


Figura 6-45 Un ejemplo del patrón `Comando` (diagrama de clase UML). En este ejemplo están desacoplados los conceptos de menú y las operaciones sobre documentos. Esto nos permite centralizar el flujo de control en los objetos comando (`ComandoCopiar` y `ComandoPegar`), en vez de repartirlos entre objetos de frontera (`ConceptoMenú`) y objetos de entidad (`Documento`).

6.4.8 Identificación de condiciones de frontera

En las secciones anteriores tratamos el diseño y el refinamiento de la descomposición en subsistemas. Ahora tenemos una mejor idea de la manera de descomponer el sistema, de distribuir casos de uso entre subsistemas, dónde guardar datos y la manera de lograr el control de acceso y garantizar la seguridad. Todavía necesitamos examinar las condiciones de frontera del sistema, esto es, decidir la manera en que el sistema se arranca, se inicia y se apaga, y necesitamos definir la manera de manejar las grandes fallas, como corrupción de datos, causadas por un error de software o por una falla de corriente eléctrica.

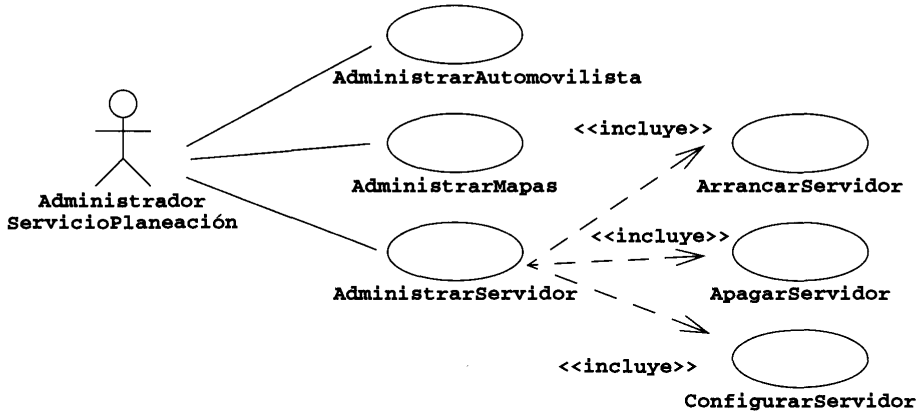


Figura 6-46 Administración de casos de uso para MiViaje (diagrama de caso de uso UML). Se llama a `AdministrarAutomovilista` para añadir, eliminar, modificar o leer datos acerca de los automovilistas (por ejemplo, nombre de usuario y contraseña, bitácora de uso, generación de clave de cifrado). Se llama a `AdministrarMapas` para añadir, eliminar o actualizar mapas que se usan para generar viajes. `AdministrarServidor` incluye todas las funciones necesarias para arrancar y apagar el servidor.

Por ejemplo, ahora tenemos una buena idea de la manera en que debe trabajar MiViaje en estado estable. Sin embargo, todavía no hemos tratado la manera en que se inicia MiViaje. Por ejemplo, ¿cómo se cargan los mapas en el `ServicioPlaneación`? ¿Cómo se instala MiViaje en el automóvil? ¿Cómo sabe MiViaje con cuál `ServicioPlaneación` tiene que conectarse? ¿Cómo se añaden automovilistas al `ServicioPlaneación`? Pronto descubrimos un conjunto de casos de uso que no se han especificado. A estos les llamamos casos de uso de administración del sistema. Los *casos de uso de administración del sistema* especifican el comportamiento de un sistema durante las fases de arranque y paro.

Es común que éstos no se especifiquen durante el análisis o que se traten en forma separada. Por otro lado, muchas funciones de administración del sistema pueden inferirse de los requerimientos de usuario diarios (por ejemplo, registro y borrado de usuarios, administración del control de acceso). Por otro lado, muchas funciones son consecuencia de decisiones de diseño (por ejemplo, tamaño del caché, ubicación del servidor de base de datos, ubicación del servidor de respaldo) y no de decisiones de requerimientos.

Ahora modificamos el modelo de análisis de MiViaje para que incluya los casos de uso de administración. En particular añadimos tres casos de uso: `AdministrarAutomovilista`, para añadir, eliminar y editar automovilistas, `AdministrarMapas` para añadir, eliminar y actualizar mapas que se usan para generar viajes y `AdministrarServidor` para realizar la configuración de rutina, el arranque y el apagado (vea la figura 6-46). `ArrancarServidor`, que es parte de `AdministrarServidor`, se proporciona como ejemplo en la figura 6-47.

Nombre del caso de uso	`ArrancarServidor`
Condición inicial	1. El `AdministradorServicioPlaneación` se registra en la máquina servidora.
Flujo de eventos	2. Después de registrarse en forma satisfactoria, el `AdministradorServicioPlaneación` ejecuta el comando `arrancarServicioPlaneación`.
	3. Si el `ServicioPlaneación` se apagó de modo normal la vez anterior, el servidor lee la lista de `Automovilista` legítimos y el índice de `Viaje` y `Mapa` activos. Si el `ServicioPlaneación` falló se le notifica al `AdministradorServicioPlaneación` y se realiza una revisión de consistencia en `BDAlmacénMapas`.
Condición final	4. El `ServicioPlaneación` está disponible y espera conexiones de los `AsistenteEnrutamiento`.

Figura 6-47 Caso de uso `ArrancarServidor` del sistema MiViaje.

En este caso, la adición de tres casos de uso, es decir, la revisión del modelo de casos de uso, no tiene impacto en la descomposición en subsistemas. Sin embargo, añadimos nuevos casos de uso a los subsistemas existentes: el `SubsistemaBDAlmacénMapas` necesita poder detectar si fue apagado en forma adecuada o no, y necesita poder realizar las revisiones de consistencia y reparar los datos corruptos en caso necesario. Revisamos la descripción de `SubsistemaBDAlmacénMapas` (figura 6-48).

Cuando examinamos condiciones de frontera también necesitamos investigar los casos excepcionales. Por ejemplo, los requerimientos no funcionales de MiViaje especifican que el sistema necesita tolerar fallas en la conexión. Por esta razón, el `AsistenteEnrutamiento` descarga el `Viaje` planeado al `Destino` inicial. También decidimos descargar los `Segmento` que están cercanos al `Viaje` para permitir una replaneación mínima aunque no se disponga de una conexión.

`SubsistemaBDAlmacén-Mapas`	El `SubsistemaBDAlmacénMapas` es responsable del almacenamiento de mapas y viajes en una base de datos para el `SubsistemaPlaneación`. Este subsistema soporta varios automovilistas y agentes de planeación concurrentes. *Cuando arranca, el SubsistemaBDAlmacénMapas detecta si fue apagado en forma adecuada. Si no es así, realiza una revisión de consistencia sobre los Mapa y Viaje, y repara datos corruptos si es necesario.*

Figura 6-48 Descripción revisada de `SubsistemaBDAlmacénMapas` basada en el caso de uso `ArrancarServidor` adicional de la figura 6-47. (Los cambios se indican en *cursivas.)*

En general, una **excepción** es un evento inesperado o error que sucede durante la ejecución del sistema. Las excepciones son causadas por una de tres fuentes diferentes:

- *Un error de usuario.* El usuario, por error o de manera deliberada, da datos que están fuera de los límites. Por ejemplo, una cantidad negativa en una transacción bancaria podría conducir a la transferencia de dinero en la dirección equivocada si el sistema no protege ante tales errores.
- *Una falla de hardware.* El hardware envejece y falla. La falla de un vínculo de red, por ejemplo, puede desconectar en forma momentánea a dos nodos del sistema. Una falla de disco duro puede dar lugar a la pérdida permanente de datos.
- *Un error de software.* Un error puede ocurrir debido a que el sistema, o alguno de sus componentes, contiene un error de diseño. Aunque es difícil la escritura de software libre de errores, los subsistemas individuales pueden anticipar errores de otros subsistemas y protegerse en contra de ellos.

El **manejo de excepciones** es el mecanismo por el cual un sistema trata una excepción. En el caso de un error de usuario el sistema debe desplegar un mensaje de error significativo ante el usuario, de tal forma que pueda corregir el valor dado. En el caso de una falla de vínculo de red el sistema debe guardar su estado temporal para recuperarlo cuando la red vuelva a estar en línea.

El desarrollo de sistemas confiables es un asunto difícil. Con frecuencia, comprometer algo de la funcionalidad facilita el diseño del sistema. En MiViaje suponemos que siempre es posible la conexión en el destino inicial y que la replaneación podría sufrir un impacto por problemas de comunicaciones a lo largo del viaje.

6.4.9 Anticipación del cambio

El diseño del sistema introduce una extraña paradoja en el proceso de desarrollo. Por un lado, queremos construir paredes sólidas entre subsistemas para manejar la complejidad dividiendo el sistema en partes más pequeñas e impedir que los cambios en un subsistema tengan un impacto en los demás. Por otro lado, queremos que la arquitectura de software sea modificable para minimizar el costo de los cambios posteriores. Estos son objetivos en conflicto que no pueden reconciliarse: tenemos que definir una arquitectura al inicio para manejar la complejidad y tenemos que pagar el precio de los cambios más adelante en el proceso de desarrollo. Sin embargo, podemos anticipar los cambios y diseñar para ellos, ya que las fuentes de los cambios posteriores tienden a ser las mismas para la mayoría de los sistemas:

- *Nuevos vendedores o nueva tecnología.* Cuando se usan componentes para construir el sistema hay que anticipar que el componente será reemplazado por uno equivalente de un vendedor diferente. Este cambio es frecuente y, por lo general, es difícil enfrentarlo. El mercado de software es dinámico, y muchos vendedores iniciarán negocios y los abandonarán antes de que se termine el proyecto.
- *Nuevas implementaciones.* Cuando los subsistemas se integran y prueban juntos, el tiempo de respuesta general del sistema es, con mucha frecuencia, más elevado que el establecido o el de los requerimientos de desempeño implícitos: la colocación de un cargo en un sistema de información bancario puede llevarse 2 minutos, un sistema de reservaciones de vuelos puede

llevarse 5 minutos para registrar un vuelo. El desempeño en el nivel de sistema es difícil de predecir y, por lo general, no se optimiza antes de la integración: los desarrolladores se enfocan primero en sus subsistemas. Esto activa la necesidad de estructuras de datos y algoritmos más eficientes y mejores interfaces, a menudo bajo restricciones de tiempo.

- *Nuevas vistas.* La prueba del software con usuarios reales descubre muchos problemas de utilidad. Con frecuencia esto se traduce en la creación de vistas adicionales de los mismos datos.
- *Nueva complejidad del dominio de aplicación.* La organización de un sistema genera ideas de nuevas generalizaciones: un sistema de información bancario para una sucursal puede conducir a la idea de un sistema de información de varias sucursales. Otras veces el dominio mismo incrementa su complejidad: antes los números de vuelo estaban asociados con un avión y sólo un avión. Con la llegada de alianzas de transporte, un avión puede tener ahora varios números de vuelo de diferentes compañías.
- *Errores.* Por desgracia muchos errores de requerimientos se descubren sólo cuando los usuarios reales comienzan a usar el sistema.

Los modernos lenguajes orientados a objetos proporcionan mecanismos que pueden minimizar el impacto del cambio cuando se le anticipa. El uso de delegación y herencia, junto con las clases abstractas, desacopla la interfaz de un subsistema con respecto a su implementación actual. En este capítulo hemos proporcionado ejemplos seleccionados de patrones de diseño [Gamma *et al.*, 1994] que manejan los cambios anteriores. La figura 6-49 resume los patrones y el tipo de cambio contra el que protegen.

`Adaptador` (vea el ejemplo de la figura 6-34)	*Nuevo vendedor, nueva tecnología, nueva implementación.* Este patrón encapsula una parte de código heredado que no fue diseñada para trabajar con el sistema. También limita el impacto de las sustituciones de fragmentos de código heredado con un componente diferente.
`Puente` (vea el ejemplo en la figura 6-37)	*Nuevo vendedor, nueva tecnología, nueva implementación.* Este patrón desacopla la interfaz de una clase con respecto a su implementación. Sirve al mismo propósito que el patrón `Adaptador`, a excepción de que el desarrollador no está restringido por un fragmento de código existente.
`Comando` (vea el ejemplo en la figura 6-45)	*Nueva funcionalidad.* Este patrón desacopla los objetos responsables del procesamiento de comandos con respecto a los comandos mismos. Este patrón protege estos objetos con respecto a cambios a causa de una nueva funcionalidad.
`Observador` (vea el ejemplo en la sección 6.3.5)	*Nuevas vistas.* Este patrón desacopla los objetos de entidad con respecto a sus vistas. Se pueden añadir vistas adicionales sin que tengan que modificarse los objetos de entidad.
`Estrategia` (vea el ejemplo en la figura 6-40)	*Nuevo vendedor, nueva tecnología, nueva implementación.* Este patrón desacopla un algoritmo con respecto a sus implementaciones. Sirve al mismo propósito que los patrones `Adaptador` y `Puente`, a excepción de que la unidad encapsulada es un comportamiento.

Figura 6-49 Patrones de diseño seleccionados y los cambios que anticipan.

Una razón del alto costo de los cambios tardíos en el proceso es la pérdida del contexto de diseño. Los desarrolladores olvidan con mucha rapidez las razones que los llevaron a diseñar desarrollos complicados o estructuras de datos complejas durante las primeras fases del proceso. Cuando se cambia el código más adelante en el proceso, es muy grande la probabilidad de introducir errores en el sistema. Para protegerse contra esas situaciones se deben registrar las suposiciones. Por ejemplo, cuando se usa un patrón de diseño para anticipar un cambio determinado (de la figura 6-49), se deberá registrar cuál cambio se está anticipando. En el capítulo 8, *Administración de la fundamentación*, describimos varias técnicas para registrar las alternativas y decisiones de diseño.

6.4.10 Revisión del diseño de sistemas

Al igual que el análisis, el diseño de sistemas es una actividad en evolución e iterativa. A diferencia del análisis, no hay un agente externo, como el cliente, que revise las iteraciones sucesivas y asegure una mejor calidad. Sin embargo, esta actividad de mejora de la calidad todavía es necesaria, y los gerentes de proyecto y desarrolladores necesitan organizar un proceso de revisión para sustituirlo. Existen varias alternativas, como usar desarrolladores que no han estado involucrados en el diseño del sistema para que actúen como revisores independientes o usar desarrolladores de otro proyecto para que actúen como revisores a la par. Estos procesos de revisión sólo funcionan si los revisores tienen un incentivo en el descubrimiento y reporte de problemas.

Además de satisfacer los objetivos de diseño que se identificaron durante el diseño del sistema, necesitamos asegurarnos que el modelo del diseño del sistema sea correcto, completo, consistente, realista y legible. El modelo del diseño del sistema es **correcto** si se puede establecer una correspondencia entre el modelo de análisis y el modelo del diseño del sistema. Deberán hacerse las siguientes preguntas para determinar si es correcto el diseño del sistema:

- ¿Puede rastrearse cada subsistema de regreso a un caso de uso o un requerimiento no funcional?
- ¿Puede establecerse la correspondencia entre cada caso de uso y un conjunto de subsistemas?
- ¿Puede rastrearse cada objetivo de diseño de regreso a un requerimiento no funcional?
- ¿Se ha tratado cada uno de los requerimientos no funcionales en el modelo de diseño del sistema?
- ¿Cada actor tiene una política de acceso?
- ¿Es consistente con los requerimientos de seguridad no funcionales?

El modelo está **completo** si se han tratado cada uno de los requerimientos y asuntos del diseño del sistema. Deberán hacerse las siguientes preguntas para determinar si el diseño del sistema está completo:

- ¿Se han manejado las condiciones de frontera?
- ¿Hubo pruebas iniciales de los casos de uso para identificar funcionalidad faltante en el diseño del sistema?
- ¿Se han examinado todos los casos de uso y se les ha asignado un objeto de control?
- ¿Se han tratado todos los aspectos del diseño del sistema (es decir, asignación de hardware, almacenamiento persistente, control de acceso, código heredado, condiciones de frontera)?
- ¿Se han definido todos los subsistemas?

El modelo es **consistente** si no contiene ninguna contradicción. Deberán hacerse las siguientes preguntas para determinar si un diseño de sistemas es consistente:

- ¿Se ha establecido la prioridad de los objetivos de diseño en conflicto?
- ¿Hay objetivos de diseño que violen algún requerimiento no funcional?
- ¿Hay varios subsistemas o clases con el mismo nombre?
- ¿Se intercambian colecciones de objetos entre subsistemas en forma consistente?

El modelo es **realista** si se puede implementar el sistema correspondiente. Deberán hacerse las siguientes preguntas para determinar si un diseño de sistemas es realista:

- ¿Hay alguna tecnología o componente nuevo en el sistema? ¿Hay algún estudio que evalúe lo apropiado o robusto de estas tecnologías o componentes?
- ¿Se han revisado los requerimientos de desempeño y confiabilidad en el contexto de la descomposición en subsistemas? Por ejemplo, ¿hay alguna conexión de red en la ruta crítica del sistema?
- ¿Se han abordado los problemas de concurrencia (por ejemplo, contención, estancamientos, exclusión mutua)?

El modelo es **legible** si los desarrolladores que no están involucrados en el diseño del sistema pueden comprender el modelo. Deberán hacerse las siguientes preguntas para asegurarse que el diseño del sistema sea legible:

- ¿Son comprensibles los nombres de los subsistemas?
- ¿Las entidades (por ejemplo, subsistemas, clases, operaciones) que tienen nombres similares indican fenómenos similares?
- ¿Están descritas todas las entidades con el mismo nivel de detalle?

En muchos proyectos encontrará que el diseño del sistema y la implementación se traslapan un poco. Por ejemplo, tal vez se construyan prototipos de algunos subsistemas seleccionados antes de que la arquitectura sea estable para evaluar nuevas tecnologías. Esto conduce a muchas revisiones parciales, en vez de una revisión completa seguida por una aceptación del cliente, como sucede en el análisis. Aunque este proceso produce una mayor flexibilidad, también requiere que los desarrolladores den seguimiento con más cuidado a los asuntos pendientes. Más adelante tendrán que resolverse muchos asuntos difíciles, no debido a su dificultad sino a que cayeron por las grietas de la organización.

6.5 Administración del diseño del sistema

En esta sección tratamos los asuntos relacionados con la administración de las actividades del diseño del sistema. Al igual que en el análisis, el reto principal en la administración del diseño del sistema es mantener la consistencia mientras se usa la mayor cantidad de recursos posible. Al final, la arquitectura de software y las interfaces del sistema deben describir un solo sistema coherente que sea comprensible para una sola persona.

Primero describimos una plantilla de documentos que puede usarse para documentar los resultados del sistema (sección 6.5.1). Luego describimos la asignación de funciones durante el diseño del sistema (sección 6.5.2) y tratamos los asuntos de comunicación durante el diseño del

sistema (sección 6.5.3). Luego tratamos los asuntos de administración relacionados con la naturaleza iterativa del diseño de sistemas (sección 6.5.4).

6.5.1 Documentación del diseño del sistema

El diseño del sistema se plasma en el Documento de Diseño del Sistema (SDD, por sus siglas en inglés). Describe los objetivos de diseño puestos para el proyecto, la descomposición en subsistemas (con diagramas de clase UML), la correspondencia entre el hardware y el software (con diagramas de despliegue UML), la administración de datos, el control de acceso, los mecanismos de flujo de control y las condiciones de frontera. El SDD se usa para definir las interfaces entre equipos de desarrolladores y como referencia cuando se necesitan revisar las decisiones en el nivel de arquitectura. La audiencia del SDD incluye al gerente del proyecto, los arquitectos del sistema (es decir, los desarrolladores que participan en el diseño del sistema) y los desarrolladores que diseñan e implementan cada subsistema. El siguiente es un ejemplo de plantilla para un SDD:

Documento del diseño del sistema

1. Introducción
   - 1.1 Propósito del sistema
   - 1.2 Objetivos de diseño
   - 1.3 Definiciones, siglas y abreviaturas
   - 1.4 Referencias
   - 1.5 Panorama
2. Arquitectura del software actual
3. Arquitectura del software propuesto
   - 3.1 Panorama
   - 3.2 Descomposición en subsistemas
   - 3.3 Correspondencia entre hardware y software
   - 3.4 Administración de datos persistentes
   - 3.5 Control de acceso y seguridad
   - 3.6 Control de software global
   - 3.7 Condiciones de frontera
4. Servicios de subsistemas

Glosario

La primera sección del SDD es una *Introducción*. Su propósito es dar un breve panorama de la arquitectura de software y los objetivos de diseño. También proporciona referencias a otros documentos e información de rastreabilidad (por ejemplo, documento de análisis de requerimientos relacionados, referencias a sistemas existentes, restricciones que tienen un impacto en la arquitectura del software).

La segunda sección, *Arquitectura del software actual*, describe la arquitectura del sistema que se está reemplazando. Si no hay un sistema anterior, esta sección puede reemplazarse con un

estudio de las arquitecturas actuales de sistemas similares. El propósito de esta sección es hacer explícita la información de fondo que usan los arquitectos del sistema, sus suposiciones y asuntos comunes que tratará el nuevo sistema.

La tercera sección, *Arquitectura del software propuesto*, documenta el modelo de diseño del sistema del nuevo sistema. Está dividida en siete subsecciones:

- *Panorama* presenta una vista a ojo de pájaro de la arquitectura del software y describe en forma breve la asignación de funcionalidad de cada subsistema.
- *Descomposición en subsistemas* describe la descomposición en subsistemas y las responsabilidades de cada uno de ellos. Éste es el producto principal del diseño del sistema.
- *Correspondencia entre hardware y software* describe la manera en que se asignan los subsistemas al hardware y los componentes hechos. También lista los asuntos introducidos por varios nodos y la reutilización del software.
- *Administración de datos persistentes* describe los datos persistentes guardados por el sistema y la infraestructura de administración de datos que se requiere para ella. Esta sección incluye, por lo general, la descripción de esquemas de datos, la selección de una base de datos y la descripción del encapsulado de la base de datos.
- *Control de acceso y seguridad* describe el modelo de usuario del sistema desde el punto de vista de una matriz de acceso. Esta sección también describe asuntos de seguridad, como la selección de un mecanismo de autentificación, el uso del cifrado y el manejo de claves.
- *Control de software global* describe la manera en que se implementa el control de software global. En particular, esta sección debe describir la manera en que se inician las peticiones y se sincronizan los subsistemas. Esta sección debe listar y tratar los asuntos de sincronización y concurrencia.
- *Condiciones de frontera* describe el comportamiento del sistema en el arranque, el apagado y en los errores. Si se descubren nuevos casos de uso para la administración del sistema deberán incluirse en el documento de análisis de requerimientos y no en esta sección.

La cuarta sección, *Servicios de subsistemas*, describe los servicios proporcionados por cada subsistema desde el punto de vista de las operaciones. Aunque, por lo general, esta sección está vacía o incompleta en las primeras versiones del SDD, sirve como una referencia para los equipos sobre las fronteras entre sus subsistemas. La interfaz de cada subsistema se deriva de esta sección y se detalla en el documento de diseño de objetos.

El SDD se escribe después de que se ha realizado la descomposición inicial en sistemas, esto es, los arquitectos del sistema no deben esperar hasta que se hayan tomado todas las decisiones del diseño del sistema para publicar el documento. Además, el SDD se actualiza a lo largo del proceso cuando se toman decisiones de diseño o se descubren problemas. El SDD, una vez que se publica, es la línea base y se pone bajo la administración de la configuración. La sección de historia de revisiones del SDD proporciona una historia de los cambios como una lista de cambios, incluyendo al autor responsable del cambio, la fecha de éste y una breve descripción del mismo.

6.5.2 Asignación de responsabilidades

A diferencia del análisis, el diseño del sistema es el reino de los desarrolladores. El cliente y el usuario final se desvanecen en el fondo. Sin embargo, observe que muchas actividades del diseño del sistema activan revisiones al modelo de análisis. El cliente y el usuario regresan al proceso para estas revisiones. En sistemas complejos el diseño del sistema está centrado alrededor del equipo de arquitectura. Éste es un equipo de funcionalidad cruzada compuesto por arquitectos (que definen la descomposición en subsistemas) y desarrolladores seleccionados (que colaboran en la implementación del subsistema). Es crítico que el diseño del sistema incluya personas que estén expuestas a las consecuencias de las decisiones del diseño del sistema. El equipo de arquitectura comienza su trabajo inmediatamente después de que el modelo de análisis es estable, y continúa funcionando hasta el final de la fase de integración. Esto crea un incentivo en el equipo de arquitectura para que anticipe los problemas que se encuentran durante la integración. A continuación se presentan los papeles principales del diseño del sistema:

- El **arquitecto** tiene el papel principal del diseño del sistema. El arquitecto asegura la consistencia en las decisiones de diseño y estilos de interfaz. El arquitecto asegura la consistencia del diseño en la administración de la configuración y los equipos de prueba, en particular en la formulación de la política de administración de la configuración, así como en la estrategia de integración del sistema. Ésta es principalmente un papel de integración que consume información de cada equipo de subsistema. El arquitecto es el líder del equipo de arquitectura de funcionalidad cruzada.
- Los **coordinadores de arquitectura** son los miembros del equipo de arquitectura. Son representantes de los equipos de subsistemas. Transmiten información desde y hacia sus equipos y negocian los cambios de interfaz. Durante el diseño del sistema se enfocan en los servicios de subsistemas, y durante la fase de implementación se enfocan en la consistencia de las APIs.
- Los **papeles de editor de documentos, administrador de configuración** y **revisor** son los mismos que para el análisis.

La cantidad de subsistemas determina el tamaño del equipo de arquitectura. Para sistemas complejos se introduce un equipo de arquitectura para cada nivel de abstracción. En todos los casos debe haber un papel de integración en el equipo para asegurar la consistencia y la comprensión de la arquitectura por un solo individuo.

6.5.3 Comunicación acerca del diseño del sistema

La comunicación durante el diseño del sistema debe ser menos desafiante que durante el análisis: la funcionalidad del sistema ya ha sido definida, los participantes en el proyecto tienen conocimientos similares y, por ahora, ya deben conocerse mejor entre ellos. La comunicación todavía es difícil a causa de nuevas fuentes de complejidad:

- *Tamaño.* La cantidad de asuntos a manejar se incrementa conforme los desarrolladores comienzan a diseñar. La cantidad de asuntos que manejan los desarrolladores se incrementa: cada parte de funcionalidad requiere muchas operaciones sobre muchos objetos. Además, los desarrolladores investigan, a menudo en forma concurrente, varios diseños y tecnologías de implementación.

- *Cambio.* La descomposición en subsistemas y las interfaces de los subsistemas están en flujo constante. Los términos que usan los desarrolladores para nombrar diferentes partes del sistema evolucionan en forma persistente. Si el cambio es rápido puede ser que los desarrolladores no estén discutiendo la misma versión del subsistema, y esto puede conducir a muchas confusiones.
- *Nivel de abstracción.* Las discusiones acerca de los requerimientos pueden concretarse usando maquetas de interfaz y analogías con los sistemas existentes. Las discusiones sobre la implementación llegan a ser concretas cuando se dispone de la integración y los resultados de las pruebas. Las discusiones sobre el diseño del sistema rara vez son concretas, ya que las consecuencias de las decisiones de diseño se sufren más adelante durante la implementación y las pruebas.
- *Renuencia a enfrentar problemas.* El nivel de abstracción de la mayoría de las discusiones también puede facilitar que se posponga la resolución de asuntos difíciles. Una resolución típica de problemas de control con frecuencia es "volvamos a ver este asunto durante la implementación". Aunque, por lo general, es deseable postergar determinadas decisiones de diseño, como las estructuras de datos internos y algoritmos usados por cada subsistema, por ejemplo, ninguna decisión que tenga impacto sobre la descomposición del sistema y las interfaces de subsistemas debe postergarse.
- *Objetivos y criterios conflictivos.* Los desarrolladores individuales con frecuencia optimizan criterios diferentes. Un desarrollador con experiencia en el diseño de interfaz de usuario estará predispuesto hacia la optimización del tiempo de respuesta. Un desarrollador con experiencia en base de datos puede optimizar la producción. Estos objetivos conflictivos, en especial cuando son intrínsecos, dan como resultado que los desarrolladores jalen la descomposición del sistema en direcciones diferentes y esto conduzca a inconsistencias.

Las mismas técnicas que tratamos en el análisis (vea la sección 5.5.3) pueden aplicarse durante el diseño del sistema:

- *Identificar y asignar prioridad a los objetivos de diseño del sistema y hacerlos explícitos* (vea la sección 6.4.2). Si los desarrolladores que tienen a su cargo el diseño del sistema tienen entrada en este proceso se les facilitará encargarse de estos objetivos de diseño. Los objetivos de diseño también proporcionan un marco de trabajo objetivo contra el cual se pueden evaluar las decisiones.
- *Poner a disposición de todos los interesados la versión actual de la descomposición del sistema.* Un documento vivo distribuido por medio de Internet es una forma para lograr la distribución rápida. El uso de una herramienta de administración de configuración para mantener los documentos de diseño del sistema ayuda a los desarrolladores a que identifiquen cambios recientes.
- *Mantener un glosario actualizado.* Al igual que en el análisis, la definición explícita de términos reduce la falta de comprensión. Cuando se identifican y modelan subsistemas hay que proporcionar definiciones además de los nombres. Un diagrama UML sólo con los nombres de los subsistemas no es suficiente para apoyar una comunicación efectiva. Una definición breve y sustancial debe acompañar a cada nombre de subsistema y clase.

- *Enfrentar problemas de diseño.* El retraso de las decisiones de diseño puede ser benéfico cuando se necesita más información antes de comprometerse a decisiones de diseño. Sin embargo, este enfoque puede impedir la confrontación de problemas de diseño difíciles. Antes de plantear un asunto se deben explorar y describir varias alternativas posibles y justificar el retraso. Esto asegura que los asuntos que se posponen lo hagan sin tener un impacto serio en la descomposición del sistema.
- *Iterar.* Excursiones seleccionadas en la fase de implementación pueden mejorar el diseño del sistema. Por ejemplo, se pueden evaluar nuevas características de un componente proporcionado por un vendedor implementando un prototipo vertical (vea la sección 6.5.4) para la funcionalidad que es más probable que se beneficie con la característica.

Por último, sin importar qué tanto esfuerzo se gaste en el diseño del sistema, la descomposición del sistema y las interfaces de subsistemas cambiarán, casi con seguridad, durante la implementación. Conforme se tiene disponible nueva información sobre tecnologías de implementación, los desarrolladores tienen una comprensión más clara del sistema y se descubren alternativas de diseño. Los desarrolladores deben anticipar los cambios y reservar algo de tiempo para actualizar el SDD antes de la integración del sistema.

6.5.4 Iterando sobre el diseño del sistema

Así como sucede con los requerimientos, el diseño del sistema se da a través de iteraciones y cambios sucesivos. Sin embargo, el cambio debe ser controlado para impedir el caos, en especial en proyectos complejos que incluyen a muchos participantes. Distinguimos tres tipos de iteraciones durante el diseño del sistema. Primero, las decisiones principales al inicio del diseño del sistema tienen un impacto en la descomposición en subsistemas conforme se inicia cada una de las diferentes actividades del diseño del sistema. Segundo, las revisiones a las interfaces de los subsistemas suceden cuando se usan los prototipos de evaluación para evaluar asuntos específicos. Tercero, los errores y omisiones que se descubren más adelante activan cambios a las interfaces de subsistemas y, a veces, a la descomposición del sistema mismo.

El primer conjunto de iteraciones se maneja mejor mediante la lluvia de ideas cara a cara y electrónica. Las definiciones todavía están fluyendo, los desarrolladores todavía no tienen una visión del sistema completo y debe darse la máxima importancia a la comunicación a expensas de la formalidad o los procedimientos. Con frecuencia, en proyectos basados en equipo, la descomposición inicial del sistema se diseña antes de que se termine el análisis. La descomposición temprana del sistema permite que se asigne la responsabilidad de sistemas diferentes a equipos diferentes. Se debe motivar el cambio y la exploración, aunque sea para ampliar la comprensión compartida de los desarrolladores o para generar evidencias que den soporte al diseño actual. Por esta razón, no deberá haber un proceso burocrático de cambio formal durante esta fase.[4]

El segundo conjunto de iteraciones está orientado a resolver asuntos difíciles y enfocados, como la selección de un vendedor o tecnología específicos. La descomposición en subsistemas ya es estable (debe ser independiente de vendedores y tecnología, vea la sección 6.4.9) y la mayoría

4. Un prototipo vertical implementa por completo una funcionalidad restringida (por ejemplo, objetos de interfaz, control y entidad para un caso de uso), mientras que un prototipo horizontal implementa en forma parcial un amplio rango de funcionalidad (por ejemplo, objetos de interfaz para varios casos de uso).

de estas exploraciones están orientadas a identificar si un paquete específico es adecuado o no para el sistema. Durante este periodo los desarrolladores también pueden realizar un prototipo vertical para un caso de uso crítico a fin de probar lo adecuado de la descomposición. Esto permite que se descubran y traten lo más pronto posible asuntos de flujo de control. De nuevo, no es necesario un proceso de cambio formal. Una lista de asuntos pendientes y su estado puede ayudar a que los desarrolladores propaguen rápido el resultado de una investigación de tecnología.

El tercer conjunto de iteraciones remedia problemas de diseño que se descubren más adelante en el proceso. Aunque los desarrolladores tratarían de evitar estas iteraciones, ya que tienden a incurrir en altos costos e introducen muchos errores nuevos en el sistema, deben anticipar cambios tardíos en el desarrollo. La anticipación de iteraciones tardías incluye la documentación de dependencias entre subsistemas, las razones de diseño para interfaces de subsistemas y cualquier rodeo que es probable que se rompa al haber un cambio. Los cambios deben manejarse con cuidado, y se deberá poner en su lugar un proceso de cambio similar al del seguimiento de cambios de requerimientos.

Podemos lograr la estabilización progresiva de la descomposición en subsistemas usando el concepto de ventana de diseño. Para motivar el cambio, y al mismo tiempo controlarlo, los asuntos críticos se dejan pendientes sólo durante un tiempo especificado. Por ejemplo, la plataforma de hardware y software que será el destino del sistema deberá resolverse lo más pronto posible en el proyecto para que las decisiones de compra del hardware puedan realizarse a tiempo para los desarrolladores. Sin embargo, las estructuras de datos internas y los algoritmos pueden dejarse pendientes hasta después de la integración, permitiendo que los desarrolladores los revisen con base en las pruebas de desempeño. Una vez que ha pasado la ventana de diseño, el asunto debe resolverse y sólo se volverá a abrir en una iteración subsecuente.

Con el ritmo cada vez más rápido de la innovación tecnológica se pueden anticipar muchos cambios cuando una parte dedicada de la organización es responsable de la administración de tecnología. Los gerentes de tecnología revisan nuevas tecnologías, las evalúan y acumulan conocimiento que se utiliza durante la selección de componentes. Con frecuencia, los cambios suceden tan rápido que las compañías no están conscientes de las tecnologías que ellas mismas proporcionan.

6.6 Ejercicios

1. La descomposición de un sistema en subsistemas reduce la complejidad que tienen que manejar los desarrolladores simplificando las partes e incrementando su coherencia. La descomposición de un sistema en partes más simples da como resultado, por lo general, un incremento en un tipo diferente de complejidad: partes más simples también significan una mayor cantidad de partes e interfaces. Si la coherencia es el principio que guía a los desarrolladores para descomponer un sistema en partes pequeñas, ¿cuál principio competidor los lleva a mantener pequeña la cantidad total de partes?
2. En la sección 6.4.2 clasificamos los objetivos de diseño en cinco categorías: desempeño, solidez, costo, mantenimiento y criterios de usuario final. Asigne una o más categorías a cada uno de los siguientes objetivos:
    - A los usuarios se les debe dar retroalimentación en menos de un segundo después de que emitan cualquier comando.
    - El `DistribuidorBoletos` debe ser capaz de emitir boletos de tren aunque haya una falla de red.

- El gabinete del `DistribuidorBoletos` debe permitir que se instalen nuevos botones por si se incrementa la cantidad de tarifas diferentes.
- La `MáquinaCajeroAutomático` debe resistir ataques de diccionario (es decir, usuarios que intentan descubrir un número de identificación mediante intentos sistemáticos).
- La interfaz de usuario del sistema debe impedir que los usuarios emitan comandos en orden erróneo.

3. Considere un sistema que incluya un servidor Web y dos servidores de bases de datos. Los dos servidores de bases de datos son idénticos: el primero actúa como servidor principal y el segundo como respaldo redundante en caso de que falle el primero. Los usuarios usan navegadores Web para tener acceso a los datos mediante el servidor Web. También tienen la opción de usar un cliente propio que acceda a las bases de datos en forma directa. Trace un diagrama de despliegue UML que represente la correspondencia entre hardware y software de este sistema.
4. Considere un sistema para reporte de problemas heredado basado en fax para un fabricante de aeronaves. Usted es parte de un proyecto de reingeniería que reemplaza la parte medular del sistema con un sistema basado en computadora, el cual incluye una base de datos y un sistema de notificación. El cliente requiere que el fax siga siendo un punto de entrada para el reporte de problemas. Usted propone un punto de entrada de correo electrónico. Describa una descomposición en subsistemas, y de ser posible un patrón de diseño, que permita ambas interfaces.
5. Usted está diseñando las políticas de control de acceso para una tienda al menudeo basada en Web. Los clientes acceden a la tienda por medio de Web, examinan información de productos, dan su dirección e información de pago y compran productos. Los proveedores pueden añadir nuevos productos, actualizar la información de productos y recibir pedidos. El propietario de la tienda asigna los precios al menudeo, hace ofertas personalizadas a los clientes con base en sus perfiles de compra y proporciona servicios de comercialización. Usted tiene que manejar tres actores: `AdministradorTienda`, `Proveedor` y `Cliente`. Diseñe una política de control de acceso para los tres actores. Los `Cliente` pueden crearse mediante Web, pero los `Proveedor` son creados por el `AdministradorTienda`.
6. Seleccione un mecanismo de control de flujo que encuentre más apropiado para cada uno de los siguientes sistemas. Debido a que en la mayoría de los casos son posibles varias alternativas, justifique las que tome.
   - Un servidor Web diseñado para soportar altas cargas.
   - Una interfaz gráfica de usuario para un procesador de palabras.
   - Un sistema incrustado de tiempo real (por ejemplo, un sistema de guía en un lanzador de satélites).
7. ¿Por qué se describen durante el diseño del sistema los casos de uso que describen condiciones de frontera (en vez de hacerlo durante la obtención de requerimientos o el análisis)?
8. Usted está desarrollando un sistema que guarda sus datos en un sistema de archivo Unix. Anticipa que transportará las versiones futuras del sistema a otros sistemas operativos que proporcionan diferentes sistemas de archivo. ¿Cuál patrón de diseño utilizaría para protegerse ante este cambio?

Referencias

[Bass *et al.*, 1999] L. Bass, P. Clements y R. Kazman, *Software Architecture in Practice.* Addison-Wesley, Reading, MA, 1999.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* 2a. ed. Benjamin/Cummings, Redwood City, CA, 1994.

[Day y Zimmermann, 1983] J. D. Day y H. Zimmermann, "The OSI Reference Model", *Proceedings of the IEEE*, diciembre de 1983, vol. 71, págs. 1334–1340.

[Erman *et al.*, 1980] L. D. Erman, F. Hayes-Roth *et al.*, "The Hearsay-II Speech-Understanding System: Integrating knowledge to resolve uncertainty". *ACM Computing Surveys*, 1980, vol. 12, núm. 2, págs. 213–253.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Reading, MA, 1997.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[JCA, 1998] *Java Cryptography Architecture.* JDK Documentation, Javasoft, 1998.

[JDBC, 1998] *JDBC™—Connecting Java and Databases*, JDK Documentation, Javasoft, 1998.

[Microsoft, 1995] Microsoft, "Chapter 9: Open Database Connectivity (ODBC) 2.0 fundamentals", *Microsoft Windows Operating Systems and Services Architecture,* Microsoft Corp., 1995.

[Mowbray y Malveau, 1997] T. J. Mowbray y R. C. Malveau, *CORBA Design Patterns.* Wiley, Nueva York, 1997.

[Nye y O'Reilly, 1992] A. Nye y T. O'Reilly. *X Toolkit Intrinsics Programming Manual: OSF/Motif 1.1 Edition for X11 Release 5 The Definitive Guides to the X Windows Systems,* O'Reilly & Associates, Sebastopol, CA, 1992, vol. 4.

[OMG, 1995] Object Management Group, *The Common Object Request Broker: Architecture and Specification.* Wiley, Nueva York, 1995.

[RMI, 1998] *Java Remote Method Invocation*, JDK Documentation, Javasoft, 1998.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Shaw y Garlan, 1996] M. Shaw y D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1996.

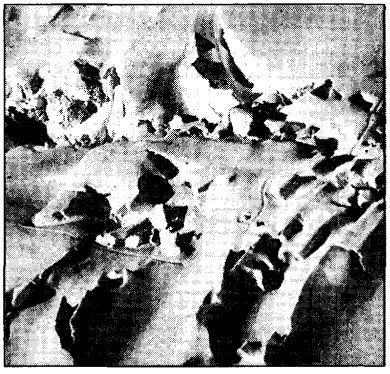
[Siewiorek y Swarz, 1992] D. P. Siewiorek y R. S. Swarz, *Reliable Computer Systems: Design and Evaluation.* 2a. ed. Digital, Burlington, MA, 1992.

[Silberschatz *et al.*, 1991] A. Silberschatz, J. Peterson y P. Galvin, *Operating System Concepts.* 3a. ed. Addison-Wesley, Reading, MA, 1991.

[Tanenbaum, 1996] A. S. Tanenbaum, *Computer Networks*, 3a. ed. Prentice Hall, Upper Saddle River, NJ, 1996.

7

7



Diseño de objetos

Si tiene un procedimiento con 10 parámetros, es probable que se le escapen algunos.
—Alan Perlis, Epigrams in Programming

Durante el análisis describimos el propósito del sistema. Esto da como resultado la identificación de los objetos de aplicación que representan a los conceptos de los usuarios. Durante el diseño del sistema describimos el sistema desde el punto de vista de su arquitectura, como su descomposición en subsistemas, su flujo de control global y la administración de la persistencia. Durante el diseño del sistema también definimos la plataforma de hardware y software sobre la que construiremos el sistema. Esto da como resultado la selección de componentes hechos que proporcionan un nivel de abstracción más elevado que el hardware. Durante el diseño de objetos cerramos el hueco entre los objetos de aplicación y los componentes hechos, identificando objetos de solución adicionales y refinando los objetos existentes.

El diseño de objetos incluye:

- *Especificación de servicios*, durante la cual describimos con precisión cada interfaz de clase.
- *Selección de componentes*, durante la cual identificamos componentes hechos y objetos de solución adicionales.
- *Reestructuración del modelo de objetos*, durante la cual transformamos el modelo de diseño de objetos para mejorar su comprensibilidad y extensibilidad.
- *Optimización del modelo de objetos*, durante la cual transformamos el modelo de diseño de objetos para tratar criterios de desempeño, como el tiempo de respuesta o la utilización de la memoria.

El diseño de objetos, al igual que el diseño del sistema, no es algorítmico. En este capítulo mostramos la manera de aplicar patrones existentes y componentes concretos en el proceso de resolución del problema. Tratamos estos bloques de construcción y sus actividades relacionadas. Concluimos tratando los asuntos administrativos asociados con el diseño de objetos. En este capítulo usamos Java y tecnologías basadas en Java. Sin embargo, las técnicas que describimos también son aplicables a otros lenguajes.

7.1 Introducción: ejemplos de películas

Considere los siguientes ejemplos:

***Speed* (1994)**

Harry, un policía de Los Ángeles, es tomado como rehén por Howard, un bombardero loco. Jack, el compañero de Harry, le dispara a Harry en la pierna para hacer más lento el avance de Howard. A Harry lo hieren en la pierna derecha. A lo largo de la película Harry cojea de la pierna izquierda.

Trilogía *de la guerra de las galaxias* (1977, 1980 y 1983)

Al final del episodio V: *El imperio contraataca* (1980), Han Solo es capturado y congelado en carbonita para enviárselo a Jaba. Al inicio del episodio VI, *El regreso del Jedi* (1983), el congelado Han Solo es recuperado por sus amigos y descongelado para que vuelva a la vida. Cuando lo congelaron Solo usaba una chaqueta. Cuando lo descongelan viste una camisa blanca.

***Titanic* (1997)**

Jack, un trotamundos, está enseñando a Rose, una dama de la alta sociedad, a escupir. Se lo demuestra poniéndole el ejemplo y motiva a Rose para que lo haga también. Durante la lección llega de improviso la madre de Rose. Cuando Jack comienza a girar para enfrentar a la madre de Rose no hay saliva en su cara. Cuando termina el giro tiene saliva en el mentón.

Los presupuestos para *Speed*, *El imperio contraataca*, *El regreso del Jedi* y *Titanic* fueron 30, 18, 32.5 y 200 millones de dólares, respectivamente.

Las películas, al igual que el software, son sistemas complejos que contienen errores (a veces muchos) cuando se entregan a los clientes. Es sorprendente, considerando su costo de producción, que cualesquiera errores obvios permanezcan en el producto final. Sin embargo, las películas son como los sistemas de software: son más complejos de lo que parecen.

Muchos factores conspiran para introducir errores en una película: las películas requieren la colaboración de muchas personas diferentes, las escenas se filman fuera de secuencia, algunas escenas se vuelven a filmar fuera de lo planeado, detalles como los accesorios y el vestuario se cambian durante la producción, la presión de la fecha de lanzamiento es alta durante el proceso de edición cuando se integran todas las partes. Cuando se filma una escena, el estado de cada uno de los objetos y actores en escena necesita ser consistente con las escenas precedentes y subsiguientes. Esto puede incluir la pose de cada actor, la condición de su vestuario, joyería, maquillaje y peinado, el contenido de los vasos, si están bebiendo (por ejemplo, vino blanco o tinto), el nivel de los vasos (por ejemplo, llenos, medio vacíos), etcétera. Cuando se combinan diferentes segmentos en una sola escena, un editor, llamado editor de continuidad, necesita asegurarse que tales detalles se hayan restaurado de manera adecuada. Cuando suceden cambios, como la adición o eliminación de un accesorio, el cambio no debe interferir con otras escenas.

Los sistemas de software, al igual que las películas, son complejos, están sujetos a cambios continuos, se integran bajo presiones de tiempo y se desarrollan en forma no lineal. Durante el diseño de objetos, los desarrolladores cierran el hueco entre los objetos de aplicación identificados durante el análisis y la plataforma de hardware y software seleccionada durante el diseño del sistema. Los desarrolladores identifican y construyen objetos de solución personalizados cuyo propósito es realizar cualquier funcionalidad que falte y cerrar el hueco entre los objetos de aplicación y la plataforma de hardware y software seleccionada. Durante el diseño de objetos, los

desarrolladores realizan objetos personalizados, en forma similar a la toma de escenas de películas. Son implementados fuera de secuencia por desarrolladores diferentes y cambian varias veces antes de que lleguen a su forma final. Con frecuencia, el que llama a una operación sólo tiene una especificación informal sobre la operación y hace suposiciones acerca de los efectos laterales y sus casos de frontera. Esto da lugar a faltas de concordancia entre el que llama y el llamado, comportamiento faltante o comportamiento incorrecto. Para resolver estos problemas, los desarrolladores construyen especificaciones precisas de las clases, atributos y operaciones en forma de restricciones. De manera similar, los desarrolladores ajustan y reutilizan componentes hechos, comentados con especificaciones de interfaz. Por último, los desarrolladores reestructuran y optimizan el modelo del diseño de objetos para tratar los objetivos de diseño, como la mantenibilidad, extensibilidad, eficiencia, tiempo de respuesta o entrega a tiempo.

En la sección 7.2, la siguiente sección, proporcionamos un panorama del diseño de objetos. En la sección 7.3 definimos los conceptos principales del diseño de objetos, como las restricciones que se usan para la especificación de interfaces. En la sección 7.4 describimos con mayor detalle las actividades del diseño de objetos. En la sección 7.5 tratamos los problemas administrativos relacionados con el diseño de objetos. No describimos actividades como la implementación de algoritmos y estructuras de datos o el uso de lenguajes de programación específicos. Primero, suponemos que el lector ya tiene experiencia en esas áreas. Segundo, esas actividades llegan a ser menos esenciales conforme se dispone cada vez de más componentes hechos y se les reutiliza.

7.2 Un panorama del diseño de objetos

Desde el punto de vista conceptual, vemos el desarrollo de sistemas como el relleno del hueco entre el problema y la máquina. Las actividades de desarrollo del sistema cierran de manera incremental ese hueco al identificar y definir objetos que realizan parte del sistema (figura 7-1).

El análisis reduce el hueco entre el problema y la máquina, identificando objetos que representan conceptos visibles para el usuario. Durante el análisis describimos el sistema desde el punto de vista del comportamiento externo, como su funcionalidad (modelo de caso de uso), los conceptos del dominio de aplicación que maneja (modelo de objetos), su comportamiento desde el punto de vista de las interacciones (modelo dinámico) y sus requerimientos no funcionales.

El diseño del sistema reduce el hueco entre el problema y la máquina definiendo una plataforma de hardware y software que proporciona un nivel de abstracción más alto que el hardware de la computadora. Esto se realiza seleccionando componentes hechos para la realización de servicios estándar, como el middleware, juegos de herramientas de interfaz de usuario, marcos de aplicación y bibliotecas de clases.

Durante el diseño de objetos refinamos los modelos de análisis y de diseño del sistema, identificamos nuevos objetos y cerramos el hueco entre los objetos de aplicación y los componentes hechos. Esto incluye la identificación de objetos personalizados, el ajuste de los componentes hechos y la especificación precisa de cada interfaz de subsistema y clase. Como resultado, el modelo de diseño de objetos puede particionarse en conjuntos de clases que pueden ser implementados por desarrolladores individuales.

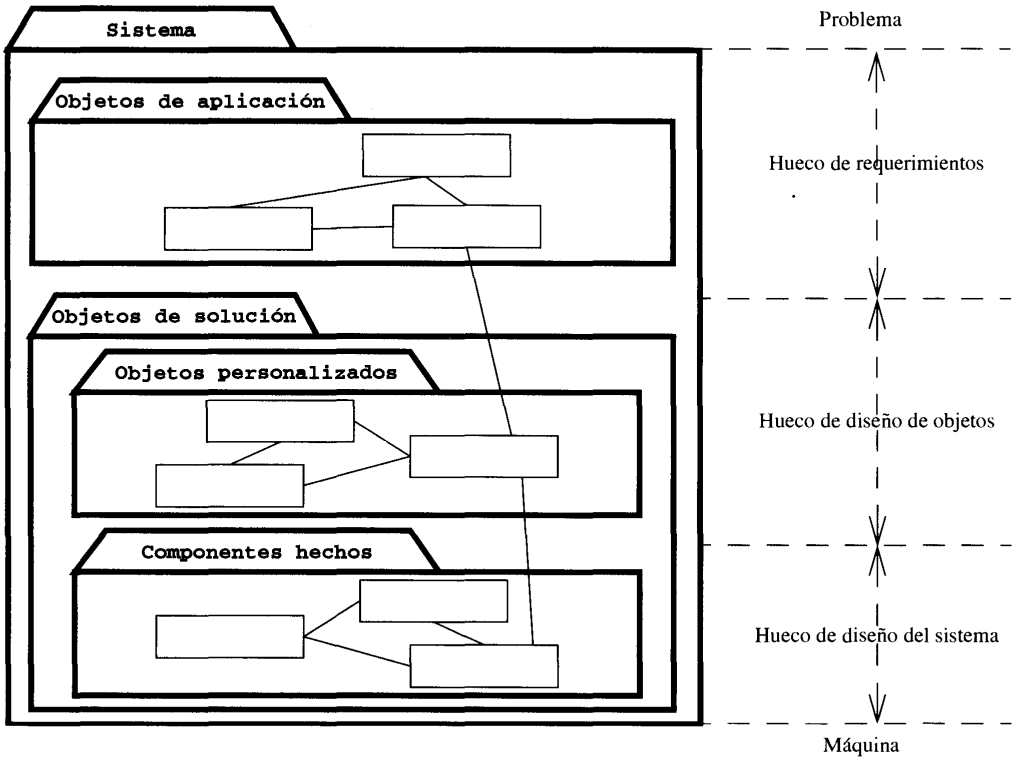


Figura 7-1 El diseño de objetos cierra el hueco entre los objetos de aplicación identificados durante los requerimientos y los componentes hechos seleccionados durante el diseño del sistema (diagrama de clase UML estilizado).

El diseño de objetos incluye cuatro grupos de actividades (vea la figura 7-2):

- *Especificación de servicios.* Durante el diseño de objetos especificamos los servicios de subsistemas (identificados durante el diseño del sistema) desde el punto de vista de interfaces de clase, incluyendo operaciones, argumentos, firmas de tipo y excepciones. Durante esta actividad también encontramos operaciones faltantes y objetos necesarios para transferir datos entre subsistemas. El resultado de la especificación de servicios es una especificación de interfaz completa para cada subsistema. A la especificación de servicios de subsistema con frecuencia se le llama **API** (siglas en inglés de interfaz de programación de aplicaciones) de subsistema.
- *Selección de componentes.* Durante el diseño de objetos usamos y adaptamos los componentes hechos identificados durante el diseño del sistema para realizar cada subsistema. Seleccionamos bibliotecas de clase y componentes adicionales para las estructuras de datos y servicios básicos. A menudo necesitamos ajustar los componentes que seleccionamos antes de poder usarlos envolviendo objetos personalizados alrededor de ellos o refinándolos usando herencia. Durante esas actividades enfrentamos el mismo compromiso de compra frente a construcción que encaramos durante el diseño del sistema.

- *Reestructuración.* Las actividades de reestructuración manipulan el modelo del sistema para incrementar la reutilización del código o satisfacer otros objetivos de diseño. Cada actividad de reestructuración puede verse como una transformación gráfica de subconjuntos de un modelo particular. Las actividades típicas incluyen la transformación de asociaciones n-arias en asociaciones binarias, la implementación de asociaciones binarias como referencias, la combinación de dos clases similares de dos subsistemas diferentes en una sola clase, la descomposición en atributos de clases que no tienen comportamiento significativo, la división de clases complejas en otras más simples, el reacomodo de clases y operaciones para incrementar la herencia y el empacado. Durante la reestructuración tratamos objetivos de diseño, como el mantenimiento, la legibilidad y la comprensibilidad del modelo del sistema.
- *Optimización.* Las actividades de optimización tratan los requerimientos de desempeño del modelo del sistema. Esto incluye el cambio de algoritmos para que respondan a los requerimientos de velocidad o memoria, la reducción de multiplicidades en asociaciones para agilizar las consultas, la adición de asociaciones redundantes para eficiencia, el reacomodo del orden de ejecución, la adición de atributos derivados para mejorar el tiempo de acceso a objetos y la apertura de la arquitectura; esto es, la adición de acceso a capas inferiores debido a requerimientos de desempeño.

El diseño de objetos no es lineal. Aunque cada una de las actividades que describimos antes aborda un problema específico de diseño de objetos, necesitan suceder en forma concurrente. Un componente hecho específico puede restringir la cantidad de tipos de excepciones mencionadas en la especificación de una operación y, por tanto, puede tener impacto en la interfaz del subsistema. La selección de un componente puede reducir el trabajo de implementación mientras introduce nuevos objetos "de unión", los cuales también tienen que especificarse. Por último, la reestructuración y la optimización pueden reducir la cantidad de objetos a implementar, incrementando la cantidad de reutilización en el sistema.

La mayor cantidad de objetos y desarrolladores, la alta tasa de cambio y la cantidad de decisiones concurrentes que se toman durante el diseño de objetos hacen que esta actividad sea mucho más compleja que el análisis o el diseño del sistema. Esto representa un reto de administración, ya que muchas decisiones importantes tienden a resolverse en forma independiente y no se comunican al resto del proyecto. El diseño de objetos requiere que mucha información se ponga a disposición de los desarrolladores para que puedan tomarse las decisiones en forma consistente con las decisiones tomadas por los demás desarrolladores y con los objetivos de diseño. El *Documento de Diseño de Objetos*, un documento vivo que describe la especificación de cada clase, apoya este intercambio de información.

7.3 Conceptos del diseño de objetos

En esta sección presentamos los principales conceptos del diseño de objetos:

- Objetos de aplicación frente a objetos de solución (sección 7.3.1)
- Tipos, firmas y visibilidad (sección 7.3.2)
- Precondiciones, poscondiciones e invariantes (sección 7.3.3)
- El lenguaje de restricción de objetos de UML (sección 7.3.4)

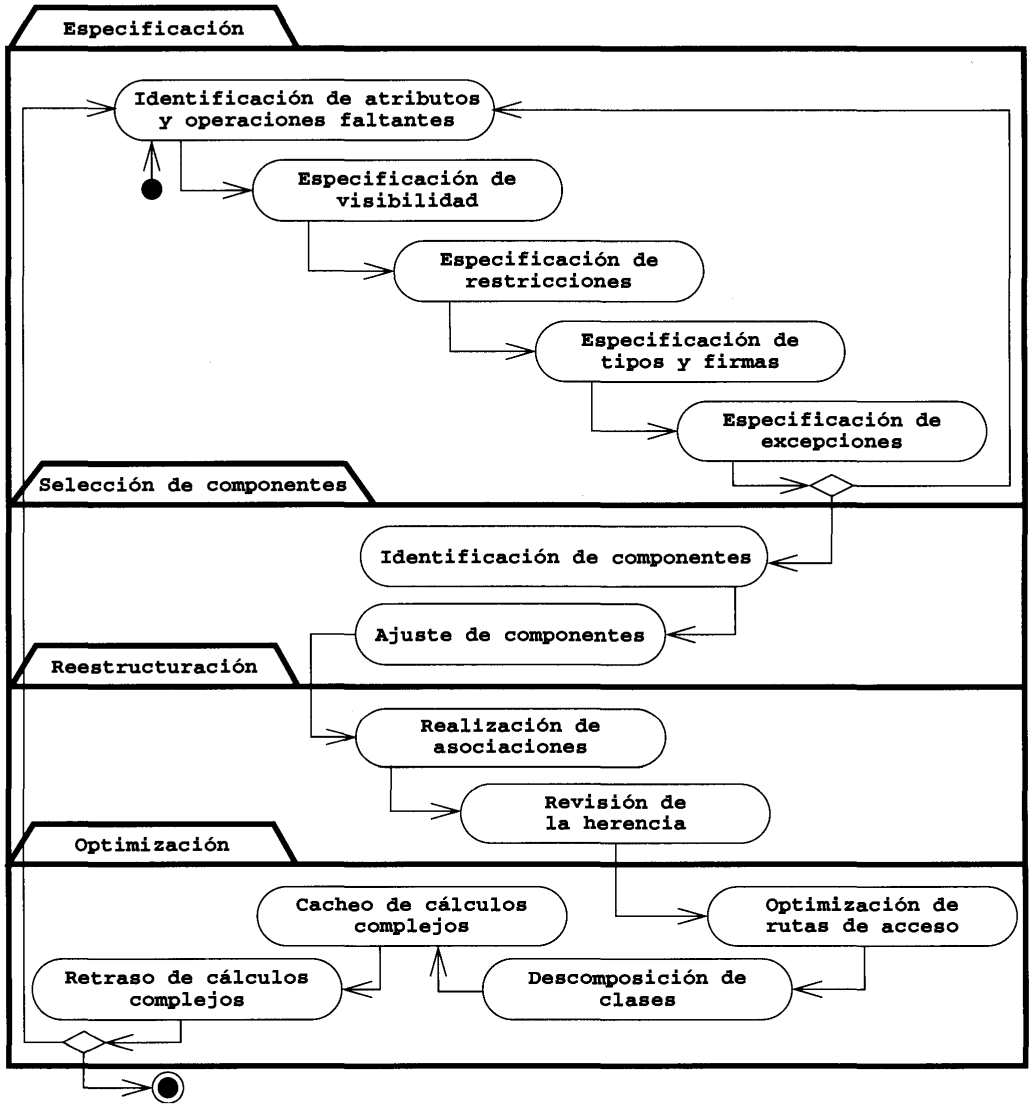


Figura 7-2 Actividades del diseño de objetos (diagrama de actividad UML).

7.3.1 Revisión de los objetos de aplicación frente a los objetos de solución

Como vimos en el capítulo 2, *Modelado con UML*, pueden usarse los diagramas de clase para modelar el dominio de aplicación y el dominio de solución. Los **objetos de aplicación**, también llamados objetos de dominio, representan conceptos del dominio que manipula el sistema. Los **objetos de solución** representan componentes de apoyo que no tienen una contraparte en el dominio de aplicación, como los almacenes de datos persistentes, los objetos de interfaz de usuario o el middleware.

Durante el análisis identificamos los objetos de aplicación, sus relaciones y atributos, y sus operaciones. Durante el diseño del sistema identificamos subsistemas y los objetos de solución más importantes. Durante el diseño de objetos refinamos y detallamos ambos conjuntos de objetos, e identificamos cualesquier objetos de solución faltantes necesarios para completar el sistema.

7.3.2 Revisión de tipos, firmas y visibilidad

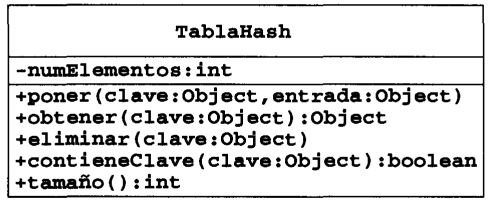
Durante el análisis identificamos atributos y operaciones sin especificar su tipo o parámetros. Durante el diseño de objetos refinamos los modelos de análisis y diseño del sistema añadiendo información de tipo y visibilidad. El **tipo** de un atributo especifica el rango de valores que puede tomar el atributo y las operaciones que pueden aplicarse al atributo. Por ejemplo, considere el atributo `numElementos` de la clase `TablaHash` (vea la figura 7-3). `numElementos` representa el número actual de entradas en una `TablaHash` dada. Su tipo es `int`, indicando que es un número entero. El tipo del atributo `numElementos` también indica las operaciones que pueden aplicarse a este atributo: podemos sumar o restar otros enteros a `numElementos`.

A los parámetros de operación y los valores de retorno se les asigna tipo de la misma forma que a los atributos. El tipo restringe el rango de valores que puede tomar el parámetro o el valor de retorno. Para toda operación, al tuplo compuesto por los tipos de sus parámetros y el tipo de valor de retorno se le llama **firma** de la operación. Por ejemplo, la operación `poner()` de `TablaHash` toma dos parámetros de tipo `Object` y no tiene valor de retorno. El tipo de firma para `poner()` es entonces `(Object, Object):void`. En forma similar, la operación `obtener()` de `TablaHash` toma un parámetro `Object` y regresa un `Object`. El tipo de firma de `obtener()` es entonces `(Object):Object`.

La **visibilidad** de un atributo o una operación especifica si puede ser usada por otras clases o no. UML define tres niveles de visibilidad:

- **Privado.** Sólo puede tener acceso a un atributo privado la clase en la que está definido. En forma similar, una operación privada sólo puede ser llamada por la clase en la que está definida. Las subclases u otras clases no pueden tener acceso a los atributos y operaciones privados.
- **Protegido.** La clase en la que están definidos y cualquier descendiente de la clase pueden tener acceso a un atributo u operación protegidos.
- **Público.** Cualquier clase puede tener acceso a un atributo u operación público.

La visibilidad se indica en UML poniendo como prefijo el símbolo: – (privado), # (protegido) o + (público) al nombre del atributo u operación. Por ejemplo, en la figura 7-3 especificamos que el atributo `numElementos` de `TablaHash` es privado, mientras que todas las operaciones son públicas.




```
class TablaHash {
    private int numElementos;

    /* Se omiten los Constructores */
    public void poner (Object clave, Object entrada) {…};
    public Object obtener(Object clave) {…};
    public void eliminar(Object clave) {…};
    public boolean contieneClave(Object clave) {…};
    public int tamaño() {…};

    /* Se omiten otros métodos */
}
```

Figura 7-3 Declaraciones para la clase `TablaHash` (modelo de clase UML y fragmentos Java).

Con frecuencia, la información del tipo por sí sola no es suficiente para especificar el rango de valores legítimos. En el ejemplo `TablaHash` el tipo `int` permite que `numElementos` tome valores negativos, los cuales no son válidos para este atributo. A continuación tratamos esta cuestión con contratos.

7.3.3 Contratos: invariantes, precondiciones y poscondiciones

Los **contratos** son restricciones sobre una clase que permiten que el llamador y el llamado compartan las mismas suposiciones acerca de la clase [Meyer, 1997]. Un contrato especifica restricciones que debe satisfacer el llamador antes de usar la clase, así como las restricciones que asegura cumplir el llamado cuando se le usa. Los contratos incluyen tres tipos de restricciones:

- Un **invariante** es un predicado que siempre es cierto para todas las instancias de una clase. Los invariantes son restricciones asociadas con clases o interfaces. Los invariantes se usan para especificar restricciones de consistencia entre atributos de clase.
- Una **precondición** es un predicado que debe ser cierto antes de que se llame a una operación. Las precondiciones están asociadas con una operación específica. Las precondiciones se usan para especificar restricciones que debe satisfacer el llamador antes de llamar a una operación.
- Una **poscondición** es un predicado que debe ser cierto después de que se llama a una operación. Las poscondiciones están asociadas con una operación específica. Las poscondiciones se usan para especificar restricciones que el objeto debe asegurar después de la invocación de la operación.

Por ejemplo, considere la interfaz Java para una TablaHash que se muestra en la figura 7-3. Esta clase proporciona un método poner() para crear una entrada en la tabla asociando una clave con un valor, un método obtener() para buscar un valor dando una clave, un método eliminar() para destruir una entrada de la TablaHash y un método claveHash() que regresa un valor booleano que indica si existe o no una entrada.

Un ejemplo de un invariante para la clase TablaHash es que la cantidad de entradas en TablaHash siempre es no negativa. Por ejemplo, si el método eliminar() da como resultado un valor negativo de numElementos, la implementación de TablaHash es incorrecta. Un ejemplo de una precondición para el método eliminar() es que una entrada debe estar asociada con la clave que se pasa como parámetro. Un ejemplo de una poscondición para el método eliminar() es que la entrada eliminada ya no debe existir en la TablaHash después de que regrese el método eliminar(). La figura 7-4 muestra la clase TablaHash comentada con invariantes, precondiciones y poscondiciones.

```
/* Clase TablaHash. Mantiene la correspondencia entre claves únicas y objetos
arbitrarios */
class Hashtable {

    /* La cantidad de elementos de TablaHash es no negativa todo el tiempo.
    * @inv numElementos >= 0 */
    private int numElementos;

    /* Se omiten los constructores */

    /* La operación poner asume que la clave especificada no se ha usado.
     * Después de que termina la operación poner se puede usar la clave
       especificada
     * para recuperar la entrada con la operación obtener(clave):
     * @pre !contieneClave(clave)
     * @post contieneClave(clave)
     * @post obtener(clave) == entrada */
    public void poner (Object clave, Object entrada) {…};


    /* La operación obtener asume que la clave especificada corresponde
     * a una entrada de la TablaHash.
     * @pre contieneClave(clave) */
    public Object obtener(Object clave) {…};

    /* La operación eliminar asume que la clave especificada existe
     * en la TablaHash.
     * @pre contieneClave(clave)
     * @post !contieneClave(clave) */
    public void eliminar(Object clave) {…};

    /* Se omiten otros métodos */
}
```

Figura 7-4 Declaración de métodos para la clase TablaHash comentada con precondiciones, poscondiciones e invariantes (Java, restricciones en la sintaxis iContract [iContract]).

Usamos invariantes, precondiciones y poscondiciones para especificar sin ambigüedades casos especiales o excepcionales. Por ejemplo, las restricciones en la figura 7-4 indican que el método `eliminar()` debe ser llamado sólo para entradas que existen en la tabla. En forma similar, el método `poner()` debe ser llamado sólo si no se está usando la clave. En la mayoría de los casos, también es posible usar restricciones para especificar por completo el comportamiento de una operación. Sin embargo, tal uso de restricciones, llamado *especificación basada en restricciones*, es difícil y puede ser más complicado que la implementación de la operación misma [Horn, 1992]. En este capítulo no describiremos especificaciones basadas en restricciones puras. En vez de ello nos enfocaremos en la especificación de operaciones usando restricciones y el lenguaje natural, enfatizando los casos de frontera.

7.3.4 Lenguaje de restricción de objetos de UML

En UML las restricciones se expresan usando OCL [OMG, 1998]. OCL es un lenguaje que permite que se especifiquen de manera formal las restricciones en elementos únicos del modelo (por ejemplo, atributos, operaciones, clases) o en grupos de elementos del modelo (por ejemplo, asociaciones y clases participantes). Una restricción se expresa como una restricción OCL que regresa el valor cierto o falso. OCL no es un lenguaje procedural y, por tanto, no puede usarse para indicar el flujo de control. En este capítulo nos enfocamos exclusivamente en los aspectos de OCL relacionados con invariantes, precondiciones y poscondiciones.

Una restricción puede mostrarse como una nota asociada al elemento UML restringido mediante una relación de dependencia. Una restricción puede expresarse en lenguaje natural o en un lenguaje formal, como OCL. La figura 7-5 muestra un diagrama de clase del ejemplo `TablaHash` de la figura 7-4 usando UML y OCL.

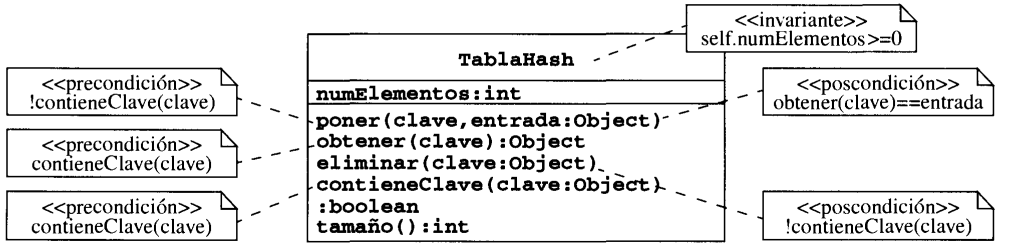


Figura 7-5 Ejemplos de invariantes, precondiciones y poscondiciones en OCL (diagrama de clase UML).

La sintaxis del OCL es similar a la de los lenguajes orientados a objetos como C++ o Java. En el caso de un invariante, el contexto para la expresión es la clase asociada con el invariante. La palabra clave `self` (por ejemplo, `self.numElementos` en la figura 7-5) indica cualquier instancia de la clase. Se tiene acceso a los atributos y operaciones usando la notación de punto (por ejemplo, `self.atributo` o `self.operación(parámetros)`). La palabra clave `self` puede omitirse si no se presenta ambigüedad. (Observe que OCL usa la palabra clave `self` para representar el mismo concepto que la palabra clave `this` en Java y C++.) Para una precondición o una poscondición, los parámetros que se pasan a la operación asociada pueden usarse en la expresión

OCL. Para las poscondiciones el sufijo `@pre` indica el valor de un parámetro o un atributo antes de la ejecución de la operación. Por ejemplo, una poscondición de la operación `poner(clave, entrada)`, que expresa que la cantidad de entradas en la tabla se incrementa en uno, puede representarse como `numElementos = numElementos@pre + 1`.

Al añadir expresiones OCL a los diagramas puede presentarse un amontonamiento. Por esta razón existe la alternativa de manifestar en forma textual las expresiones OCL. La palabra clave context introduce un nuevo contexto para una expresión OCL. La palabra que está a continuación de la palabra clave context se refiere a una clase, un atributo o una operación. Luego sigue alguna de las palabras clave `inv`, `pre` y `post`, que corresponden a los estereotipos UML `<<invariante>>`, `<<precondición>>` y `<<poscondición>>`, respectivamente. Luego sigue la expresión OCL en sí. Por ejemplo, el invariante para la clase `TablaHash` y las restricciones para la operación `TablaHash.poner()` se escriben de la siguiente manera:

```
context Hashtable inv:
numElementos >= 0

context TablaHash::poner(clave, entrada) pre:
!contieneClave(clave)

context TablaHash::poner(clave, entrada) post:
contieneClave(clave) and obtener(clave) = entrada
```

7.4 Actividades del diseño de objetos

Como ya se mencionó en la introducción, el diseño de objetos incluye cuatro grupos de actividades: especificación, selección de componentes, reestructuración y optimización.

Las actividades de especificación incluyen:

- Identificación de atributos y operaciones faltantes (Sección 7.4.1)
- Especificación de tipos de firma y visibilidad (sección 7.4.2)
- Especificación de restricciones (sección 7.4.3)
- Especificación de excepciones (sección 7.4.4)

Las actividades de selección de componentes incluyen:

- Identificación y ajuste de bibliotecas de clase (sección 7.4.5)
- Identificación y ajuste de marcos de aplicación (sección 7.4.6)
- Un ejemplo de marco: WebObjects (sección 7.4.7)

Las actividades de reestructuración incluyen:

- Realización de asociaciones (sección 7.4.8)
- Incremento de la reutilización (sección 7.4.9)
- Eliminación de dependencias de implementación (sección 7.4.10)

Las actividades de optimización incluyen:

- Revisión de las rutas de acceso (sección 7.4.11)
- Descomposición de objetos: conversión de objetos en atributos (sección 7.4.12)
- Cacheo del resultado de cálculos costosos (sección 7.4.13)
- Retraso de cálculos costosos (sección 7.4.14)

Para ilustrar con mayor detalle estos cuatro grupos de actividades usamos como ejemplo un sistema de modelado de emisiones llamado JEWEL (taller ambiental conjunto y laboratorio de emisiones [Bruegge *et al.*, 1995], [Kompanek *et al.*, 1996]). JEWEL permite que los usuarios finales simulen y visualicen la contaminación del aire como función de fuentes puntuales (por ejemplo, fábricas, plantas eléctricas), fuentes de área (por ejemplo, ciudades) y fuentes móviles (por ejemplo, automóviles, camiones, trenes). El área bajo estudio está dividida en celdas de cuadrícula. Luego se estiman las condiciones para cada celda de cuadrícula y para cada hora del día. Una vez que termina la simulación, el usuario final puede visualizar la concentración de varios contaminantes en un mapa, junto con las fuentes de emisión (vea la figura 7-6). JEWEL está destinado a agencias gubernamentales que regulan la calidad del aire y tratan de que las áreas pobladas que tienen problemas se apeguen a los reglamentos.

Tomando en cuenta su enfoque sobre la visualización de datos geográficos, JEWEL incluye un subsistema de información geográfica (GIS, por sus siglas en inglés) que es responsable del almacenamiento y la manipulación de mapas. El `GIS` de JEWEL administra la información geográfica como conjuntos de polígonos y segmentos. Diferentes tipos de información, como caminos, ríos y fronteras políticas, están organizados en capas diferentes que pueden mostrarse en forma independiente. Además, los datos están organizados de tal forma que pueden verse en diferentes niveles de abstracción. Por ejemplo, una vista de alto nivel de un mapa sólo contiene los caminos principales, mientras que una vista detallada también incluye los secundarios. El `GIS` es un ejemplo ideal para el diseño de objetos, tomando en cuenta su rico dominio de aplicación y su complejo dominio de solución. Primero comenzamos con la especificación del `GIS`.

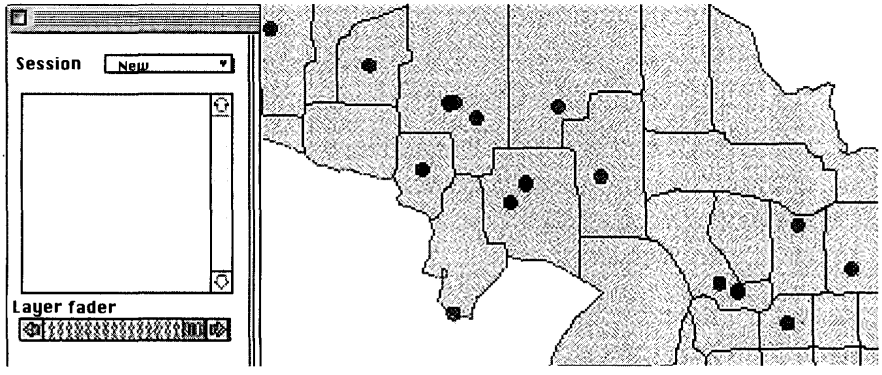


Figura 7-6 Mapa con fronteras políticas y fuentes de emisión (maqueta de JEWEL).

Actividades de especificación

En la figura 7-7 el modelo de objetos para el GIS de JEWEL describe una organización en tres capas (es decir, la capa de caminos, la capa de agua y la capa política). Cada capa está compuesta de elementos. Algunos de estos elementos, como las autopistas, caminos secundarios y ríos, se despliegan con líneas compuestas por varios segmentos. Otros, como los lagos, estados y condados, se despliegan como polígonos, que también son representados como una colección de segmentos de líneas.

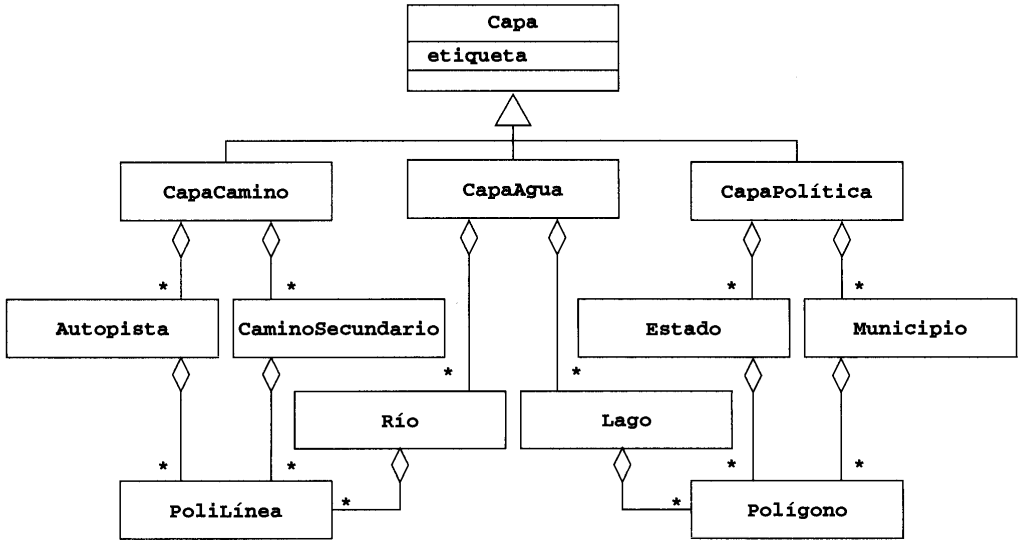


Figura 7-7 Modelo de objetos para el GIS de JEWEL (diagrama de clase UML).

El caso de uso AcercamientoMapa (figura 7-8) describe la manera en que los usuarios pueden acercarse o alejarse alrededor de un punto seleccionado del mapa. El modelo de análisis todavía es abstracto. Por ejemplo, en este momento no contiene ninguna información sobre la manera en que se incrementa el acercamiento o cómo se seleccionan los puntos.

El modelo de diseño del sistema se enfoca en la descomposición en subsistemas y las decisiones de sistema globales, como la correspondencia entre hardware y software, el almacenamiento persistente o el control de acceso. Identificamos los subsistemas de nivel superior y los definimos desde el punto de vista de los servicios que proporciona. En JEWEL, por ejemplo (figuras 7-9 y 7-10), identificamos al GIS proporcionando servicios para la creación, almacenamiento y borrado de elementos geográficos, su organización en capas y la recuperación de su contorno como una serie de puntos. Estos servicios los usa el subsistema Visualización, el cual recupera información geográfica para el trazado de mapas. Los datos geográficos se proporcionan como un conjunto de archivos planos y son tratados por JEWEL como datos estáticos. En consecuencia, no necesitamos soportar la edición interactiva de datos geográficos. A partir de los casos de uso de JEWEL también sabemos que los usuarios necesitan ver los datos geográficos con diferentes criterios de acercamiento. Durante el diseño del sistema decidimos que el GIS proporcione los servicios de acercamiento y recorte. El

Nombre del caso de uso	`AcercamientoMapa`
Condición inicial	El mapa se despliega en una ventana y es visible al menos una capa.
Flujo de eventos	1. El usuario final selecciona la herramienta Acercamiento desde la barra de herramientas. El sistema cambia el cursor a una lupa. 2. El usuario final selecciona un punto del mapa usando el ratón y haciendo clic con el botón izquierdo o derecho del ratón. El punto seleccionado por el usuario se convertirá en el nuevo centro del mapa. 3. El usuario final hace clic con el botón izquierdo del ratón para solicitar un incremento en el nivel de detalle (es decir, acercamiento) o con el botón derecho del ratón para solicitar un decremento del nivel de detalle (es decir, alejamiento). 4. El sistema calcula el nuevo cuadro limitante y recupera del `GIS` los puntos y líneas correspondientes para cada capa visible. 5. Luego el sistema despliega cada capa usando un color predefinido en el nuevo cuadro limitante.
Condición final	El mapa se desplaza y escala a la posición y nivel de detalle solicitados.

Figura 7-8 Caso de uso `AcercamientoMapa` de JEWEL.

subsistema `Visualización` especifica el nivel de detalle y el cuadro limitante del mapa, y el `GIS` realiza el acercamiento y recorte y regresa sólo los puntos que necesitan trazarse. Esto minimiza la cantidad de datos que hay que transferir entre subsistemas. Aunque el modelo de diseño del sistema está cercano a la máquina, todavía tenemos que describir con detalle la interfaz del `GIS`.

Las actividades de especificación durante el diseño de objetos incluyen:

- Identificación de atributos y operaciones faltantes (sección 7.4.1)
- Especificación del tipo de firmas y visibilidad (sección 7.4.2)
- Especificación de restricciones (sección 7.4.3)
- Especificación de excepciones (sección 7.4.4)

GIS de JEWEL

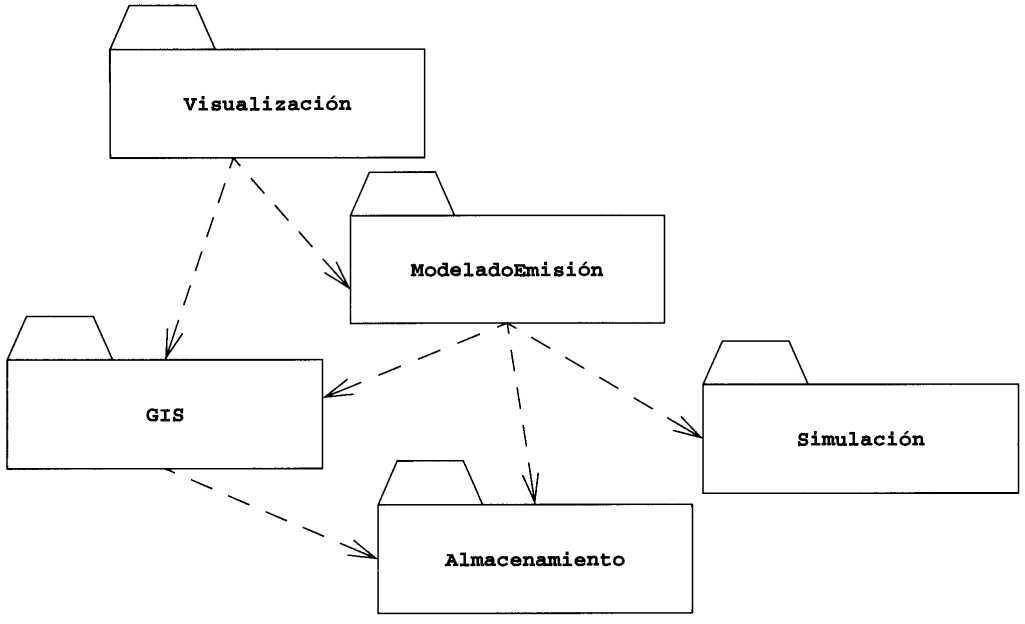
Propósito

- Almacenar y mantener la información geográfica para JEWEL

Servicio

- Creación y borrado de elementos geográficos (caminos, ríos, lagos y fronteras)
- Organización de los elementos en capas
- Acercamiento (selección de puntos en un nivel de detalle dado)
- Recorte (selección de puntos dentro de un cuadro limitante dado)

Figura 7-9 Descripción de subsistemas para el `GIS` de JEWEL.



ModeladoEmisión	El subsistema `ModeladoEmisión` es responsable del ajuste de simulaciones y la administración de sus resultados.
GIS	El `GIS` mantiene información geográfica para `Visualización` y para el `ModeladoEmisión`.
Simulación	El subsistema `Simulación` es responsable de la simulación de emisiones.
Almacenamiento	El subsistema `Almacenamiento` es responsable de todos los datos persistentes en el sistema, incluyendo los datos geográficos y de emisión.
Visualización	El subsistema `Visualización` es responsable del desplegado de datos geográficos y de emisiones ante el usuario.

Figura 7-10 Descomposición de subsistemas de JEWEL (diagrama de clase UML).

7.4.1 Identificación de atributos y operaciones faltantes

Durante este paso examinamos la descripción de servicios del subsistema e identificamos atributos y operaciones faltantes. Durante el análisis podemos haber olvidado muchos atributos debido a que nos enfocamos en la funcionalidad del sistema. Además describimos la funcionalidad del sistema principalmente con el modelo de caso de uso (y no con el modelo de objetos). Cuando construimos el modelo de objetos nos enfocamos en el modelo de aplicación y, por tanto, ignoramos detalles relacionados con el sistema que son independientes del dominio de aplicación.

En el ejemplo JEWEL, la creación, borrado y organización de capas y de elementos de capas ya está soportado por la clase `Capa`. Sin embargo, necesitamos identificar las operaciones

para realizar los servicios de recorte y acercamiento. El recorte no es un concepto que esté relacionado con el dominio de aplicación sino que está relacionado con la interfaz de usuario del sistema y, por tanto, es parte del dominio de solución.

Trazamos un diagrama de secuencia que representa el flujo de control y datos necesario para realizar la operación de acercamiento (figura 7-11). Nos enfocamos en especial en la clase `Capa`. Cuando trazamos el diagrama de secuencia nos damos cuenta que una `Capa` necesita tener acceso a todos los elementos contenidos para recopilar su geometría para el recorte y acercamiento. Observamos que el recorte puede realizarse en forma independiente del tipo de elemento que se está desplegando; esto es, el recorte de segmentos de línea que están asociados con un camino o un río puede hacerse usando la misma operación. En consecuencia, identificamos una nueva clase, la clase abstracta `ElementoCapa` (vea la figura 7-12), la cual proporciona operaciones para todos los elementos que son parte de una `Capa` (es decir, `Autopista`, `CaminoSecundario`, `Río`, `Lago`, `Estado` y `Municipio`).

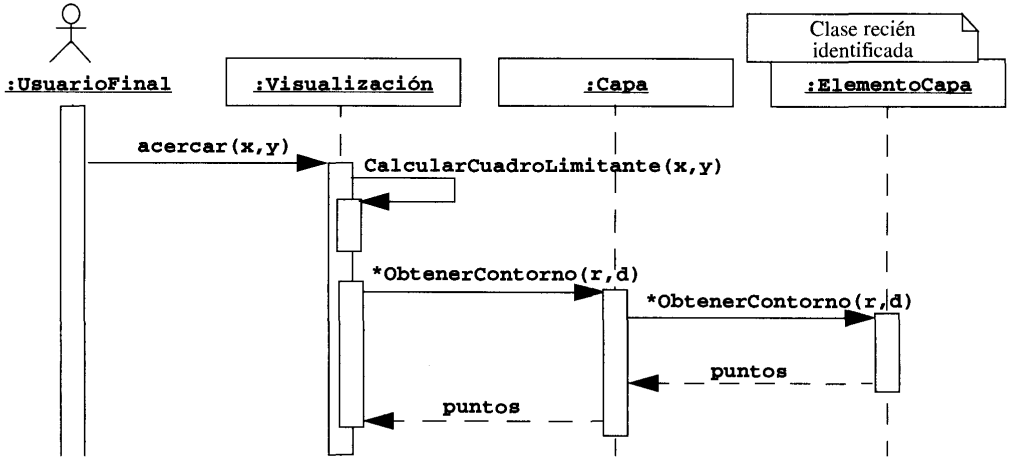


Figura 7-11 Un diagrama de secuencia para la operación `acercar()` (diagrama de secuencia UML). Este diagrama de secuencia conduce a la identificación de una nueva clase, `ElementoCapa`. Debido a que nos estamos enfocando en el `GIS`, tratamos al subsistema `Visualización` como un solo objeto.

Identificamos la operación `obtenerContorno()` de la clase `ElementoCapa`, la cual es responsable del escalamiento y recorte de líneas y polígonos de elementos individuales de acuerdo con un cuadro limitante y un nivel de detalle dados. La operación `obtenerContorno()` usa el nivel de detalle para escalar cada línea y polígono y para reducir la cantidad de puntos para niveles de detalle más bajos. Por ejemplo, cuando el usuario reduce el mapa por un factor de 2, el `GIS` regresa sólo la mitad de la cantidad de puntos para un elemento de capa dado, debido a que se necesita menos detalle. Luego identificamos la operación `obtenerContorno()` de la clase `Capa`, la cual es responsable del llamado de la operación `obtenerContorno()` de cada `ElementoCapa` y de recopilar todas las líneas y polígonos de la capa en una sola estructura de datos. Ambas operaciones `obtenerContorno()` regresan colecciones de líneas y polígonos. El subsistema

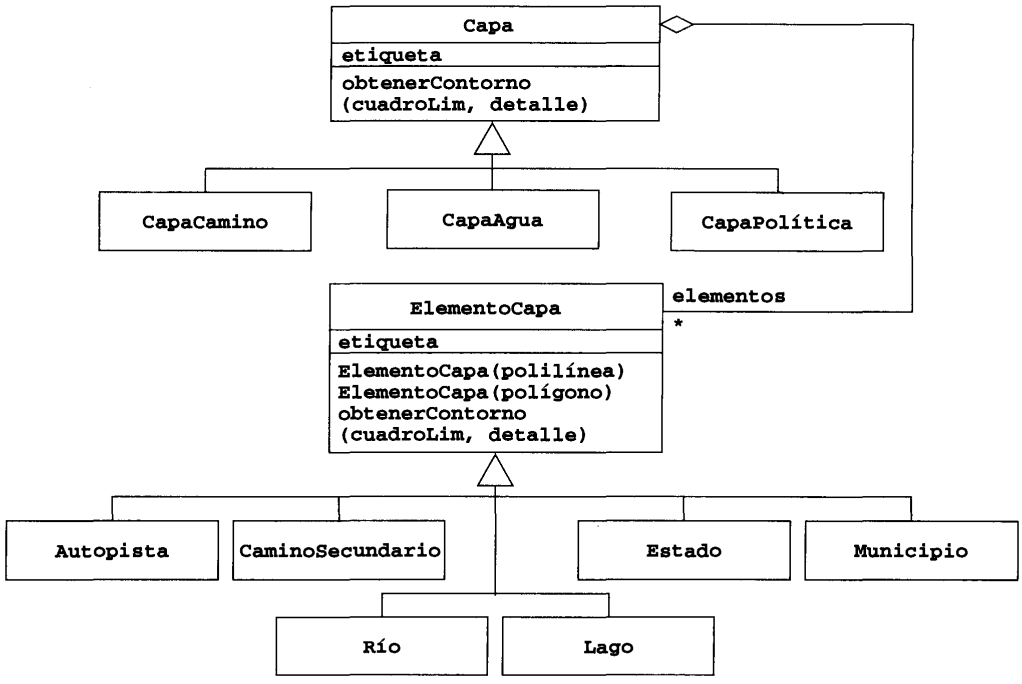


Figura 7-12 Adición de operaciones al modelo de objetos de `GIS` de JEWEL para realizar acercamiento y recorte (diagrama de clase UML).

`Visualización` traduce luego las coordenadas de `GIS` a coordenadas de pantalla, ajustando la escala y el desplazamiento, y traza los segmentos de línea en la pantalla.

Durante una revisión del modelo de objetos nos damos cuenta que el algoritmo de acercamiento de la operación `ElementoCapa.obtenerContorno()` no es trivial: no es suficiente seleccionar un subconjunto de puntos del `ElementoCapa` y escalar sus coordenadas. Debido a que diferentes `ElementoCapa` pueden compartir puntos (por ejemplo, dos caminos que se conectan, dos municipios vecinos) es necesario seleccionar el mismo conjunto de puntos para mantener una imagen visual consistente para el usuario final.

Por ejemplo, la figura 7-13 muestra ejemplos de un algoritmo sencillo para la selección de puntos aplicado a caminos que se conectan y municipios vecinos. La columna izquierda muestra los puntos que se seleccionan para un nivel de detalle alto. La columna derecha muestra los puntos que se seleccionan para un nivel de detalle bajo. En este caso, el algoritmo selecciona en forma arbitraria puntos alternados sin tomar en cuenta si los puntos se comparten o no. Esto da lugar a elementos que no están conectados cuando se despliegan en niveles de detalle bajos.

Para resolver este problema decidimos incluir más inteligencia en las clases `PoliLínea`, `Polígono` y `Punto` (figura 7-14). Primero, decidimos representar los puntos compartidos exactamente por un objeto `Punto`; esto es, si dos líneas comparten un punto, ambos objetos `Línea` tienen una referencia al mismo objeto `Punto`. Esto es manejado por el constructor `Punto(x,y)`,

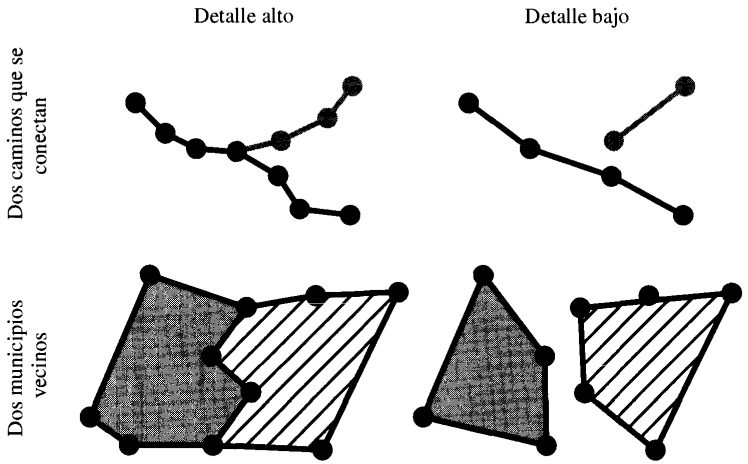


Figura 7-13 Un algoritmo de selección de puntos sencillo para el `GIS`. La columna izquierda representa un cruce de caminos y dos municipios vecinos. La columna derecha muestra que el cruce de caminos y los municipios vecinos pueden mostrarse en forma incorrecta cuando no se tiene cuidado en la selección de puntos.

el cual revisa para ver si las coordenadas especificadas corresponden a un punto existente. Segundo, añadimos atributos a los objetos `Punto` para guardar los niveles de detalle en los que participa. El atributo `enNivelesDetalle` es un conjunto de todos los niveles de detalle en los cuales participa este `Punto`. El atributo `noEnNivelesDetalle` es un conjunto de todos los niveles de detalle en los que este punto ha sido excluido. Si un nivel de detalle no está en ninguno de estos conjuntos, significa que este `Punto` todavía no ha sido considerado para el nivel de detalle dado. Los atributos `enNivelesDetalle` y `noEnNivelesDetalle` (y sus operaciones asociadas) son usados luego por la operación `ElementoCapa.obtenerContorno()` para seleccionar los puntos compartidos y mantener la conectividad.

Hasta este momento hemos identificado los atributos y operaciones faltantes necesarios para soportar el acercamiento y recorte de `ElementoCapa`. Volveremos a revisar algunos de estos problemas más adelante cuando seleccionemos componentes existentes o realicemos el diseño de objetos de los subsistemas dependientes. A continuación, procedemos a especificar la interfaz de cada una de las clases usando tipos, firmas, contratos y visibilidad.

7.4.2 Especificación de tipo, firmas y visibilidad

Durante este paso especificamos los tipos de los atributos, las firmas de las operaciones y la visibilidad de atributos y operaciones. La especificación de los tipos refina el modelo de diseño de objetos de dos maneras. Primero, añadimos detalle al modelo especificando el rango de cada atributo. Por ejemplo, mediante la determinación del tipo de coordenadas tomamos decisiones acerca de la posición del punto de origen (`Punto` 0, 0) y los valores máximo y mínimo para todas las coordenadas. Mediante la selección de un factor de punto flotante y doble pre-

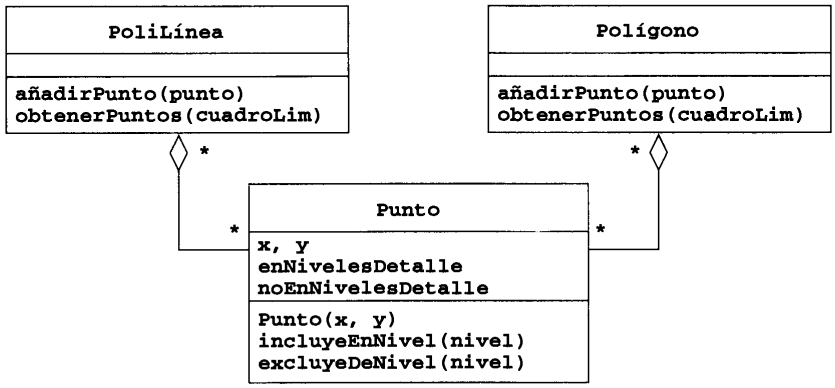


Figura 7-14 Atributos y métodos adicionales para la clase `Punto` a fin de soportar la selección inteligente de puntos y el acercamiento (diagrama de clase UML).

cisión para los niveles detallados, calculamos coordenadas en diferentes niveles de detalle tan sólo multiplicando el nivel de detalle por las coordenadas. Segundo, establecemos la correspondencia entre clases y atributos del modelo de objetos para los tipos integrados proporcionados por el ambiente de desarrollo. Por ejemplo, mediante la selección de `String` para representar el atributo `etiqueta` de `Capa` y `ElementoCapa` podemos usar todas las operaciones proporcionadas por la clase `String` para manipular valores de `etiqueta`.

Durante este paso también consideramos la relación entre las clases que identificamos y las clases de los componentes hechos. Por ejemplo, varias clases que implementan colecciones se proporcionan en el paquete `java.util`. La interfaz `Enumeration` proporciona una forma para tener acceso a una colección ordenada de objetos. La interfaz `Set` proporciona una abstracción de conjuntos matemáticos implementada por varias clases, como `HashSet`. Seleccionamos la interfaz `Enumeration` para el regreso de contornos y la interfaz `Set` del paquete `java.util` para la representación de los atributos `enNivelesDetalle` y `noEnNivelesDetalle`.

Por último, durante este paso determinamos la visibilidad de cada atributo y operación. Al hacerlo determinamos cuáles atributos son manejados por completo por una clase, cuáles deben ser accesibles sólo mediante las operaciones de la clase y cuáles atributos son públicos y pueden ser modificados por cualquier otra clase. En forma similar, la visibilidad de las operaciones nos permite distinguir entre las operaciones que son parte de la interfaz de la clase y aquellas que son métodos de utilería a los que sólo puede tener acceso la clase. En el caso de clases abstractas y clases que se pretende que se refinen, también definimos atributos y métodos protegidos para uso único de las subclases. La figura 7-15 muestra la especificación refinada de las clases `Capa`, `ElementoCapa`, `PoliLínea` y `Punto` después de que se han asignado los tipos, firmas y visibilidad.

Una vez que hemos especificado el tipo de cada atributo, la firma de cada operación y su visibilidad, nos enfocamos en la especificación del comportamiento y los casos de frontera de cada clase usando contratos.

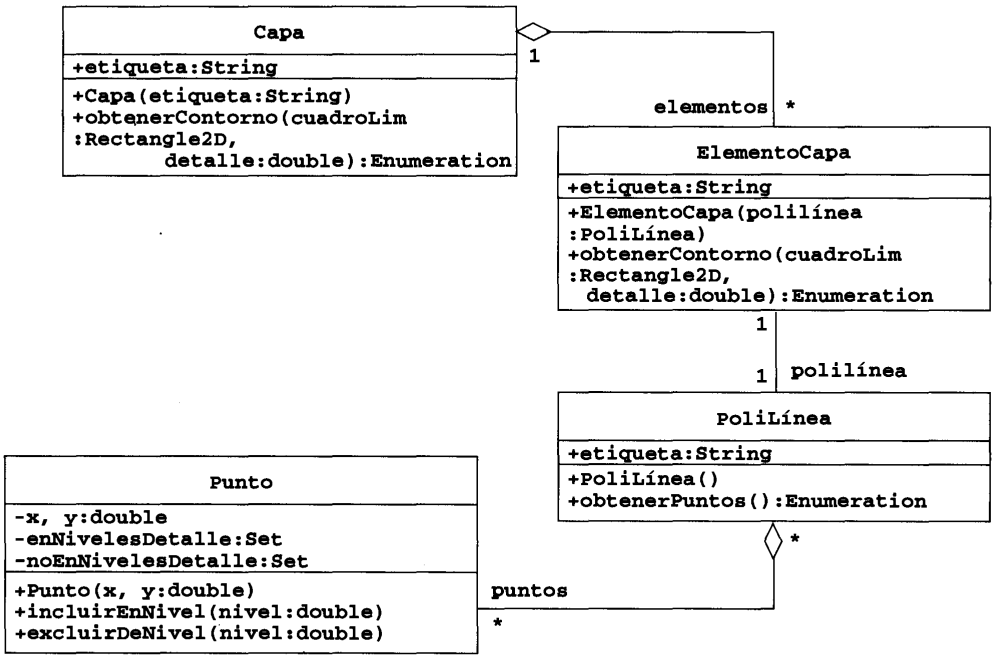


Figura 7-15 Adición de información de tipo al modelo de objetos de GIS (diagrama de clase UML). Sólo se muestran unas clases seleccionadas por brevedad.

7.4.3 Especificación de restricciones

Durante este paso añadimos restricciones a las clases y operaciones para especificar con mayor precisión su comportamiento y casos de frontera. Nuestro objetivo principal es eliminar la mayor cantidad de ambigüedad posible del modelo. Especificamos contratos de clase usando tres tipos de restricciones. Los invariantes representan condiciones de los atributos de una clase que siempre son ciertas. Las precondiciones representan condiciones que deben ser satisfechas (por lo general, por quien llama) antes de llamar a una operación dada. Las poscondiciones representan condiciones que están garantizadas por quien fue llamado después de que termina la operación. Como se describió en las secciones 7.3.3 y 7.3.4, podemos usar OCL [OMG, 1998] para añadir restricciones a los modelos UML.

En el ejemplo JEWEL, el comportamiento más complejo está asociado con el recorte y acercamiento; en particular las operaciones `obtenerContorno()` de las clases `Capa` y `ElementoCapa`. A continuación desarrollamos restricciones para aclarar las operaciones `obtenerContorno()`, enfocándonos en el comportamiento asociado con los puntos compartidos. De manera más concreta, estamos interesados en especificar las siguientes restricciones.

1. Todos los puntos regresados por `Capa.obtenerContorno()` están dentro del cuadro limitante especificado.
2. El resultado de `Capa.obtenerContorno()` es la concatenación de la invocación de `ElementoCapa.obtenerContorno()` sobre sus elementos.

3. A lo mucho, un Punto en el sistema representa una coordenada (x,y) dada.
4. Un nivel de detalle no puede ser parte de los conjuntos enNivelesDetalle y noEnNivelesDetalle al mismo tiempo.
5. Para un nivel de detalle dado, ElementoCapa.obtenerContorno() sólo puede regresar Punto que contengan el nivel detalle en su atributo de conjunto enNivelesDetalle.
6. Los conjuntos enNivelesDetalle y noEnNivelesDetalle sólo pueden crecer a consecuencia de ElementoCapa.obtenerContorno(). En otras palabras, una vez que un nivel de detalle está en alguno de esos conjuntos no puede ser eliminado.

Primero nos enfocamos en el recorte. Tomando un ElementoCapa dado, la enumeración de los puntos regresados por la operación obtenerContorno(cuadroLim,detalle) debe estar dentro del rectángulo cuadroLim especificado. Además, cualquier punto regresado por obtenerContorno() debe estar asociado con ElementoCapa. Representamos esto usando una poscondición sobre la operación obtenerContorno() de ElementoCapa. Observe que, debido a que actualmente nos enfocamos en el recorte, ignoramos el parámetro detalle.

```
/* Restricción 1 */
context ElementoCapa::obtenerContorno(cuadroLim, detalle) post:
resultado->forAll(p:Punto|cuadroLim.contiene(p) and puntos->incluye(p))
```

El campo resultado representa el resultado de la operación obtenerContorno(). La construcción forAll de OCL aplica la restricción a todos los puntos de resultado. Por último, la restricción expresa que todos los puntos de resultado deben estar contenidos en un rectángulo cuadroLim pasado como parámetro y deben estar incluidos en la asociación de agregación puntos de la figura 7-15.

Luego definimos la operación obtenerContorno() sobre una Capa como la concatenación de las enumeraciones regresadas por la operación obtenerContorno() de ElementoCapa. En OCL usamos la construcción iterate sobre colecciones para avanzar por cada ElementoCapa y recopilar su contorno en una sola enumeración. La construcción including añade sus parámetros a la colección. OCL aplana de manera automática la colección resultante.

```
/* Restricción 2 */
context Capa::obtenerContorno(cuadroLim, detalle) post:
elementos->iterate(ec:ElementoCapa; resultado:Enumeration|
    resultado->including(ec.obtenerCapa(cuadroLim,detalle))
```

Luego nos enfocamos en las restricciones relacionadas con el acercamiento. Recuerde que añadimos atributos y operaciones a la clase Punto para representar puntos compartidos. Primero especificamos la unicidad de Punto con un invariante aplicado a todas las instancias de Punto:

```
/* Restricción 3 */
context Punto inv:
Punto.allInstances->forAll(p1, p2:Punto |
    (p1.x = p2.x and p1.y = p2.y) implica p1 = p2)
```

Dejamos la derivación de las tres últimas restricciones como ejercicio al lector (vea el ejercicio 2).

Con estas seis restricciones describimos con más precisión el comportamiento de las operaciones `obtenerContorno()` y su relación con los atributos y operaciones de la clase `Punto`. Observe que no hemos descrito en ninguna forma el algoritmo por el cual `ElementoCapa` selecciona `Punto` en un nivel de detalle dado. Dejamos esta decisión a la actividad de implementación del diseño de objetos.

A continuación describimos las excepciones que puede presentar cada operación.

7.4.4 Especificación de excepciones

Durante este paso especificamos restricciones que necesita satisfacer quien llama antes de llamar una operación. En otras palabras, especificamos condiciones que las operaciones detectan y tratan como errores elevando una excepción. Los lenguajes como Java y Ada tienen mecanismos integrados para el manejo de excepciones. Otros lenguajes como el C y las primeras versiones de C++ no soportan el manejo explícito de excepciones, y por eso los desarrolladores necesitan establecer convenciones y mecanismos para el manejo de excepciones (por ejemplo, valores de retorno o un subsistema especializado). Las condiciones excepcionales se asocian, por lo general, con la violación de precondiciones. En UML añadimos precondiciones OCL a las operaciones y asociamos la precondición con una dependencia hacia un objeto de excepción.

En el ejemplo JEWEL (figura 7-16) especificamos que el parámetro `cuadroLim` de la operación `Capa.obtenerContorno()` debe tener anchura y altura positivas, y que el parámetro detalle debe ser positivo. Asociamos las excepciones `CuadroLimitanteCero` y `DetalleCero` con cada condición.

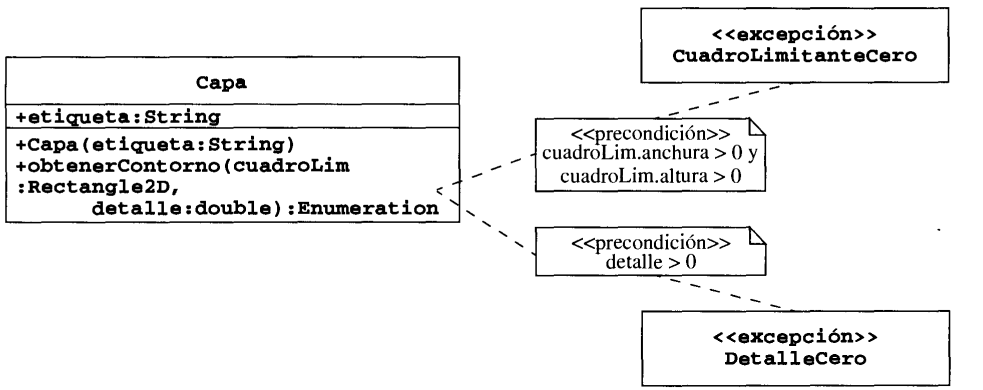


Figura 7-16 Ejemplos de precondiciones y excepciones para la clase `Capa` del `GIS` de JEWEL.

Las excepciones pueden encontrarse de manera sistemática examinando cada parámetro de la operación y examinando los estados en los que puede ser invocada la operación. Para cada parámetro y conjunto de parámetros identificamos valores o combinaciones de valores que no deben aceptarse. Por ejemplo, para el nivel de detalle rechazamos valores que no sean positivos debido a que el parámetro de detalle representa un factor de multiplicación. Un valor cero para el parámetro de

detalle daría como resultado el colapso de todas las coordenadas en el origen. Un nivel de detalle negativo daría como resultado una imagen invertida. Observe que el descubrimiento sistemático de todas las excepciones para todas las operaciones es un ejercicio que consume tiempo pero que es útil. Para los sistemas en los cuales la confiabilidad no es un objetivo de diseño principal, la especificación de excepciones puede limitarse a la interfaz pública de los subsistemas.

Actividades para la selección de componentes

En este punto del diseño de objetos seleccionamos la plataforma de software y hardware en la que se ejecutará el sistema. Esta plataforma incluye componentes hechos como los sistemas de administración de base de datos, marcos middleware, marcos de infraestructura o marcos de aplicación empresarial. El objetivo principal en la selección de componentes hechos es la reutilización de la mayor cantidad de objetos posible, minimizando así la cantidad de objetos personalizados que es necesario desarrollar. Es más, un componente hecho proporciona, con frecuencia, una solución más confiable y eficiente que la que podría esperar producir cualquier desarrollador en el contexto de un solo sistema. Una biblioteca de clases de interfaz, por ejemplo, pone mucha atención a un algoritmo de desplegado eficiente o a buenos tiempos de respuesta. Además, un componente hecho ha sido usado por mucho más sistemas y usuarios y, por tanto, es más robusto. Sin embargo, los componentes hechos tienen un costo. Su propósito es soportar una amplia variedad de sistemas y, en consecuencia, normalmente son complejos. El uso de un componente hecho requiere una inversión en su aprendizaje y, a menudo, requiere algún grado de personalización. El uso de componentes hechos es, por lo general, una mejor alternativa que la construcción del sistema completo a partir de cero.

7.4.5 Identificación y ajuste de bibliotecas de clase

Supongamos que seleccionamos las clases fundamentales Java (JFC, por sus siglas en inglés) [JFC, 1999] como componente hecho para la realización del subsistema `Visualización`. Necesitamos desplegar el mapa como una serie de polilíneas y polígonos regresados por la operación `Capa.obtenerContorno()`. Esto, por lo general, no es directo, ya que tenemos que reconciliar la funcionalidad proporcionada por el `GIS` y las JFC para realizar los servicios de `Visualización`. Por ejemplo, esto puede introducir la necesidad de objetos personalizados cuya única función es convertir datos de un subsistema hacia otro.

Para el desplegado de gráficos, las JFC proporcionan varios componentes reutilizables para la composición de una interfaz de usuario. Las JFC acomodan los componentes en una jerarquía contenedora que restringe el orden en el que éstos se dibujan. Las JFC trazan al último el componente que está más cercano a la parte inferior de la jerarquía (por lo general, componentes atómicos) para que aparezcan por encima de todos los demás componentes. Un `JFrame`, también conocido como ventana principal, define un área que es propiedad exclusiva de la aplicación. Un `JFrame` a menudo se compone de varios `JPanel`, donde cada uno es responsable de la capa de varios componentes atómicos. Un `JScrollPane` proporciona una vista desplazable de un componente. Permite que un usuario vea un subconjunto de un componente que es demasiado grande para desplegarlo por completo.

Para el subsistema `Visualización` de JEWEL (vea la figura 7-17) seleccionamos un `JFrame` como contenedor de nivel superior, un `JToolbar` y dos `JButton` para agrandamiento y achicamiento y un `JScrollPane` para el desplazamiento del mapa. Realizamos el mapa propiamente con la clase `ÁreaMapa`, la cual refina a `JPanel` y sobrepone la operación `paintContents()`. La nueva operación

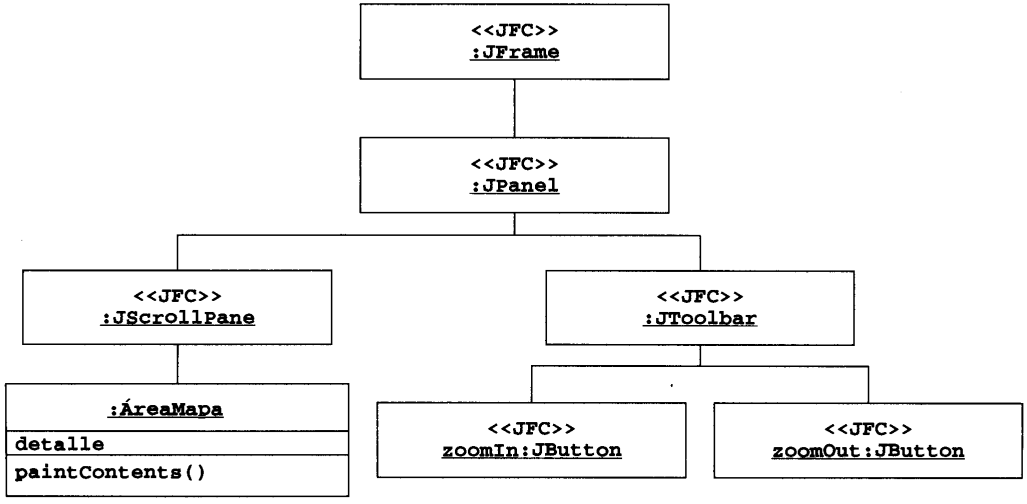


Figura 7-17 Componentes de las JFC para el subsistema `Visualización` de JEWEL (diagrama de objetos UML). Las asociaciones indican la jerarquía contenedora usada para ordenar los componentes del trazado. Usamos estereotipos para distinguir entre las clases de JEWEL y las proporcionadas por las JFC.

`paintContents()` calcula el cuadro limitante visible a partir de atributos de `JScrollPane`, recupera listas de puntos de las clases `Capa`, las escala y las traza. La clase `ÁreaMapa` también mantiene el nivel de detalle actual. Las acciones asociadas con `zoomIn:JButton` y `zoomOut:JButton` tienen acceso a operaciones en `ÁreaMapa` para incrementar y decrementar el nivel de detalle, respectivamente. Esto activa la operación `repaint()` de `ÁreaMapa`, la cual refresca el desplegado del mapa.

Cuando examinamos los primitivos de trazado proporcionados por las JFC nos damos cuenta que las JFC y el `GIS` representan las líneas en forma diferente. Por un lado, las operaciones `drawPoligon()` y `drawPolyline()` de la clase `Graphics` aceptan dos arreglos de coordenadas (uno para las coordenadas x de los puntos y el otro para las coordenadas y, vea la figura 7-18). Por otro lado, la operación `obtenerContorno()` del `GIS` regresa una `Enumeration` de `Punto` (vea la figura 7-16). Hay dos enfoques para resolver esta falta de concordancia. Podemos escribir un método de utilería en la clase `ÁreaMapa` para traducir entre las dos estructuras de datos diferentes o podemos pedirle a los desarrolladores responsables del `GIS` que cambien la interfaz de la clase `Capa`.

```
// del paquete java.awt
class Graphics
//...
    void drawPolyline(int[] xPoints, int[] yPoints, int nPoints) {…};
    void drawPolygon(int[] xPoints, int[] yPoints, int nPoints) {…};
```

Figura 7-18 Declaración para las operaciones `drawPolyline()` y `drawPolygon()` [JFC, 1999].

Debemos cambiar la API de la clase `Capa` si tenemos control sobre ella. Sin embargo, en el caso general, con frecuencia es necesario escribir operaciones de unión y clases. Por ejemplo, si el `GIS` también estuviera establecido con componentes hechos, no podríamos cambiar su API. Podemos usar el patrón `Adaptador` para resolver esta falta de concordancia (vea la sección 6.4.4 en el capítulo 6, *Diseño del sistema*).

7.4.6 Identificación y ajuste de marcos de aplicación

Un **marco de aplicación** es una aplicación parcial reutilizable que puede especializarse para producir aplicaciones personalizadas [Johnson *et al.*, 1988]. A diferencia de las bibliotecas de clase, los marcos están orientados hacia tecnologías particulares, como el procesamiento de datos o las comunicaciones celulares, o hacia dominios de aplicación, como las interfaces de usuario o la aviónica en tiempo real. Los beneficios principales de los marcos de aplicación son la reutilización y la extensibilidad. La reutilización de los marcos se apoya en el conocimiento del dominio de aplicación y los esfuerzos anteriores de desarrolladores experimentados para evitar volver a crear y validar soluciones recurrentes. Un marco de aplicación mejora la extensibilidad proporcionando **métodos gancho**, que son sobrepuestos por la aplicación para extender el marco de aplicación. Los métodos gancho desacoplan en forma sistemática las interfaces y los comportamientos de un dominio de aplicación con respecto a las variaciones requeridas por una aplicación en un contexto particular. La extensibilidad de los marcos es esencial para asegurar la personalización oportuna de nuevos servicios y características de la aplicación.

Los marcos pueden clasificarse por su posición en el proceso de desarrollo del software.

- Los **marcos de infraestructura** ayudan a simplificar el proceso de desarrollo de software. Ejemplos de esto incluyen marcos para sistemas operativos [Campbell-Islam, 1993], depuradores [Bruegge *et al.*, 1993], tareas de comunicación [Schmidt, 1997] y diseño de interfaces de usuario [Weinand *et al.*, 1988]. Los marcos de infraestructura de sistema se usan en forma interna dentro de un proceso de software y, por lo general, no se entregan al cliente.
- Los **marcos middleware** se usan para integrar aplicaciones y componentes distribuidos existentes. Ejemplos comunes incluyen las MFC y DCOM de Microsoft, RMI de Java, WebObjects [Wilson y Ostrem, 1999], las implementaciones de CORBA [OMG, 1995] y bases de datos transaccionales.
- Los **marcos de aplicación empresariales** son específicos de la aplicación, y se enfocan en dominios como las telecomunicaciones, la aviónica, el modelado ambiental [Bruegge y Riedel, 1994], las manufacturas, la ingeniería financiera [Birrer, 1993] y las actividades de negocios empresariales.

Los marcos de infraestructura y middleware son esenciales para crear en forma rápida sistemas de software de alta calidad, pero por lo general no son solicitados por los clientes externos. Sin embargo, los marcos empresariales soportan el desarrollo de aplicaciones de usuario final. En consecuencia, la compra de marcos de infraestructura y middleware es más efectiva en costo que su construcción [Fayad y Hamu, 1997].

Los marcos también pueden clasificarse por las técnicas que se usan para extenderlos.

- Los **marcos de caja blanca** se apoyan en la herencia y el enlace dinámico para la extensibilidad. La funcionalidad existente se extiende haciendo subclases con las clases básicas del marco y sobreponiendo métodos gancho predefinidos, usando patrones como el patrón de método de plantilla [Gamma *et al.*, 1994].
- Los **marcos de caja negra** soportan la extensibilidad definiendo interfaces para los componentes que pueden enchufarse en el marco. La funcionalidad existente se reutiliza mediante la definición de componentes que se apegan a una interfaz particular e integrando estos componentes con el marco usando delegación.

Los marcos de caja blanca requieren un conocimiento íntimo de la estructura interna del marco. Los marcos de caja blanca producen sistemas que están fuertemente acoplados con los detalles específicos de las jerarquías de herencia del marco y, por tanto, los cambios del marco pueden requerir que se vuelva a compilar la aplicación. Los marcos de caja negra son más fáciles de usar que los de caja blanca, debido a que se apoyan en la delegación en vez de en la herencia. Sin embargo, los marcos de caja negra son más difíciles de desarrollar, debido a que requieren la definición de interfaces y ganchos que anticipen un amplio rango de casos de uso potenciales. Además, es más fácil extender y reconfigurar marcos de caja negra en forma dinámica, ya que enfatizan las relaciones de objetos dinámicas en vez de las relaciones de clase estáticas [Johnson *et al.*, 1988].

Los marcos se relacionan íntimamente con los patrones de diseño, las bibliotecas de clases y los componentes.

Patrones de diseño frente a marcos. La principal diferencia entre los marcos y los patrones es que los marcos se enfocan en la reutilización de diseños, algoritmos e implementaciones concretos en un lenguaje de programación particular. Por el contrario, los patrones se enfocan en la reutilización de un diseño abstracto y en pequeñas colecciones de clases cooperativas. Los marcos se enfocan en un dominio de aplicación particular, mientras que los patrones de diseño pueden ser vistos más como bloques de construcción de marcos.

Bibliotecas de clase frente a marcos. Las clases en un marco cooperan para proporcionar un esqueleto arquitectónico reutilizable para una familia de aplicaciones relacionadas. Por el contrario, las bibliotecas de clases son menos específicas del dominio y proporcionan un alcance de reutilización más pequeño. Por ejemplo, los componentes de bibliotecas de clases, como las clases para cadenas, números complejos, arreglos y conjuntos de bits, pueden usarse a través de muchos dominios de aplicación. Las bibliotecas de clases, por lo general, son pasivas; esto es, no implementan ni restringen el flujo de control. Sin embargo, los marcos son activos; es decir, controlan el flujo de control dentro de una aplicación. En la práctica, los marcos y las bibliotecas de clases son tecnologías complementarias. Por ejemplo, los marcos usan bibliotecas de clases en forma interna, como las clases fundamentales, para simplificar el desarrollo del marco. En forma similar, el código específico de la aplicación, llamado por los manejadores de eventos del marco, usan bibliotecas de clases para realizar tareas básicas, como el procesamiento de cadenas, administración de archivos y análisis numérico.

Componentes frente a marcos. Los componentes son instancias de clases autocontenidas que se conectan para formar aplicaciones completas. En términos de reutilización, un componente es una caja negra que define un conjunto de operaciones cohesivo que puede ser usado basándose únicamente en el conocimiento de la sintaxis y semántica de su interfaz. Comparados con los marcos, los componentes están acoplados con menos fuerza e incluso pueden utilizarse en el nivel de código binario. Esto es, las aplicaciones pueden reutilizar los componentes sin tener que hacer subclases a partir de clases base existentes. La ventaja es que las aplicaciones no siempre tienen que volver a compilarse cuando cambian los componentes. La relación entre marcos y componentes no está predeterminada. Por un lado, pueden usarse los marcos para desarrollar componentes, donde la interfaz del componente proporciona un patrón de fachada para la estructura de clases interna del marco. Por otro lado, los componentes pueden conectarse a marcos de caja negra. En general, los marcos se usan para simplificar el desarrollo de software de infraestructura y middleware, mientras que los componentes se usan para simplificar el desarrollo de software de aplicaciones de usuario final.

7.4.7 Un ejemplo de marco: WebObjects

WebObjects es un conjunto de marcos para el desarrollo de aplicaciones Web que tienen acceso a datos existentes en bases de datos relacionales. WebObjects consta de dos marcos de infraestructura. El marco WebObjects[1] maneja la interacción entre los navegadores Web y los servidores Web. El marco de objetos empresarial (EOF, por sus siglas en inglés) maneja la interacción entre los servidores Web y las bases de datos relacionales. El EOF soporta adaptadores de bases de datos que permiten que las aplicaciones se conecten con sistemas de administración de bases de datos de vendedores particulares. Por ejemplo, el EOF proporciona adaptadores para servidores Informix, Oracle y SyBase, y adaptadores apegados a ODBC para bases de datos que ejecutan en la plataforma Windows. A continuación nos concentramos en el marco WebObjects. En [Wilson y Ostrem, 1999] podrá encontrar más información acerca del EOF.

La figura 7-19 muestra un ejemplo de un sitio de edición dinámico construido con WebObjects. El `WebBrowser` origina una petición HTTP en forma de un URL, la cual se envía al `WebServer`. Si el `WebServer` detecta que la petición es una página HTML estática la pasa al objeto `StaticHTML`, el cual selecciona y envía la página de regreso al navegador Web como respuesta. El navegador Web la presenta entonces ante el usuario. Si el `WebServer` detecta que la petición requiere una página HTML dinámica pasa la petición al `WOAdaptor` de WebObjects. El `WOAdaptor` empaca la petición HTML entrante y la pasa al objeto `WebObjectsApplication`. Con base en `Template` definidas por el desarrollador y datos relevantes recuperados de `RelationalDatabase`, la `WebObjectsApplication` genera una página HTML de respuesta, la cual se pasa mediante el `WOAdaptor` hacia el `WebServer`. El `WebServer` envía luego la página al `WebBrowser`, el cual la presenta ante el usuario.

1. Por desgracia, "WebObjects" es el nombre tanto del ambiente de desarrollo completo como del marco Web. Cuando nos referimos al marco siempre usamos la frase marco WebObjects. Cuando nos referimos al ambiente de desarrollo simplemente usamos el término WebObjects.

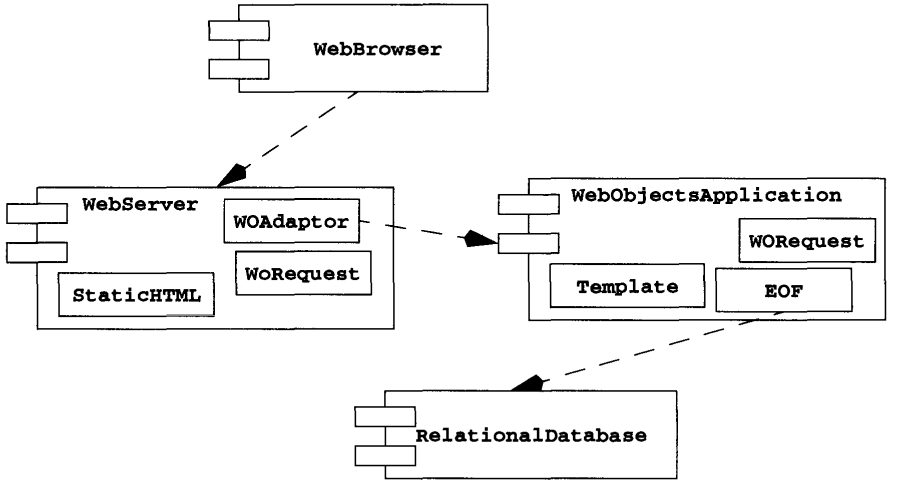


Figura 7-19 Un ejemplo de un sitio dinámico con WebObjects (diagrama de componentes UML).

Una abstracción clave proporcionada por el marco WebObjects es una extensión al protocolo HTTP para manejar el estado. El HTTP es un protocolo de petición y respuesta sin estados; esto es, se formula una respuesta para cada petición pero no se conserva ningún estado entre peticiones sucesivas. Sin embargo, en muchas aplicaciones basadas en Web es necesario mantener el estado entre peticiones. Por ejemplo, en JEWEL los cálculos de emisiones pueden llevarse 30 días. El usuario final debe poder supervisar y tener acceso al estado de los cálculos de las emisiones aunque vuelva a arrancar su navegador Web. Se han propuesto varias técnicas para llevar cuenta de la información de estado en aplicaciones Web, incluyendo URL generados en forma dinámica, cookies y campos HTML ocultos. WebObjects proporciona las clases que se muestran en la figura 7-20 para lograr el mismo propósito.

La clase `WOApplication` representa la aplicación que está ejecutando en el `WebServer`, esperando peticiones del `WebBrowser` asociado. Cada vez que el `WOAdaptor` recibe una petición HTTP entrante se inicia un ciclo de peticiones y respuestas. El `WOAdaptor` empaca esta petición en un objeto `WORequest` y lo pasa al objeto de aplicación de la clase `WOApplication`. Las peticiones siempre son activadas por un URL enviado por el `WebBrowser`. Un URL de nivel más alto representa una petición especial y causa la creación de una nueva instancia del tipo `WOSession`. La clase `WOSession` encapsula el estado de una sesión individual, lo que permite el seguimiento de diferentes usuarios, aun dentro de una sola aplicación. Una `WOSession` consta de uno o más `WOComponent`, los cuales representan una página Web reutilizable, o parte de una página Web para desplegarla dentro de una sesión individual. `WOComponent` puede contener elementos dinámicos. Cuando una aplicación tiene acceso a la base de datos, uno o más de los elementos dinámicos de un componente se llenan con información recuperada de la base de datos. El `WOSessionStore` proporciona persistencia para los objetos `WOSession`: guarda sesiones en el servidor y las restaura para la aplicación cuando las solicitan.

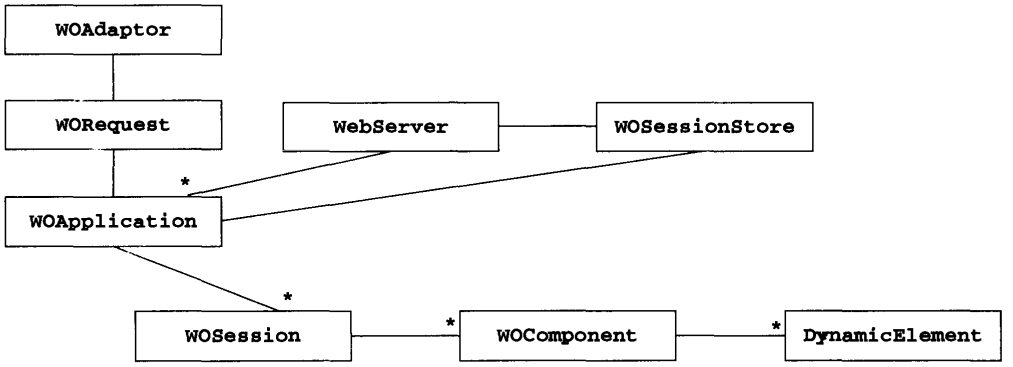


Figura 7-20 Clases de administración de estado de WebObjects. El protocolo HTTP no tiene estados inherentes. Las Clases de administración de estado permiten conservar información entre peticiones individuales.

La esencia de la construcción de aplicaciones de WebObjects es refinar las clases `WOApplication`, `WOSession` y `WOComponent` e interceptar el flujo de peticiones enviadas y recibidas por ellas. Los métodos heredados de esas clases se sobreponen cuando el desarrollador necesita extender el comportamiento predeterminado. El primer punto de control para la refinación de objetos de tipo `WOApplication` es cuando se les construye. El último punto de control es cuando termina el objeto de aplicación. Mediante la adición de código al constructor del objeto de aplicación, o mediante la sobreposición del método `terminate()` de `WOApplication`, el desarrollador puede personalizar el comportamiento de la aplicación WebObjects como lo desee.

Una vez que se ha extendido el modelo de diseño de objetos con componentes hechos y sus clases relacionadas reestructuramos el modelo para mejorar la reutilizabilidad y la extensibilidad.

Actividades de reestructuración

Una vez que especificamos las interfaces de subsistemas, identificamos clases de solución adicionales, seleccionamos componentes y los adaptamos para que se ajusten a nuestra solución, necesitamos transformar el modelo de diseño de objetos hacia una representación que esté más cercana a la máquina de destino. En esta sección describimos tres actividades de reestructuración:

- Realización de asociaciones (sección 7.4.8).
- Revisión de la herencia para incrementar la reutilización (sección 7.4.9).
- Revisión de la herencia para eliminar dependencias de implementación (sección 7.4.10).

7.4.8 Realización de asociaciones

Las asociaciones son conceptos UML que indican colecciones de vínculos bidireccionales entre dos o más objetos. Sin embargo, los lenguajes de programación orientados a objetos no proporcionan el concepto de asociación. En vez de ello proporcionan referencias en donde un objeto guarda una manija hacia otro objeto. Las referencias son unidireccionales y se realizan entre dos objetos. Durante el diseño de objetos realizamos asociaciones desde el punto de vista de referencias,

tomando en cuenta la multiplicidad de las asociaciones y su dirección. Observe que muchas herramientas de modelado UML realizan la transformación de asociaciones hacia referencias en forma automática. Aunque una herramienta realice esta transformación, sigue siendo importante que los desarrolladores comprendan su fundamentación, ya que tiene que ver con el código generado.

Asociaciones unidireccionales de uno a uno. La asociación más simple es la de uno a uno. Por ejemplo, `AcciónAcercamiento`, el objeto de control que implementa el caso de uso `Acercamiento`, tiene una asociación de uno a uno con el `ÁreaMapa` cuyo nivel de `detalle` modifica el objeto `AcciónAcercamiento` (figura 7-21). Además, supongamos que esta asociación es unidireccional; esto es, un `AcciónAcercamiento` tiene acceso al `ÁreaMapa` correspondiente, pero un `ÁreaMapa` no necesita tener acceso al objeto `AcciónAcercamiento` correspondiente. En este caso realizamos esta asociación usando una referencia desde `AcciónAcercamiento`; esto es, un atributo de `AcciónAcercamiento` llamado `mapaDestino` de tipo `ÁreaMapa`.

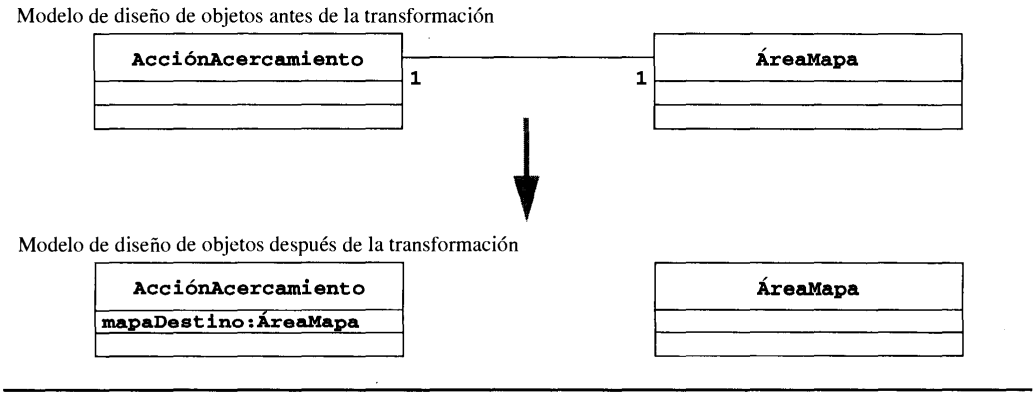
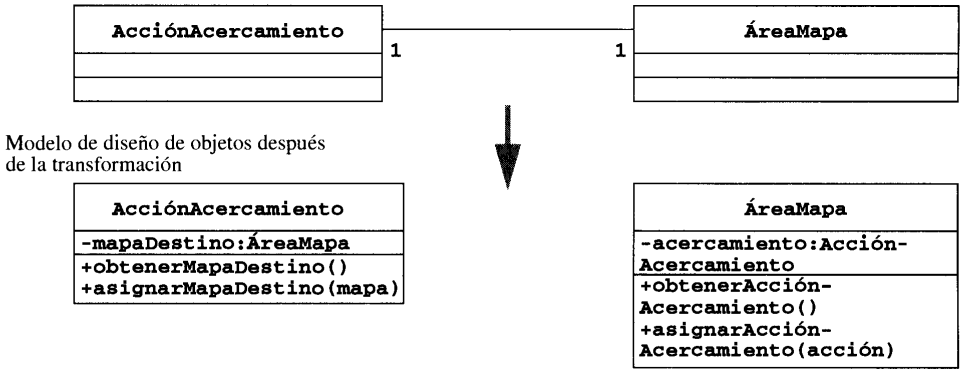


Figura 7-21 Realización de una asociación unidireccional de uno a uno (diagrama de clase UML; la flecha indica la transformación del modelo de objetos).

La creación de una asociación entre `AcciónAcercamiento` y `ÁreaMapa` se traduce en la especificación del atributo `mapaDestino` para que haga referencia al objeto `ÁreaMapa` correcto. Debido a que cada objeto `AcciónAcercamiento` está asociado exactamente con un `ÁreaMapa`, sólo puede suceder un valor `nulo` para el atributo `mapaDestino` cuando se está creando un objeto `AcciónAcercamiento`. En los demás casos, un `mapaDestino` `nulo` se considera un error.

Asociaciones bidireccionales de uno a uno. Supongamos que modificamos la clase `ÁreaMapa` para que el usuario pueda hacer acercamientos simplemente haciendo clic en el mapa con los botones izquierdo y derecho. En este caso, un `ÁreaMapa` necesita acceder a su objeto `AcciónAcercamiento` correspondiente. Por consecuencia, la asociación entre estos dos objetos necesita ser bidireccional. Añadimos el atributo `acercamiento` a `ÁreaMapa` (figura 7-22). Sin embargo, esto no es suficiente: mediante la adición de un segundo atributo para realizar la asociación introducimos redundancia al modelo. Necesitamos asegurarnos que si un `ÁreaMapa` dada tiene una referencia hacia un `AcciónAcercamiento` específico, el `AcciónAcercamiento` tenga una referencia hacia la misma `ÁreaMapa`. Para asegurar la consistencia cambiamos la visibilidad del atributo a privada y añadimos dos métodos a cada clase para acceder a ellas y modificarlas. `asignarAcciónAcercamiento()` en

ÁreaMapa asigna el atributo acercamiento de su parámetro y luego llama a asignarMapaDestino() de AcciónAcercamiento para cambiar su atributo mapaDestino.[2] Por último, necesitamos tratar la iniciación de la asociación y su destrucción llamando a asignarMapaDestino() y asignarAcciónAcercamiento() cuando se crean y destruyen los objetos ÁreaMapa y AcciónAcercamiento. Esto asegura que ambos atributos de referencia sean consistentes todo el tiempo.




```
class ÁreaMapa extends JPanel {
    private AcciónAcercamiento acercamiento;
    /* Se omiten otros métodos */
    void asignarAcciónAcercamiento (acción:AcciónAcercamiento) {
        if (acercamiento != acción) {
            acercamiento = acción;
            acercamiento.asignarMapaDestino (this);
        }
    }
}
class AcciónAcercamiento extends AbstractAction {
    private ÁreaMapa MapaDestino;
    /* Se omiten otros métodos */
    void asignarMapaDestino(mapa:ÁreaMapa) {
        if (MapaDestino != mapa) {
            MapaDestino = mapa;
            MapaDestino.asignarAcciónAcercamiento (this);
        }
    }
}
```

Figura 7-22 Realización de una asociación bidireccional de uno a uno (diagrama de clase UML y fragmentos Java; la flecha indica la transformación del modelo de diseño de objetos).

2. Observe que los métodos asignarAcciónAcercamiento() y asignarMapaDestino() necesitan revisar primero si el atributo necesita modificarse antes de llamar al otro método para que eviten una recursión infinita (vea el código en la figura 7-22).

La dirección de una asociación a menudo puede cambiar durante el desarrollo del sistema. Las asociaciones unidireccionales pueden realizarse de manera mucho más simple. Las asociaciones bidireccionales son más complejas e introducen dependencias mutuas entre clases. Por ejemplo, en la figura 7-22 las dos clases, `ÁreaMapa` y `AcciónAcercamiento`, necesitan volver a compilarse y probarse cuando cambiamos alguna de ellas. En el caso de una asociación unidireccional de la clase `AcciónAcercamiento` hacia la clase `ÁreaMapa`, no necesitamos preocuparnos acerca de la clase `ÁreaMapa` cuando cambiamos la clase `AcciónAcercamiento`. Sin embargo, las asociaciones bidireccionales son, a veces, necesarias en el caso de clases pares que necesitan trabajar en forma muy estrecha. La selección entre una asociación unidireccional o bidireccional es un compromiso que tenemos que evaluar en el contexto de un par específico de clases. Sin embargo, para facilitar el compromiso podemos hacer de manera sistemática que todos los atributos sean privados y proporcionar las operaciones `asignarAtributo()` y `obtenerAtributo()` para modificar la referencia. Esto minimiza los cambios a las interfaces de clases cuando hacemos que una asociación unidireccional sea bidireccional (y viceversa).

Asociaciones de uno a muchos. Las asociaciones de uno a muchos, a diferencia de las asociaciones de uno a uno, no pueden realizarse usando una sola referencia o un par de referencias. En vez de ello, realizamos la parte de "muchos" usando una colección de referencias. Por ejemplo, la clase `Capa` del `GIS` de JEWEL tiene una asociación de uno a muchos con la clase `ElementoCapa`. Debido a que `ElementoCapa` no tiene un orden específico con respecto a `Capa`, y debido a que `ElementoCapa` puede ser parte, a lo sumo, de una `Capa` a la vez, usamos un conjunto de referencias para modelar la parte "muchos" de la asociación. Además, decidimos realizar esta asociación como una asociación bidireccional y, por tanto, añadimos los métodos `añadirElemento()`, `eliminarElemento()`, `obtenerCapa()` y `asignarCapa()` a las clases `Capa` y `ElementoCapa` para actualizar los atributos `elementosCapa` y `contenidoEn` (vea la figura 7-23). Al igual que en el ejemplo de uno a uno, es necesario iniciar y destruir la asociación cuando se crean y destruyen los objetos `Capa` y `ElementoCapa`.

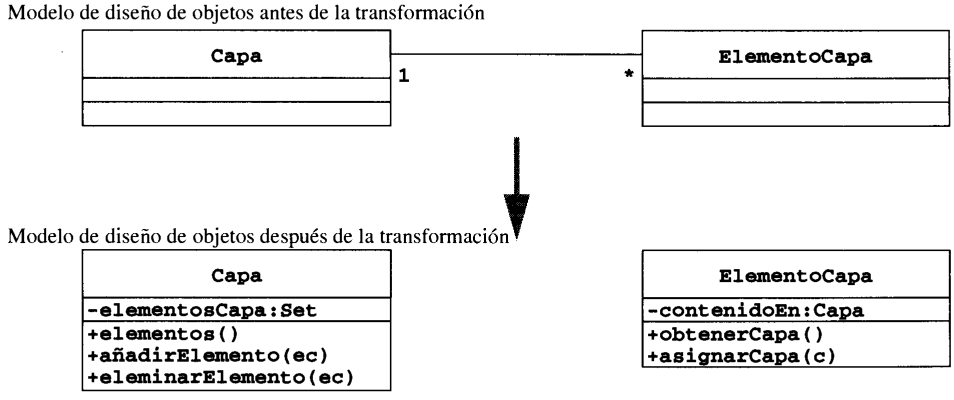


Figura 7-23 Realización de una asociación bidireccional de uno a muchos (diagrama de clase UML; la flecha indica la transformación del modelo de diseño de objetos).

Observe que la colección del lado "muchos" de la asociación depende de las restricciones de la asociación. Por ejemplo, si el `ElementoCapa` de una `Capa` necesita estar ordenado (por ejemplo, para indicar el orden en que deben trazarse), necesitamos usar un `Array` o un `Vector` en vez de un `Set`. En forma similar, si una asociación está calificada usamos una `Hashtable` para guardar las referencias.

Asociaciones de muchos a muchos. En este caso, ambas clases terminales tienen atributos que son colecciones de referencias y operaciones para mantener consistentes estas colecciones. Por ejemplo, las clase `PoliLínea` del `GIS` de JEWEL tiene una asociación ordenada de muchos a muchos con la clase `Punto`. Esta asociación se realiza usando un atributo `Vector` en cada clase, el cual es modificado por las operaciones `añadirPunto()`, `eliminarPunto()`, `añadirPolilínea()` y `eliminarPolilínea()` (vea la figura 7-24). Al igual que en el ejemplo anterior, estas operaciones aseguran que ambos `Vector` sean consistentes. Sin embargo, observe que la asociación entre `PoliLínea` y `Punto` debe ser unidireccional, tomando en cuenta que ninguna de las operaciones de `Punto` necesita tener acceso a las `PoliLínea` que incluyen un `Punto` dado. Podemos entonces eliminar el atributo `polilíneas` y sus métodos relacionados y, en este caso, una asociación unidireccional de muchos a muchos o una asociación unidireccional de uno a muchos llega a ser idéntica en el nivel del diseño de objetos.

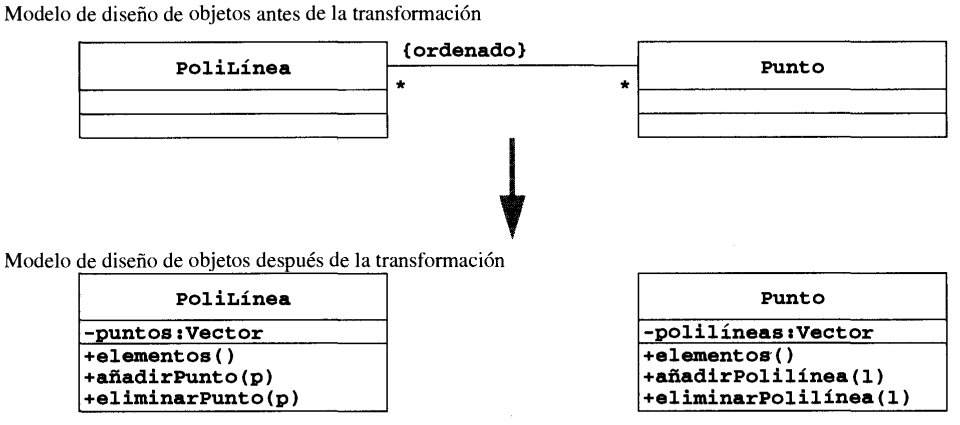


Figura 7-24 Realización de una asociación bidireccional de muchos a muchos (diagrama de clase UML; la flecha indica la transformación del modelo de diseño de objetos).

Las asociaciones como objetos separados. En UML las asociaciones pueden estar asociadas con una clase de asociación que guarda los atributos y operaciones de la asociación. Primero transformamos la clase de asociación en un objeto separado y varias asociaciones binarias. Por ejemplo, considere la asociación `EjecutarSimulación` en JEWEL (figura 7-25). Una `EjecutarSimulación` relaciona un objeto `FuenteEmisión` y un objeto `ResultadoSimulación`. La clase de asociación `EjecutarSimulación` también guarda atributos específicos de la corrida, como la fecha en que se creó, el usuario que ejecutó la simulación y el tiempo de CPU que se requirió para terminar la

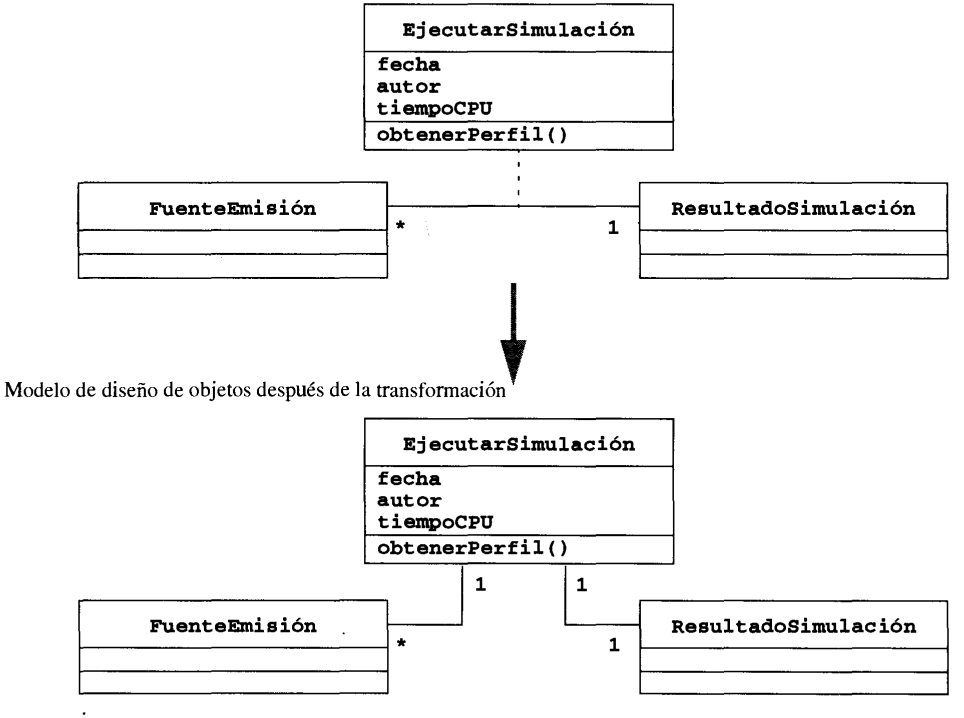


Figura 7-25 Transformación de una clase de asociación en un objeto y dos asociaciones binarias (diagrama de clase UML; la flecha indica la transformación del modelo de diseño de objetos). Una vez que el modelo contiene sólo asociaciones binarias, cada asociación se realiza usando atributos de referencia y colecciones de referencias.

simulación. Primero convertimos la asociación en un objeto, llamado `EjecutarSimulación`, y dos asociaciones binarias entre el objeto `EjecutarSimulación` y los demás objetos. Luego podemos usar las técnicas tratadas antes para convertir cada asociación binaria en un conjunto de atributos de referencia.

Asociaciones calificadas. En este caso, uno o ambos extremos de la asociación están asociados con una clave que se usa para diferenciar entre asociaciones. Las asociaciones calificadas se realizan en la misma forma que las asociaciones uno a muchos y muchos a muchos, a excepción del uso de un objeto `Hashtable` en el extremo calificado (a diferencia de un `Vector` o un `Set`). Por ejemplo, considere la asociación entre `Escenario` y `EjecutarSimulación` en JEWEL (figura 7-26). Un `Escenario` representa una situación que están investigando los usuarios (por ejemplo, una fuga en un reactor nuclear). Para cada `Escenario` los usuarios pueden crear varias `EjecutarSimulación`, usando cada una un conjunto diferente de `FuenteEmisión` o un `ModeloEmisión` diferente. Tomando en cuenta que `EjecutarSimulación` es costoso, los usuarios también reutilizan las ejecuciones a través de `Escenario` similares. El extremo de la asociación de `Escenario` está calificado con un nombre que permite al usuario distinguir entre varias `EjecutarSimulación` con el mismo `Escenario`.

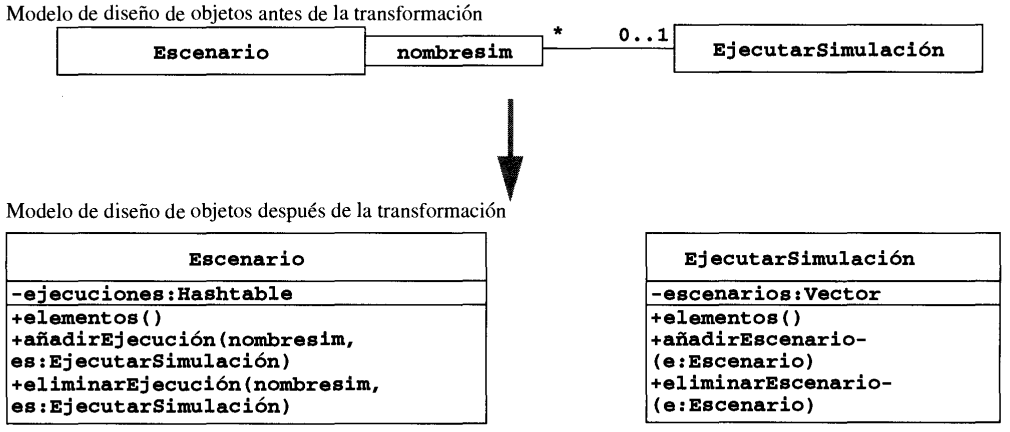


Figura 7-26 Realización de una asociación bidireccional calificada (diagrama de clase UML; la flecha indica la transformación del modelo de diseño de objetos).

Realizamos esta asociación calificada creando un atributo `ejecuciones` en `Escenario` y un atributo `escenarios` en `EjecutarSimulación`. El atributo `ejecuciones` es una `Hashtable` que está indexada por el nombre de una `EjecutarSimulación`. Debido a que el nombre está guardado en la `Hashtable`, una `EjecutarSimulación` específica puede tener varios nombres diferentes a través de `Escenario`. El extremo `EjecutarSimulación` se realiza, como antes, como un `Vector` en la clase `EjecutarSimulación`.

7.4.9 Incremento de la reutilización

La herencia permite que los desarrolladores reutilicen el código a través de varias clases similares. Por ejemplo, las JFC, al igual que la mayoría de los juegos de herramientas de interfaz de usuario, proporcionan cuatro tipos de botones:

- Un botón oprimible (`JButton`) que activa una acción cuando el usuario final hace clic en el botón.
- Un botón de radio (`JRadioButton`) que permite que un usuario final seleccione una opción entre un conjunto de opciones.
- Un cuadro de verificación (`JCheckBox`) que permite que un usuario final active o desactive una opción.
- Un concepto de menú (`JMenuItem`) que activa una acción cuando se le selecciona en un menú desplegable o emergente.

Estos cuatro botones comparten un conjunto de atributos (por ejemplo, un rótulo, un icono) y un comportamiento (por ejemplo, algo sucede cuando los selecciona el usuario final). Sin embargo, el comportamiento de cada tipo de botón es ligeramente diferente. Para acomodar estas diferencias, y al mismo tiempo reutilizar la mayor cantidad de código posible, las JFC introducen dos clases abstractas, `AbstractButton` y `JToggleButton`, y organiza estos cuatro tipos de botones en la jerarquía de herencia que se muestra en la figura 7-27. La clase `AbstractButton` define el comportamiento compartido por todos los botones de las JFC. `JToggleButton` define el comportamiento compartido por los dos botones de estado (es decir, `JRadioButton` y `JCheckBox`).

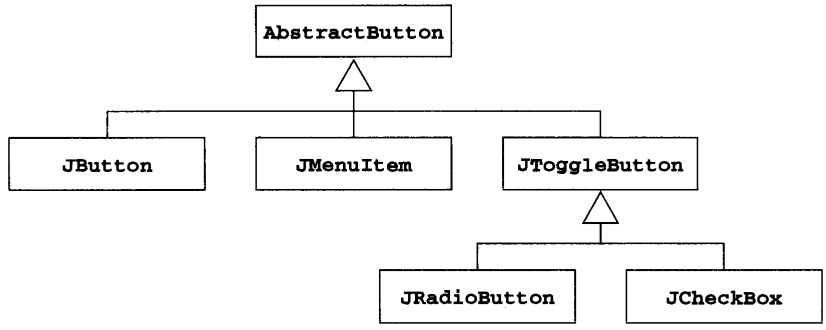


Figura 7-27 Un ejemplo de reutilización de código con herencia (diagrama de clase UML).

Hay dos ventajas principales en la utilización de una jerarquía de herencia bien diseñada. La primera es que se reutiliza más código, y esto conduce a menos redundancias y, por tanto, a menos oportunidades para que haya defectos. La segunda es que el código es extensible, incluyendo una interfaz bien documentada para la creación de especializaciones futuras (por ejemplo, nuevos tipos de botones en el caso de las JFC). Sin embargo, la reutilización mediante herencia tiene un costo. Los desarrolladores deben anticipar en forma correcta cuál comportamiento deberá compartirse y cuál será refinado por la especialización, a menudo sin conocer todas las especializaciones posibles. Además, una vez que los desarrolladores definen una jerarquía de herencia y un paradigma para la compartición del código, las interfaces de las clases abstractas se hacen cada vez más rígidas ante el cambio conforme cada vez más clases cliente dependen de ellas. El diseño de objetos representa la última oportunidad durante el desarrollo para revisar las jerarquías de herencia entre objetos de aplicación y de solución. Cualquier cambio posterior en el proceso puede introducir errores difíciles de detectar e incrementa de manera considerable el costo del sistema.

Hay dos enfoques principales para el diseño de una jerarquía de herencia a fin de reutilizarla. Primero, podemos examinar varias clases similares y abstraer su comportamiento común. El ejemplo `AbstractButton` de la figura 7-27 es un ejemplo de este enfoque. Segundo, podemos desacoplar una clase cliente de un cambio anticipado introduciendo un nivel de abstracción. La mayoría de los patrones de diseño [Gamma *et al.*, 1994], incluyendo al patrón `FábricaAbstracta` que se muestra a continuación, usan herencia para protegerse contra un cambio anticipado.

Considere el problema de escribir una sola aplicación que funcione con varios estilos de ventanas (por ejemplo, Windows, Macintosh y Motif). Tomando una plataforma dada, el usuario trabaja con un conjunto consistente de ventanas, barras de desplazamiento, botones y menús. La aplicación misma no debe saber o depender de un apariencia específica. El **patrón Fábrica Abstracta** (figura 7-28) resuelve este problema proporcionando una clase abstracta para cada objeto que puede ser sustituido (por ejemplo, `VentanaAbstracta` y `BotónAbstracto`) y proporcionando una interfaz para la creación de grupos de objetos (es decir, la `FábricaAbstracta`). Clases concretas implementan cada clase abstracta para cada fábrica. Por ejemplo, la clase `BotónAbstracto` está refinada

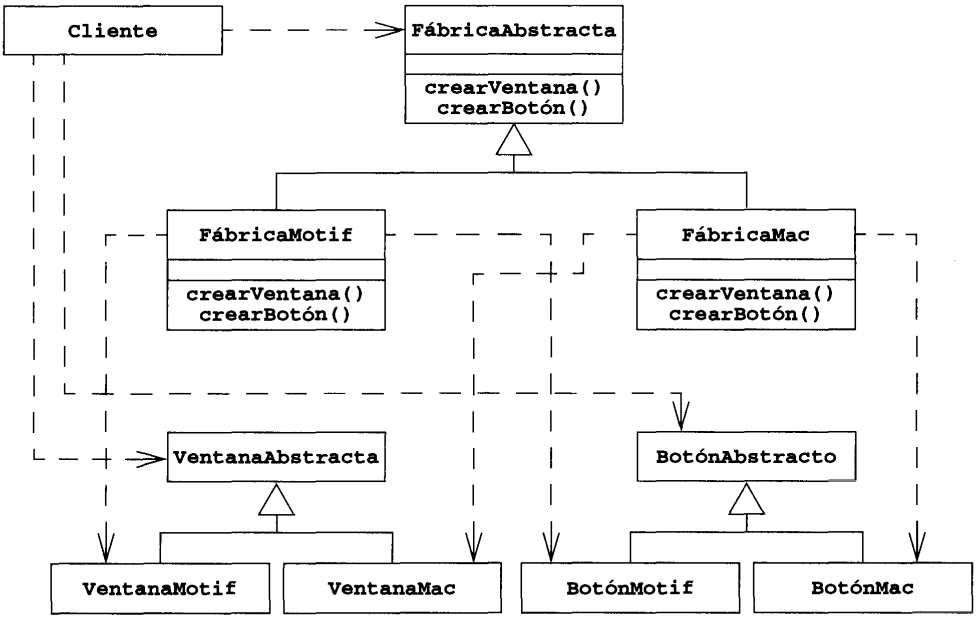


Figura 7-28 Patrón de diseño `FábricaAbstracta` (diagrama de clase UML, las dependencias representan relaciones `<<llama>>`). Este patrón de diseño usa herencia para soportar diferentes apariencias (por ejemplo, Motif y Macintosh). Si se añade una nueva especialización no es necesario cambiar al cliente.

por la clase `BotónMac` y la clase `BotónMotif`. La interfaz `FábricaAbstracta` proporciona una operación `crearBotón()` para crear un botón. Una fábrica concreta implementa la interfaz `FábricaAbstracta` para cada opción. El método `FábricaMotif.crearBotón()` regresa un `BotónMotif`, mientras que el método `FábricaMac.crearBotón()` regresa un `BotónMac`. Observe que ambos métodos `crearBotón()` tienen la misma interfaz para ambas especializaciones. En consecuencia, quien llama sólo accede a la interfaz `FábricaAbstracta` y las clases abstractas y, por tanto, está escudado ante las implementaciones concretas. Además, esto permite que en el futuro se implementen nuevas fábricas (por ejemplo, `FábricaBeOS` y `BotónBeOS`) sin cambiar la aplicación.

7.4.10 Eliminación de las dependencias de implementación

En el modelado del sistema usamos relaciones de generalización para clasificar a los objetos en jerarquías de generalización/especificación. Esto nos permite diferenciar el comportamiento común del caso general con respecto al comportamiento que es específico de los objetos especializados. En un lenguaje de programación orientado a objetos la generalización se realiza mediante la herencia. Esto nos permite reutilizar los atributos y operaciones de clases de más alto nivel. Por un lado, la herencia, cuando se usa como mecanismo de generali-

zación, da como resultado menor cantidad de dependencias. Por ejemplo, en el patrón de diseño FábricaAbstracta (figura 7-28), las dependencias entre la aplicación y una apariencia específica se eliminan usando las clases abstractas FábricaAbstracta, BotónAbstracto y VentanaAbstracta. Por otro lado, la herencia introduce dependencias a lo largo de la jerarquía. Por ejemplo, las clases VentanaMotif y VentanaWindows están fuertemente acopladas con VentanaAbstracta. En el caso de la generalización esto es aceptable, tomando en cuenta que VentanaAbstracta, VentanaMotif y VentanaMac son conceptos fuertemente relacionados. Estas dependencias fuertes pueden llegar a ser un problema cuando se usa la herencia para propósitos diferentes a la generalización. Considere el siguiente ejemplo.

Supongamos por un momento que Java no proporciona una abstracción Set y que necesitamos escribir una nuestra. Decidimos reutilizar la clase java.util.Hashtable para implementar una abstracción de conjunto a la que llamaremos MiConjunto. La inserción de un elemento en MiConjunto equivale a revisar si existe la clave correspondiente en la tabla y la creación de una entrada si es necesario. Revisar si un elemento está en MiConjunto equivale a revisar si una entrada está asociada con la clave correspondiente (vea la figura 7-29, columna izquierda).

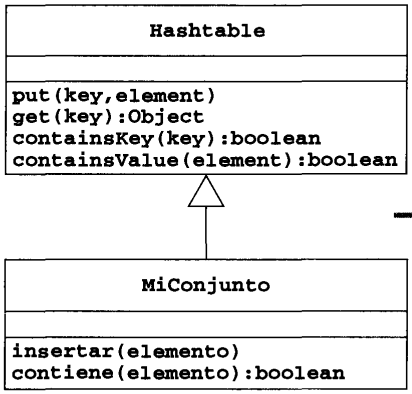
Dicha implementación de un Set nos permite reutilizar código y nos proporciona el comportamiento deseado. Sin embargo, también nos proporciona comportamiento no deseado. Por ejemplo, Hashtable implementa la operación containsKey() para revisar si existe el objeto especificado como una clave en la Hashtable y la operación containsValue() para revisar si el objeto especificado existe como una entrada. Estas dos operaciones son heredadas por MiConjunto. Tomando en cuenta nuestra implementación, la operación containsValue() invocada sobre un objeto MiConjunto siempre regresa null, lo cual es contrario a lo esperado. Lo que sería peor, un desarrollador que usara MiConjunto podría confundir con facilidad las operaciones contains() y containsValue() e introducir una falla en el sistema que es difícil de detectar. Para tratar este asunto podríamos sobreponer todas las operaciones heredadas de Hashtable que no deben usarse en MiConjunto. Esto conduciría a una clase MiConjunto que sería difícil de comprender y reutilizar.

El problema fundamental en este ejemplo es que, aunque Hashtable proporciona comportamiento que nos gustaría reutilizar en la implementación de Set, debido a que nos ahorraría tiempo, el concepto Set no es un refinamiento del concepto Hashtable. Por el contrario, la clase VentanaMac del ejemplo FábricaAbstracta sí es un refinamiento de la clase VentanaAbstracta.

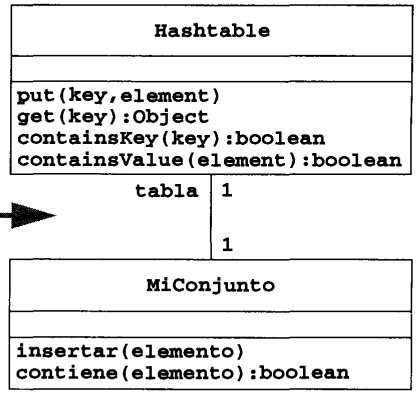
Al uso de la herencia con el único propósito de la reutilización del código le llamamos **herencia de implementación**. La herencia de implementación permite que los desarrolladores reutilicen código haciendo con rapidez subclases de una clase existente y refinando su comportamiento. Un Set implementado por herencia de una Hashtable es un ejemplo de herencia de implementación. Por el contrario, a la clasificación de conceptos en jerarquías de especialización-generalización se le llama **herencia de interfaz**. La herencia de interfaz se usa para manejar la complejidad que se presenta para gran número de conceptos relacionados. A la herencia de interfaz también se le llama subtipeado y, en este caso, a la superclase se le llama **supertipo** y a la subclase se le llama **subtipo**. Por ejemplo, Real e Integer son subtipos de Number. ÁreaMapa es un subtipo de JPanel.

Debe evitarse la herencia de implementación. Aunque proporciona un mecanismo tentador para la reutilización del código, sólo produce beneficios de corto plazo y da como resultado sistemas que son difíciles de modificar. La **delegación** es una mejor alternativa, en vez de la herencia de implementación, si se puede reutilizar el código. Se dice que una clase delega hacia otra clase si implementa una operación simplemente reenviando un mensaje a otra clase. La delegación hace

Modelo de diseño de objetos antes de la transformación



Modelo de diseño de objetos después de la transformación




```
/* Implementación de MiConjunto usando
herencia */
class MiConjunto extends Hashtable {
    /* Se omite el constructor */
    MiConjunto() {
    }

    void insert(Object element) {
        if (this.get(element) == null){
            this.put(element, this);
        }
    }
    boolean contains(Object element){
        return
        (this.get(element)!=null);
    }
    /* Se omiten otros métodos */
}
```

```
/* Implementación de MiConjunto usando
delegación */
class MiConjunto {
    Hashtable table;
    MiConjunto() {
        tabla = Hashtable();
    }
    void insertar(Object elemento) {
        if (tabla.get(elemento)==null){
            tabla.put(elemento,this);
        }
    }
    boolean contiene(Object elemento) {
        return
        (tabla.get(elemento) != null);
    }
    /* Se omiten otros métodos */
}
```

Figura 7-29 Un ejemplo de herencia de implementación. La columna izquierda muestra una implementación cuestionable de `MiConjunto` usando herencia de implementación. La columna derecha muestra una implementación mejorada usando delegación (diagrama de clase UML y Java).

explícitas las dependencias entre la clase reutilizada y la nueva clase. La columna derecha de la figura 7-29 muestra una implementación de `MiConjunto` usando delegación en vez de herencia de implementación. Observe que la única adición significativa es el atributo tabla y su iniciación en el constructor `MiConjunto()`.

Para una exposición amplia de los intercambios relacionados con la herencia y la delegación el lector deberá consultar [Meyer, 1997].

Actividades de optimización

La traducción directa del modelo de análisis da como resultado un modelo que con frecuencia es ineficiente. Durante el diseño de objetos optimizamos el modelo de objetos de acuerdo a objetivos de diseño, como la minimización del tiempo de respuesta, tiempo de ejecución o recursos de memoria. En esta sección describimos cuatro optimizaciones simples:

- La adición de asociaciones para la optimización de rutas de acceso.
- La descomposición de objetos en atributos.
- El cacheo de resultados de cálculos costosos.
- El retraso de cálculos costosos.

Cuando se aplican optimizaciones, los desarrolladores deben establecer un equilibrio entre la eficiencia y la claridad. Las optimizaciones incrementan la eficiencia del sistema, pero también hacen que sea más compleja y difícil la comprensión de los modelos del sistema.

7.4.11 Revisión de las rutas de acceso

Una fuente común del desempeño ineficiente del sistema es el recorrido repetido de asociaciones múltiples cuando se tiene acceso a la información que se necesita. Para identificar las rutas de acceso ineficientes los diseñadores de objetos deben hacerse las siguientes preguntas [Rumbaugh *et al.*, 1991]:

- **Para cada operación:** ¿Con cuánta frecuencia se llama a la operación? ¿Cuáles asociaciones tiene que recorrer la operación para obtener la información que necesita? Las operaciones frecuentes no deben requerir muchos recorridos, sino que deben tener una conexión directa entre el objeto que consulta y el objeto consultado. Si falta esa conexión directa deberá añadirse una asociación adicional entre esos dos objetos.
- **Para cada asociación:** Si tiene una asociación de "muchos" en uno o ambos lados, ¿se necesita la multiplicidad? ¿Con cuánta frecuencia está involucrado el lado "muchos" de una asociación en una búsqueda? Si esto es frecuente el diseñador de objetos deberá tratar de reducir "muchos" a "uno". Si no se puede, ¿deberá ordenarse o indexarse el lado "muchos" para mejorar el tiempo de acceso?

En proyectos de interfaz y reingeniería, las estimaciones para la frecuencia de las rutas de acceso pueden derivarse del sistema heredado. En proyectos de ingeniería greenfield (es decir, sistemas que se desarrollan a partir de cero y que no se pretende que reemplacen a un sistema heredado) es más difícil estimar la frecuencia de las rutas de acceso. En este caso no deben añadirse asociaciones redundantes antes de que un análisis dinámico del sistema completo, por ejemplo, durante las pruebas del sistema, haya determinado cuáles asociaciones participan en los cuellos de botella del desempeño.

Otra fuente del desempeño ineficiente del sistema es el modelado excesivo. Durante el análisis se identifican muchas clases que no tienen comportamiento interesante. En este caso los diseñadores de objetos deben preguntarse:

- **Para cada atributo:** ¿Cuáles operaciones usan el atributo? ¿Son `asignar()` y `obtener()` las únicas operaciones que se realizan sobre ese atributo? Si es así, ¿en realidad pertenece este atributo a este objeto o debe movérsele al objeto que lo llama?

El examen sistemático del modelo de objetos usando las preguntas anteriores debe conducir a un modelo con asociaciones redundantes seleccionadas, con menos asociaciones de muchos a muchos ineficientes y menos clases.

7.4.12 Descomposición de objetos: conversión de objetos en atributos

Durante el análisis los desarrolladores identifican muchas clases que están asociadas con conceptos de dominio. Durante el diseño del sistema y el diseño de objetos se reestructura y optimiza el modelo de objetos, y a menudo deja a algunas de estas clases con sólo unos cuantos atributos y poco comportamiento. Tales clases, cuando se asocian sólo con otra clase, pueden descomponerse hacia un atributo, reduciendo, por tanto, la complejidad general del modelo.

Considere, por ejemplo, un modelo de objetos que incluye `Persona` identificadas por un objeto `SeguroSocial`. Puede ser que se hayan identificado dos clases durante el análisis. Cada `Persona` está asociada con una clase `SeguroSocial`, la cual guarda una cadena de ID único que identifica a la `Persona`. El modelado adicional no revela ningún comportamiento adicional del objeto `SeguroSocial`. Además, ninguna otra clase tiene asociaciones con la clase `SeguroSocial`. En este caso, la clase `SeguroSocial` debe descomponerse en un atributo de la clase `Persona` (vea la figura 7-30).

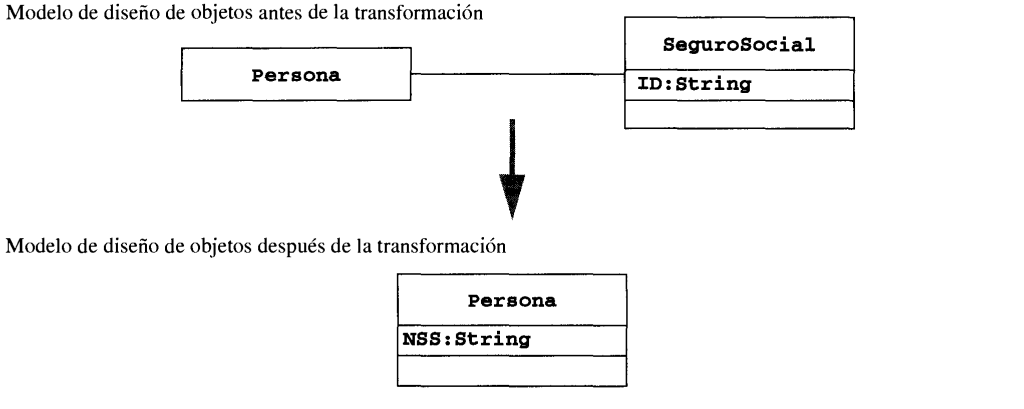


Figura 7-30 Representaciones alternas de un identificador único para una `Persona` (diagramas de clase UML).

La decisión de descomponer clases no siempre es obvia. En el caso de un sistema de seguridad social, la clase `SeguroSocial` puede tener mucho más comportamiento, como rutinas especializadas para la generación de nuevos números basados en fechas de nacimiento y la ubicación de la inscripción original. En general, los desarrolladores deberán retrasar las decisiones de descomposición hasta el inicio de la implementación cuando se tienen claras las responsabilidades de cada clase.

7.4.13 Cacheo de resultados de cálculos costosos

Con frecuencia los cálculos costosos sólo necesitan realizarse una vez, debido a que los valores con los que se realiza el cálculo no cambian o lo hacen despacio. En tales casos se puede cachear el resultado del cálculo como un atributo privado. Considere, por ejemplo, la operación `Capa.obtenerContorno()`. Supongamos que todos los `ElementoCapa` se definen una sola vez como parte de la configuración del sistema y no cambian durante la ejecución. Entonces, el vector de `Punto` regresado por la operación `Capa.obtenerContorno()` siempre es el mismo para un `cuadroLim` y `detalle` dados. Además, los usuarios finales tienen la tendencia a enfocarse en una cantidad limitada de puntos alrededor del mapa conforme se enfocan en una ciudad o región específica. Tomando en cuenta estas observaciones, una optimización simple es añadir un atributo privado `puntosCacheados` a la clase `Capa`, el cual recuerda el resultado de la operación `obtenerContorno()` para un par dado de `cuadroLim` y `detalle`. Luego la operación `obtenerContorno()` revisa primero el atributo `puntosCacheados` y si lo encuentra regresa el `Vector Punto` correspondiente, y en caso contrario llama a la operación `obtenerContorno()` sobre cada `ElementoCapa` contenido. Observe que este enfoque incluye un intercambio: por un lado mejoramos el tiempo de respuesta para la operación `obtenerContorno()`, pero por el otro consumimos espacio de memoria guardando información redundante.

7.4.14 Retraso de cálculos costosos

Un enfoque alterno para los cálculos costosos es retrasarlos lo más posible. Por ejemplo, considere un objeto que representa a una imagen guardada como un archivo. La carga desde el archivo de todos los pixeles que constituyen la imagen es costosa. Sin embargo, los datos de la imagen no necesitan cargarse sino hasta que se vaya a desplegar la imagen. Podemos realizar esta optimización usando un **patrón Apoderado** [Gamma *et al.*, 1994]. Un objeto `ApoderadoImagen` toma el lugar de `Imagen` y proporciona la misma interfaz que el objeto `Imagen` (figura 7-31). Las operaciones simples (como `anchura()` y `altura()`) son manejadas por `ApoderadoImagen`. Sin embargo, cuando se necesita trazar la `Imagen`, `ApoderadoImagen` carga los datos desde el disco y crea un objeto `ImagenReal`. Si el cliente no llama a la operación `trazar()` no se crea el objeto `ImagenReal`, ahorrando, por tanto, mucho tiempo de cálculos. Las clases llamadoras sólo acceden al `ApoderadoImagen` y a la `ImagenReal` mediante la interfaz de `Imagen`.

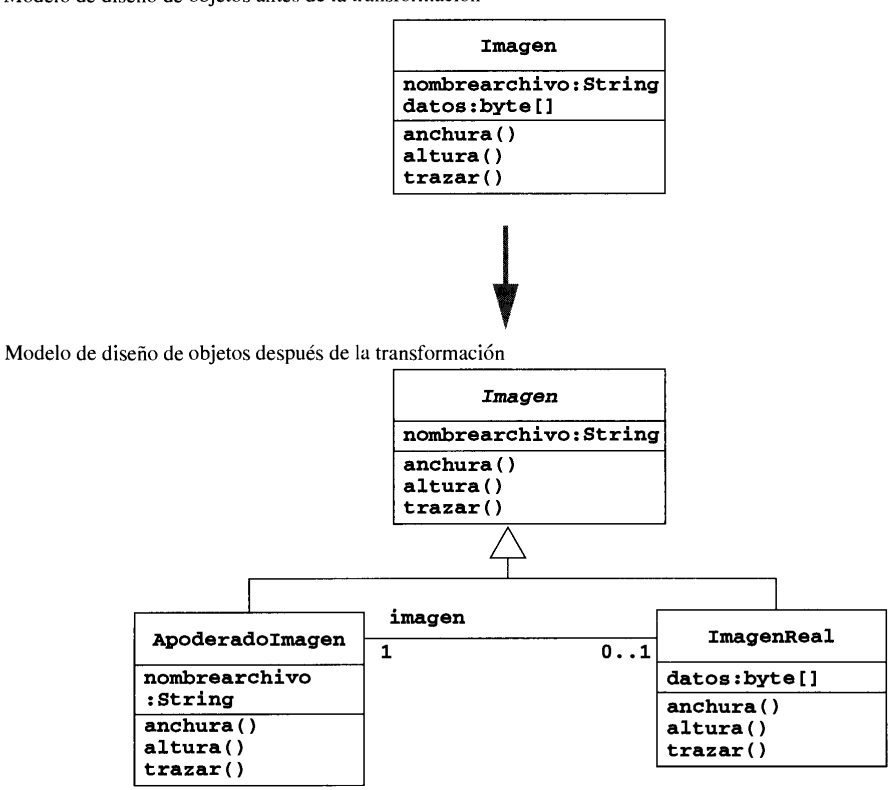


Figura 7-31 Retraso de cálculos costosos usando un patrón `Apoderado` (diagrama de clase UML).

7.5 Administración del diseño de objetos

En esta sección tratamos los asuntos de administración relacionados con el diseño de objetos. Hay dos retos administrativos principales durante el diseño de objetos:

- *Incremento en la complejidad de la comunicación.* La cantidad de participantes involucrados durante esta fase del desarrollo se incrementa en forma dramática. Los modelos de diseño de objetos y el código son el resultado de la colaboración de muchas personas. La administración necesita asegurarse que las decisiones entre los desarrolladores se tomen en forma consistente con los objetivos del proyecto.
- *Consistencia con decisiones y documentos anteriores.* Los desarrolladores a menudo no aprecian por completo las consecuencias de las decisiones del análisis y diseño del sistema antes del diseño de objetos. Cuando detallan y refinan el modelo de diseño de objetos los desarrolladores pueden cuestionarse algunas de estas decisiones y volverlas a evaluar a la luz de las lecciones aprendidas. El reto de la administración es mantener un registro de estas decisiones revisadas y asegurarse que todos los documentos reflejen el estado actual del desarrollo.

En la sección 7.5.1 tratamos el documento de diseño de objetos, su desarrollo y mantenimiento y su relación con otros documentos. En la sección 7.5.2 describimos los papeles asociados con el diseño de objetos.

7.5.1 Documentación del diseño de objetos

El diseño de objetos se documenta en el Documento de diseño de objetos (ODD, por sus siglas en inglés). Describe los compromisos del diseño de objetos tomados por los desarrolladores, los lineamientos que siguieron para las interfaces de subsistemas, la descomposición de subsistemas en paquetes y clases y las interfaces de clases. El ODD se usa para intercambiar la información de interfaz entre equipos y como una referencia durante las pruebas. La audiencia del ODD incluye a los arquitectos del sistema (es decir, los desarrolladores que participaron en el diseño del sistema), los desarrolladores que implementan cada subsistema y quienes hacen las pruebas.

El ODD permite que los desarrolladores comprendan el subsistema lo bastante bien como para que puedan usarlo. Además, una buena especificación de interfaz permite que otros desarrolladores implementen las clases en forma concurrente. En general, una especificación de interfaz debe satisfacer los siguientes criterios [Liskov, 1986]:

- *Restrictividad.* Una especificación debe ser lo suficientemente precisa para excluir implementaciones no deseadas. Las precondiciones y poscondiciones que especifican casos de frontera son una forma para lograr especificaciones restrictivas.
- *Generalidad.* Sin embargo, una especificación no debe restringir su implementación. Esto permite que los desarrolladores desarrollen y sustituyan implementaciones más eficientes o elegantes que tal vez no se pensaron cuando se especificó el subsistema.
- *Claridad.* Los desarrolladores deben comprender una especificación con facilidad y sin ambigüedades. Sin importar cuán restrictiva y general sea una especificación, es inútil si es difícil de comprender. Determinados comportamientos se describen con más facilidad en lenguaje natural, mientras que los casos de frontera pueden describirse con restricciones y excepciones.

Hay tres enfoques principales para documentar el diseño de objetos.

- *ODD autocontenido generado a partir del modelo.* El primer enfoque es documentar el modelo de diseño de objetos en la misma forma en que documentamos el modelo de análisis o el modelo de diseño del sistema: escribimos y mantenemos un modelo UML usando una herramienta y generamos el documento en forma automática. Este documento duplicará cualquier objeto de aplicación identificado durante el análisis. Las desventajas de esta solución incluyen la redundancia con el Documento de análisis de requerimientos (RAD, por sus siglas en inglés) y un alto nivel de esfuerzo para mantener la consistencia con el RAD. Además, el ODD duplica información en el código fuente y requiere un alto nivel de esfuerzo cada vez que cambia el código. Esto conduce a menudo a un RAD y a un ODD que no son precisos o que están caducos.
- *El ODD como extensión del RAD.* El segundo enfoque es tratar al modelo de diseño de objetos como una extensión del modelo de análisis. En otras palabras, se considera al

diseño de objetos como el conjunto de objetos de aplicación aumentado con los objetos de solución. La ventaja de esta solución es que se facilita el mantenimiento de la consistencia entre el RAD y el ODD a consecuencia de la reducción de la redundancia. Las desventajas de esta solución incluyen la contaminación del RAD con información que es irrelevante para el cliente y el usuario. Además, el diseño de objetos rara vez es tan simple como la identificación de objetos de solución adicionales. Con frecuencia los objetos de aplicación se cambian o transforman para acomodar objetivos de diseño o asuntos de eficiencia.

- *ODD incrustado en el código fuente.* El tercer enfoque es incrustar el ODD en el código fuente. Al igual que en el primer enfoque, primero representamos al ODD usando una herramienta de modelado (vea la figura 7-32). Una vez que el ODD llega a ser estable usamos la herramienta de modelado para generar los stubs de clase. Describimos cada interfaz de clase usando comentarios etiquetados que distinguen entre los comentarios del código fuente y las descripciones de diseño de objetos. Luego podemos generar el ODD usando una herramienta que analice el código fuente y extraiga la información relevante (por ejemplo, Javadoc [Javadoc, 1999a]). Una vez que el modelo de diseño de objetos está documentado en el código abandonamos el modelo de diseño de objetos original. La ventaja de este enfoque es que es más fácil mantener la consistencia entre el modelo de diseño de objetos y el código fuente: cuando se hacen cambios al código fuente es necesario actualizar los comentarios etiquetados y volver a generar el ODD. En esta sección examinamos este enfoque.

El asunto fundamental es el mantenimiento de la consistencia entre dos modelos y el código fuente. De manera ideal, queremos mantener el modelo de análisis, el modelo de diseño de objetos y el código fuente usando una sola herramienta. Entonces los objetos se describirían una sola vez y la consistencia entre la documentación, los stubs y el código se mantendría en forma automática.

Sin embargo, al momento presente las herramientas de modelado UML proporcionan facilidades para la generación de un documento a partir de un modelo o stubs de clase a partir de un modelo. La ayuda para la generación de documentación puede usarse, por ejemplo, para generar el RAD a partir del modelo de análisis (figura 7-32). La ayuda para la generación de stubs de clase (llamada ingeniería hacia delante) puede usarse en el enfoque de ODD autocontenido para generar las interfaces de clase y stubs para cada método.

Algunas herramientas de modelado proporcionan ayudas para la ingeniería inversa, esto es, volver a crear un modelo UML a partir del código fuente. Tales ayudas son útiles para la creación del modelo de objetos a partir de código heredado. Sin embargo, requieren mucho procesamiento manual, ya que la herramienta no puede volver a crear asociaciones bidireccionales basada sólo en los atributos de referencia.

En la actualidad el soporte de las herramientas se queda corto en el mantenimiento de dependencias de dos vías, en particular entre el modelo de análisis y el código fuente. Algunas herramientas, como Rationale Rose [Rational, 1998], tratan de realizar esta funcionalidad incrustando información acerca de las asociaciones y otras construcciones UML en los comentarios del código fuente. Aunque esto permita que la herramienta recupere los cambios sintácticos del código fuente, los desarrolladores todavía tienen que actualizar las descripciones del modelo para que reflejen los cambios. Debido a que los desarrolladores necesitan diferentes herramientas para cambiar el código fuente y el modelo, por lo general el modelo queda atrasado.

Hasta que las herramientas de modelado proporcionen un mejor soporte para el mantenimiento de la consistencia entre los modelos de objetos y el código fuente, encontramos que la

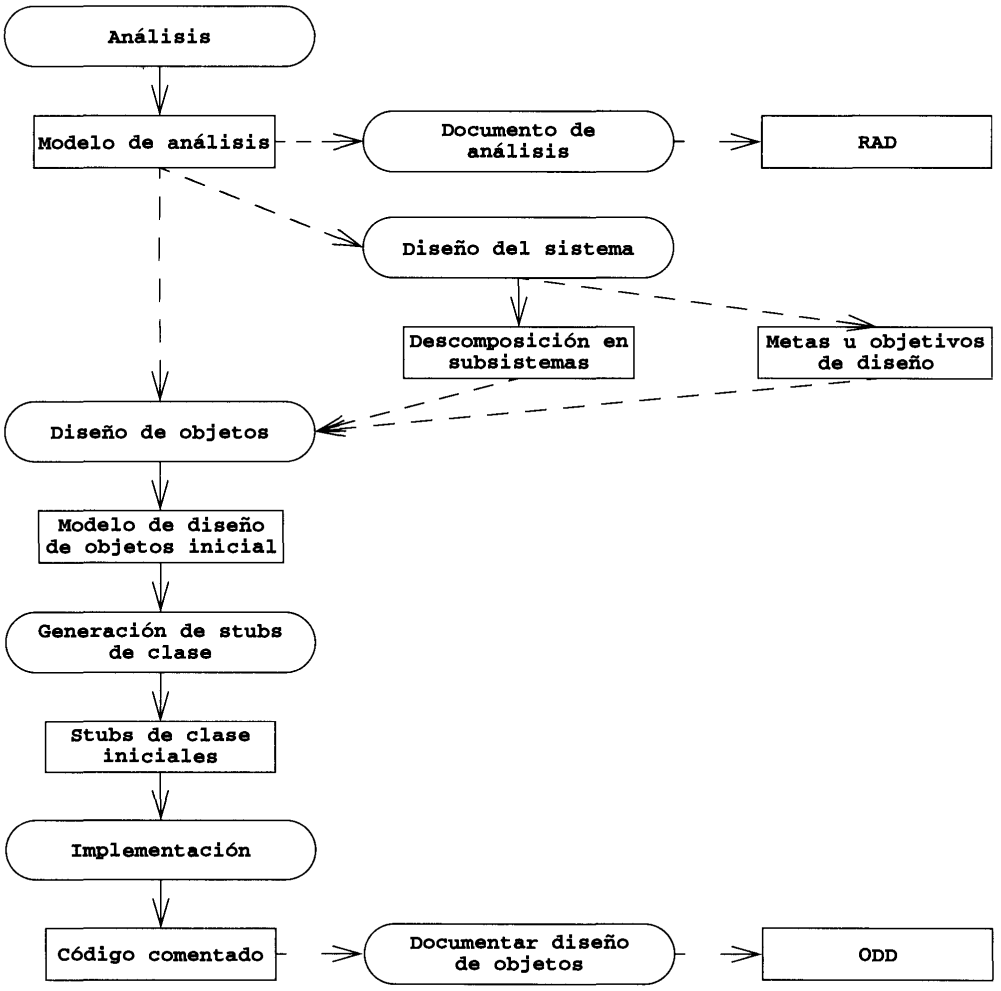


Figura 7-32 Enfoque ODD incrustado. Los stubs de clase se generan a partir del modelo de diseño de objetos. Luego el modelo de diseño de objetos se documenta como comentarios etiquetados en el código fuente. El modelo de diseño de objetos original se abandona y, en vez de ello, el ODD se genera a partir del código fuente usando una herramienta como Javadoc (diagrama de actividad UML).

generación del ODD a partir del código fuente y el enfoque del RAD sobre el dominio de la aplicación es lo más práctico. Reduce la cantidad de información redundante que es necesario mantener y ubica la información del diseño de objetos en donde está más accesible; esto es, en el código fuente. La consistencia entre el código fuente y el modelo de análisis todavía necesita mantenerse en forma manual. Sin embargo, esta tarea es más fácil, debido a que son menos los cambios del código que tienen un impacto en el modelo de análisis que los que tienen un impacto en el modelo de diseño de objetos.

La siguiente es una plantilla de ejemplo para un ODD generado:

Documento de diseño de objetos

1. Introducción
   1.1 Intercambios del diseño de objetos
   1.2 Lineamientos para la documentación de interfaces
   1.3 Definiciones, siglas y abreviaturas
   1.4 Referencias
2. Paquetes
3. Interfaces de clases

Glosario

La primera sección del ODD es una introducción del documento. Describe los compromisos generales de los desarrolladores (por ejemplo, comprar o construir, espacio de memoria frente a tiempo de respuesta), lineamientos y convenciones (por ejemplo, convenciones de denominación, casos de frontera, mecanismos para el manejo de excepciones) y un panorama del documento.

Los lineamientos para la documentación de interfaces y convenciones de codificación son el único factor más importante que puede mejorar la comunicación entre desarrolladores durante el diseño de objetos. Esto incluye una lista de reglas que deben seguir los desarrolladores cuando diseñan y nombran interfaces. A continuación se dan unos ejemplos de tales convenciones.

- A las clases se les nombra con nombres en singular.
- A los métodos se les nombra con frases verbales, y a los campos y parámetros con frases nominativas.
- El estado de error se regresa sólo mediante una excepción y no con un valor de retorno.
- Las colecciones y contenedores tienen un método `elementos()` que regresa una `Enumeration`.
- Las `Enumeration` regresadas por los métodos `elementos()` son robustas ante la eliminación de elementos.

Tales convenciones ayudan a que los desarrolladores diseñen interfaces de manera consistente, aunque muchos desarrolladores contribuyan a la especificación de la interfaz. Además, al hacer explícitas esas convenciones antes del diseño de objetos se facilita que los desarrolladores las sigan. En general, estas convenciones no deben evolucionar durante el proyecto.

La segunda sección del ODD, *Paquetes*, describe la descomposición de subsistemas en paquetes y la organización de archivos del código. Esto incluye un panorama de cada paquete, sus dependencias con otros paquetes y su uso esperado.

La tercera sección, *Interfaces de clases*, describe a las clases y sus interfaces públicas. Esto incluye un panorama de cada clase, sus dependencias con otras clases y paquetes, sus atributos públicos, sus operaciones y las excepciones que pueden presentar.

La versión inicial del ODD puede escribirse tan pronto como sea estable la descomposición en subsistemas. El ODD se actualiza cada vez que se dispone de nuevas interfaces o se revisan las existentes. Aunque un subsistema todavía no sea funcional, tener una interfaz en código fuente per-

mite que los desarrolladores codifiquen con mayor facilidad a los subsistemas dependientes y se comuniquen sin ambigüedades. Por lo general, en esta etapa los desarrolladores descubren parámetros faltantes y nuevos casos de frontera. El desarrollo del ODD es diferente con respecto a otros documentos, ya que están involucrados más participantes y el documento se revisa con mayor frecuencia. Para acomodar una alta tasa de cambio y muchos desarrolladores, pueden generarse las secciones 2 y 3 con una herramienta a partir de los comentarios del código fuente.

En Java esto puede hacerse con Javadoc, una herramienta que genera páginas Web a partir de comentarios de código fuente. Los desarrolladores comentan las interfaces y declaraciones de clase con comentarios etiquetados. Por ejemplo, la figura 7-33 muestra la especificación de interfaz para la clase `Capa` del ejemplo JEWEL. El comentario de encabezado en el archivo describe el propósito de la clase `Capa`, sus autores, su versión actual y referencias cruzadas hacia clases relacionadas. Javadoc utiliza las etiquetas `@see` para crear referencias cruzadas entre clases. Después del comentario de encabezado están las declaraciones de clase y de métodos. Cada comentario de método contiene una breve descripción del propósito del método, sus parámetros y el resultado que regresa. Cuando se usan restricciones también se incluyen las precondiciones y poscondiciones en el encabezado del método. La primera frase del comentario y los comentarios etiquetados son extraídos y formateados por Javadoc. El mantenimiento del material para el ODD con el código fuente permite que los desarrolladores mantengan consistencia de manera más fácil y rápida. Esto es esencial cuando están involucradas varias personas.

Para cualquier sistema de tamaño útil, el ODD representa una gran cantidad de información que puede traducirse en varios cientos o miles de páginas de documentación. Además, el ODD evoluciona con rapidez durante el diseño de objetos y su integración, conforme los desarrolladores comprenden mejor las necesidades de los demás subsistemas y encuentran fallas en sus especificaciones. Por estas razones, todas las versiones del ODD deben estar disponibles en forma electrónica, por ejemplo, como un conjunto de páginas Web. Además, los diferentes componentes del ODD deben ser puestos bajo la administración de la configuración y sincronizados con sus archivos de código fuente correspondientes. En el capítulo 10, *Administración de la configuración del software*, describimos con mayor detalle los problemas de la administración de la configuración.

7.5.2 Asignación de responsabilidades

El diseño de objetos se caracteriza por una gran cantidad de participantes que tienen acceso y modifican una gran cantidad de información. Para asegurar que los cambios a las interfaces se documenten y comuniquen en forma ordenada, varios papeles colaboran para controlar, comunicar e implementar los cambios. Estos incluyen a los miembros del equipo de arquitectura que son responsables del diseño del sistema y de las interfaces de subsistemas, a los coordinadores que son responsables de la comunicación entre equipos y a los gerentes de configuración que son responsables del seguimiento de los cambios.

A continuación se presentan los papeles principales del diseño de objetos.

- El **arquitecto principal** desarrolla los lineamientos y convenciones de codificación antes de que comience el diseño de objetos. Al igual que muchas convenciones, el conjunto actual de convenciones no es tan importante como el compromiso de todos los arquitectos y desarrolladores para usar esas convenciones. El arquitecto principal también es responsable de asegurar la consistencia con las decisiones anteriores documentadas en el SDD y el RAD.

```
/* La clase Capa es un contenedor de ElementoCapa, y cada uno
 * representa a un polígono o a una polilínea.
 * Por ejemplo, JEWEL tiene, por lo general, una capa de caminos,
 * una capa de agua, una capa política y una capa de emisiones.
 * @author John Smith
 * @version 0.1
 * @see ElementoCapa
 * @see Punto
 */
class Capa {

    /* Se omiten las variables miembro, los constructores y otros métodos */
    Enumeration elementos() {…};

    /* La operación obtenerContorno regresa una enumeration de puntos
     * que representan a los elementos de las capas en un nivel
     * de detalle especificado. La operación sólo regresa los puntos
     * contenidos dentro del rectángulo cuadroLim.
     * @param       cuadroLim   El rectángulo limitante en coordenadas universales
     * @param       detalle     El nivel de detalle (números grandes significa
                    más detalle)
     * @return      Una enumeration de puntos en coordenadas universales.
     * @throws      DetalleCero
     * @throws      CuadroLimitanteCero
     * @pre         detalle > 0.0 and cuadroLim.anchura > 0.0 and
                    cuadroLim.altura > 0.0
     * @post        forall ElementoCapa ec en this.elementos() |
     *                  forall Punto p en ec.puntos() |
     *                      resultado.contiene(p)
     */
    Enumeration obtenerContorno (Rectangle2D cuadroLim, double detalle) {…};

    /* Se omiten otros métodos */
}
```

Figura 7-33 Descripción de interfaz de la clase `Capa` usando comentarios etiquetados Javadoc (fragmento Java).

- Los **coordinadores de arquitectura** documentan las interfaces de los sistemas públicos de los que son responsables. Esto conduce a un primer borrador del ODD que utilizan los desarrolladores. Los coordinadores de arquitectura también negocian los cambios a las interfaces públicas cuando es necesario. A menudo, el asunto no es de consenso sino de comunicación: los desarrolladores que dependen de la interfaz pueden agradecer el cambio si se les notifica primero. Los coordinadores de arquitectura y el arquitecto principal forman el equipo de arquitectura.
- Los **diseñadores de objetos** refinan y detallan la especificación de la interfaz de la clase o subsistema que implementan.
- El **gerente de configuración** de un subsistema libera los cambios a las interfaces y al ODD una vez que se encuentran disponibles. El gerente de configuración también lleva cuenta de la relación entre el código fuente y las revisiones del ODD.

- Los **escritores técnicos** del equipo de documentación pulen la versión final del ODD. Ellos aseguran que el documento sea consistente desde un punto de vista estructural y de contenido. También revisan que se apegue a los lineamientos.

Al igual que en el diseño del sistema, el equipo de arquitectura es la fuerza integradora del diseño de objetos. El equipo de arquitectura asegura que los cambios sean consistentes con los objetivos del proyecto. El equipo de documentación, incluyendo a los escritores técnicos, asegura que los cambios sean consistentes con los lineamientos y convenciones.

7.6 Ejercicios

1. Considere las clases `PoliLínea`, `Polígono` y `Punto` de la figura 7-14. Escriba las siguientes restricciones en OCL:
   - Un `Polígono` está compuesto por una secuencia de al menos tres `Punto`.
   - Un `Polígono` está compuesto por una secuencia de `Punto` que comienza y termina en el mismo `Punto`.
   - Los `Punto` regresados por el método `obtenerPuntos(cuadroLim)` de un `Polígono` están dentro del rectángulo `cuadroLim`.
2. Considere las clases `Capa`, `ElementoCapa`, `PoliLínea` y `Punto` de la figura 7-15. Escriba en OCL las restricciones que se encuentran a continuación. Observe que las dos últimas restricciones requieren el uso del operador OCL `forAll` sobre las colecciones.
   - Un nivel de `detalle` no puede ser parte en forma simultánea de los conjuntos `enNivelesDetalle` y `noEnNivelesDetalle` de un `Punto`.
   - Para un nivel de `detalle` dado, `ElementoCapa.obtenerContorno()` sólo puede regresar `Punto` que contengan el nivel de detalle en su atributo de conjunto `enNivelesDetalle`.
   - Los conjuntos `enNivelesDetalle` y `noEnNivelesDetalle` sólo pueden crecer a consecuencia de `ElementoCapa.obtenerContorno()`. En otras palabras, una vez que un nivel de detalle está en alguno de esos conjuntos ya no puede eliminarse.
3. Considere la clase `Punto` de las figuras 7-14 y 7-15. Suponga que estamos evaluando un diseño alterno en el que un objeto global, llamado `TablaDetalle`, lleva cuenta de cuáles `Punto` han sido incluidos o excluidos de un nivel de detalle dado (en vez de que cada `Punto` tenga un atributo `enNivelesDetalle` y `noEnNivelesDetalle`). Esto se realiza mediante dos asociaciones entre `TablaDetalle` y `Punto`, las cuales están indexadas por `nivelDetalle` (vea la figura 7-34). Escriba restricciones OCL que especifiquen que, a un `nivelDetalle` dado, una `TablaDetalle` sólo puede tener un vínculo hacia un `Punto` dado (es decir, una `TablaDetalle` no puede tener en forma simultánea una asociación `incluyePunto` y otra `excluyePunto` para un `Punto` y `nivelDetalle` dados).
4. Usando las transformaciones descritas en las secciones 7.4.8 a 7.4.10, reestructure el modelo de diseño de objetos de la figura 7-34.
5. El cálculo en forma incremental de los atributos `enNivelesDetalle` y `noEnNivelesDetalle` de la clase `Punto` que se muestra en la figura 7-14 es una optimización. Usando los términos que presentamos en la sección 7.4, nombre el tipo de optimización que se realiza.

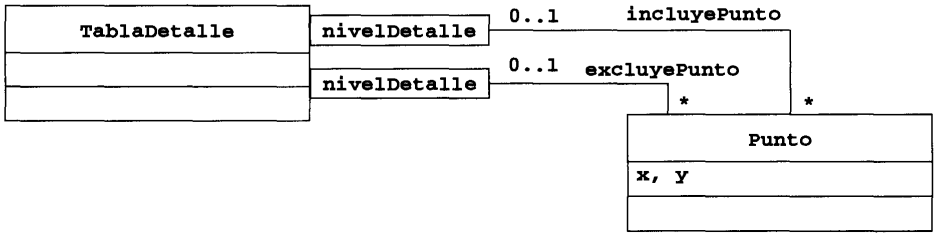


Figura 7-34 `TablaDetalle` es un objeto global que lleva cuenta de cuáles `Punto` han sido incluidos o excluidos de un `nivelDetalle` especificado. Ésta es una alternativa para los conjuntos `enNivelesDetalle` y `noEnNivelesDetalle` que se muestran en la figura 7-14.

6. Discuta las ventajas relativas de la clase `Punto` del ejercicio 3 contra la clase `Punto` de la figura 7-14 con respecto a los puntos de vista de tiempo de respuesta y espacio de memoria.
7. Suponga que está usando un marco de aplicación en vez del `GIS` que describimos en la sección 7.4. El marco de aplicación `GIS` proporciona un método `obtenerContorno()` que regresa una `Enumeration` de `PuntoGIS` para representar una `PoliLínea`. El método `drawPolyline()` de las JFC toma como parámetros dos arreglos de coordenadas y un entero que indica la cantidad de puntos de la `Polyline` (figura 7-18). ¿Cuál patrón de diseño usaría para resolver esta falta de concordancia de interfaces? Trace un diagrama de clase UML para ilustrar la solución.
8. Usted es el desarrollador responsable del método `obtenerContorno()` de la clase `Capa` de la figura 7-15. Encuentra que la versión actual de `obtenerContorno()` no excluye en forma adecuada a las `PoliLínea` que consisten de un solo `Punto` (a consecuencia del recorte). Usted repara el error. ¿A quiénes debe notificarlo?
9. Usted es el desarrollador responsable del método `obtenerContorno()` de la clase `Capa` de la figura 7-15. Cambia el método para representar `nivelDetalle` (que está guardado en `enNivelesDetalle` o `noEnNivelesDetalle`) usando un entero positivo en vez de un número de punto flotante. ¿A quiénes debe notificarlo?
10. ¿Por qué es difícil mantener la consistencia entre el modelo de análisis y el modelo de diseño de objetos? Ilustre su punto de vista con un cambio al modelo de diseño de objetos.

Referencias

[Birrer, 1993] E. T. Birrer, "Frameworks in the financial engineering domain: An experience report", *ECOOP'93 Proceedings*, Lecture Notes in Computer Science, No. 707, 1993.

[Bruegge *et al.*, 1993] B. Bruegge, T. Gottschalk, y B. Luo, "A framework for dynamic program analyzers", *OOPSLA' 93, (Object-Oriented Programming Systems, Languages, and Applications)*, págs. 65–82, Washington, DC, septiembre de 1993.

[Bruegge y Riedel, 1994] B. Bruegge y E. Riedel. "A geographic environmental modeling system: Towards an object-oriented framework", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP-94)*, Bologna, Italia, Lecture Notes in Computer Science, Springer Verlag, Berlin, julio de1994.

[Bruegge *et al.*, 1995] B. Bruegge, E. Riedel, G. McRae y T. Russel, "GEMS: An environmental modeling system", *IEEE Journal for Computational Science and Engineering*, págs. 55–68, septiembre de1995.

[Fayad y Hamu, 1997] M. E. Fayad y D. S. Hamu, "Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection", *The Communications of ACM*, 1997.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[Horn, 1992] B. Horn. "Constraint patterns as a basis for object-oriented programming", *Proceedings of the OOPSLA'92*, Vancouver, Canadá, 1992.

[Hueni *et al.*, 1995] H. Hueni, R. Johnson y R. Engel, "A framework for network protocol software", *Proceedings of OOPSLA*, Austin, TX, octubre de 1995.

[iContract] Java Design by Contract Tool, http://www.reliable-systems.com/tools/iContract/iContract.htm.

[Javadoc, 1999a] Sun Microsystems, Javadoc homepage, http://java.sun.com/products/jdk/javadoc/.

[Javadoc, 1999b] Sun Microsystems, "How to write doc comments for Javadoc", http://java.sun.com/products/jdk/javadoc/writingdoccomments.html.

[JFC, 1999] *Java Foundation Classes*, JDK Documentation. Javasoft, 1999.

[Johnson *et al.*, 1988] R. Johnson y B. Foote, "Designing reusable classes", *Journal of Object-Oriented Programming*, 1988, vol. 1, No. 5, págs. 22–35.

[Kompanek *et al.*, 1996] A. Kompanek, A. Houghton, H. Karatassos, A. Wetmore y B. Bruegge, "JEWEL: A distributed system for emissions modeling", *Conference for Air and Waste Management*, Nashville, TN, junio de 1996.

[Liskov, 1986] B. Liskov y J. Guttag, *Abstraction and Specification in Program Development.* McGraw-Hill, Nueva York, 1986.

[Meyer, 1997] Bertrand Meyer, *Object-Oriented Software Construction*, 2a ed. Prentice Hall, Upper Saddle River, 1997.

[OMG, 1995] Object Management Group, *The Common Object Request Broker: Architecture and Specification.* Wiley, Nueva York, 1995.

[OMG, 1998] Object Management Group, *OMG Unified Modeling Language Specification.* Framingham, MA, 1998, http://www.omg.org.

[Rational, 1998] Rational Software Corp., *Rationale Rose 98: Using Rose*. Cupertino, CA, 1998.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Schmidt, 1997] D. C. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software", *Handbook of Programming Languages*, Vol. 1, Peter Salus (ed.), MacMillan Computer, 1997.

[Szyperski, 1998] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press, Nueva York, Addison-Wesley, 1998.

[Weinand *et al.*, 1988] A. Weinand, E. Gamma y R. Marty. "ET++ – An object-oriented application framework in C++". In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, San Diego, CA, septiembre de 1988.

[Wilson y Ostrem, 1999] G. Wilson y J. Ostrem, *WebObjects Developer's Guide*, Apple, Cupertino, CA, 1998.

PARTE III
Manejo del cambio

8

8

Administración de la fundamentación

La descripción [de la motocicleta] debe cubrir el "qué" de la motocicleta desde el punto de vista de sus componentes, el "cómo" del motor desde el punto de vista de sus funciones. Se necesitaría con urgencia un análisis de "dónde" en forma de ilustración y también un análisis de "por qué" en forma de principios de ingeniería que dan lugar a esta conformación particular de partes.

—Robert Pirsig, *en Zen and the Art of Motorcycle Maintenance*

La fundamentación es la justificación de las decisiones. Los modelos que hemos descrito hasta ahora representan el sistema. Los modelos de fundamentación representan el razonamiento que conduce al sistema, incluyendo su funcionalidad y su implementación. La fundamentación es esencial en dos áreas: apoya la toma de decisiones y apoya la captura del conocimiento. La fundamentación incluye:

- Los asuntos que se trataron
- Las alternativas que se consideraron
- Las decisiones que se tomaron para resolver los asuntos
- Los criterios que se usaron para guiar las decisiones
- El debate que sostuvieron los desarrolladores para llegar a una decisión

En el contexto de la toma de decisiones, la fundamentación mejora la calidad de las decisiones haciendo explícitos los elementos de la decisión, como los criterios, prioridades y argumentos. En el contexto de la captura del conocimiento, la fundamentación es la información más importante del proceso de desarrollo cuando se hacen cambios al sistema. Por ejemplo, cuando se añade funcionalidad al sistema, la fundamentación permite que los desarrolladores tengan pistas sobre las decisiones que necesitan revisar y cuáles alternativas ya se han evaluado. Cuando se asigna nuevo personal al proyecto, los nuevos desarrolladores pueden familiarizarse con las decisiones anteriores teniendo acceso a la fundamentación del sistema.

Por desgracia, la fundamentación también es la información más compleja que generan los desarrolladores y, por tanto, la más difícil de mantener y actualizar. Además, la captura de las razones representa una inversión inicial con dividendos a largo plazo. En este capítulo describimos el modelado de problemas, una representación para la fundamentación del modelado. Luego describimos las actividades de creación, mantenimiento y acceso a los modelos de fundamentación. Concluimos el capítulo describiendo los problemas administrativos relacionados con el manejo de la fundamentación, como el apoyo para las decisiones y la negociación.

8.1 Introducción: un ejemplo del jamón

Los modelos del sistema son abstracciones de lo que hace éste. El modelo de análisis de requerimientos, incluyendo el modelo de casos de uso, el modelo de clases y los diagramas de secuencia (vea el capítulo 4, *Obtención de requerimientos*, y el capítulo 5, *Análisis*), representa el comportamiento del sistema desde el punto de vista del usuario. El modelo de diseño del sistema (vea el capítulo 6, *Diseño del sistema*) representa al sistema desde el punto de vista de subsistemas, objetivos de diseño, nodos de hardware, almacenes de datos, control de acceso, etcétera. El modelo de fundamentación representa la razón por la que un sistema dado está estructurado y se comporta en la forma en que lo hace.[1] ¿Por qué debemos capturar el *porqué*? Considere el siguiente ejemplo:[2]

> María le pregunta a Juan, su esposo, por qué siempre corta ambos extremos del jamón antes de ponerlo en el horno. Juan le responde que está siguiendo la receta de su madre y que siempre ha visto que le corta los extremos al jamón. En realidad nunca se había preguntado por qué lo hace, pero cree que es parte de la receta. María, intrigada por la respuesta, llama a su suegra para saber más acerca de esta receta de jamón.
>
> Ana, la madre de Juan, le proporciona más detalles sobre el corte del jamón, pero ninguna justificación culinaria: le dice que siempre le ha recortado casi una pulgada a cada extremo, como lo hacía su madre, suponiendo que tiene algo que ver para mejorar el sabor.
>
> María continúa su investigación y llama a Luisa, la abuela de Juan. Al principio Luisa se sorprende mucho: ella no le corta los extremos al jamón y no puede imaginarse en qué forma esta práctica mejoraría el sabor. Después de discutirlo mucho, Luisa recuerda al fin que, cuando Ana era una niñita, acostumbraba cocinar en una estufa más angosta a la que no le cabían los trozos de carne de tamaño estándar. Para resolver este problema acostumbraba cortar casi una pulgada de cada extremo del jamón. Dejó de hacerlo cuando tuvo una estufa más ancha.

Los desarrolladores y los cocineros son buenos para la diseminación de nuevas prácticas y técnicas. Sin embargo, por lo general se pierden las razones que hay tras esas prácticas, dificultando su mejora conforme cambia el contexto de aplicación. El error Y2K es un ejemplo de esto: en los años sesenta y setenta los costos de la memoria hicieron que los desarrolladores representaran la información en la forma más compacta posible. Por esta razón, con frecuencia se representaba al año con dos caracteres en vez de cuatro (por ejemplo, "1998" se representaba como "98"). La suposición de los desarrolladores fue que el software sólo se utilizaría durante unos cuantos años. Las operaciones aritméticas sobre los años asumían que todas las fechas eran del mismo siglo. Por desgracia para el software que realizaba esta aritmética con años de dos dígitos, esta abreviatura dejó de funcionar al final del siglo. Por ejemplo, al calcular la edad de una persona nacida en 1949, en 2001 se le considera con una edad de 01 – 49 = –48 años. La práctica de codificar años con dos dígitos se hizo estándar, aunque los precios de la memoria cayeran en

1. Desde una perspectiva histórica, gran parte de la investigación acerca de la fundamentación se ha enfocado en el diseño y, por tanto, en la literatura se usa con más frecuencia el término *fundamentación del diseño*. En vez de ello, usamos el término *fundamentación* para evitar confusiones y enfatizar que los modelos de fundamentación pueden usarse durante todas las fases del desarrollo.

2. Ejemplo inverosímil adaptado para este capítulo, se desconoce al autor original.

forma considerable y se aproximara el año 2000. Además, los nuevos sistemas necesitaban ser compatibles hacia atrás con los antiguos. Por estas razones, muchos sistemas distribuidos en fechas tan recientes como 1990 todavía tenían errores Y2K.

Los modelos de fundamentación permiten que los desarrolladores y los cocineros aborden el *cambio*, como estufas más grandes o memorias más baratas. La captura de la justificación de las decisiones modela en forma efectiva las dependencias entre las suposiciones iniciales y las decisiones. Cuando cambian las suposiciones se pueden revisar las decisiones. En este capítulo describimos técnicas para capturar, mantener y tener acceso a los modelos de fundamentación. En este capítulo:

- Proporcionamos un panorama general de las actividades relacionadas con los modelos de fundamentación (sección 8.2).
- Describimos el modelado de problemas, la técnica que usamos para representar la fundamentación (sección 8.3).
- Detallamos las actividades necesarias para la creación y acceso a los modelos de fundamentación (sección 8.4).
- Describimos los problemas administrativos relacionados con el mantenimiento de los modelos de fundamentación (sección 8.5).

Primero definamos el concepto de modelo de fundamentación.

8.2 Un panorama de la fundamentación

Una **fundamentación** es la motivación que hay detrás de una decisión. De manera más específica, incluye:

- **Problemas.** A cada decisión corresponde un problema que necesita ser resuelto para que continúe el desarrollo. Una parte importante de la fundamentación es una descripción del problema específico que se está resolviendo. Los problemas por lo general se formulan como preguntas: ¿Cómo debe cocinarse un jamón? ¿Cómo deben representarse los años?
- **Alternativas.** Las alternativas son soluciones posibles que pueden resolver el problema que se está considerando. Esto incluye alternativas que se exploraron y se descartaron porque no satisfacían uno o más criterios. Por ejemplo, comprar una estufa más grande cuesta demasiado; la representación de años con un número binario de 16 bits requiere demasiado procesamiento.
- **Criterios.** Los criterios son cualidades deseables que debe satisfacer la solución seleccionada. Por ejemplo, una receta para jamón debe ser realizable en equipo de cocina estándar. En los años sesenta los desarrolladores minimizaron el espacio de memoria. Durante el análisis de requerimientos, los criterios son requerimientos no funcionales y restricciones (por ejemplo, utililidad, cantidad de errores de entrada al día). Durante el diseño del sistema, los criterios son objetivos de diseño (por ejemplo, confiabilidad, tiempo de respuesta). Durante la administración del proyecto, los criterios son los objetivos y compromisos de administración (por ejemplo, entrega a tiempo frente a calidad).
- **Argumentación.** Las decisiones al cocinar y al desarrollar software no son algorítmicas. Los cocineros y los desarrolladores descubren problemas, prueban soluciones y argumentan sobre sus beneficios relativos. Sólo hasta después de muchas discusiones se llega a un consenso o se impone una decisión. Esta argumentación se da sobre todos los aspectos del proceso de decisión, incluyendo criterios, justificaciones, alternativas exploradas y compromisos.

- **Decisiones.** Una decisión es la resolución de un problema que representa la alternativa seleccionada de acuerdo al criterio que se usó para la evaluación y justificación de la selección. El corte de una pulgada a cada extremo del jamón y la representación de los años con dos dígitos son decisiones. Las decisiones ya están capturadas en los modelos del sistema que desarrollamos durante el análisis de requerimientos y el diseño del sistema. Además, muchas decisiones se toman sin explorar alternativas o sin examinar los problemas correspondientes.

Tomamos decisiones a todo lo largo del proceso de desarrollo y, por tanto, podemos usar los modelos de fundamentación durante cualquier actividad de desarrollo:

- Durante la *obtención de requerimientos* y *el análisis de requerimientos* tomamos decisiones acerca de la funcionalidad del sistema, a menudo junto con el cliente. Las decisiones están motivadas por las necesidades del usuario o de la organización. La justificación de esas decisiones es útil para la creación de casos de prueba durante la integración del sistema y la aceptación del usuario.
- Durante el *diseño del sistema* seleccionamos objetivos de diseño y diseñamos la descomposición en subsistemas. Por ejemplo, cuando identificamos objetivos de diseño con frecuencia basamos nuestra decisión en requerimientos no funcionales. La captura de la fundamentación de estas decisiones nos permite trazar las dependencias entre los objetivos de diseño y los requerimientos no funcionales. Esto nos permite revisar los objetivos de diseño cuando cambian los requerimientos.
- Durante la *administración del proyecto* hacemos suposiciones acerca de los riesgos relativos presentes en el proceso de desarrollo. Es más probable que comencemos las tareas de desarrollo relacionadas con un componente recién liberado en vez de hacerlo con uno maduro. La captura de la justificación que hay tras los riesgos y planes de reserva nos permite un mejor manejo cuando esos riesgos se convierten en problemas reales.
- Durante la *integración y las pruebas* descubrimos faltas de concordancia entre los subsistemas. Al tener acceso a la fundamentación de esos subsistemas a menudo podemos determinar cuál cambio o suposición introdujo la falta de concordancia y corregir la situación con un impacto mínimo en el resto del sistema.

El mantenimiento de la fundamentación es una inversión de recursos para manejar el cambio: capturamos información *ahora* para facilitar *después* la revisión de las decisiones cuando suceden los cambios. La cantidad de recursos que estemos dispuestos a invertir depende del tipo de proyecto.

Si estamos construyendo un sistema complejo para un solo cliente es muy probable que revisemos y mejoremos el sistema varias veces durante un largo tiempo. En este caso puede ser que el mismo cliente requiera que se registre la fundamentación. Si se está construyendo un prototipo conceptual para un nuevo producto, es muy probable que se deseche el prototipo una vez que se aprueba y comienza a realizar el desarrollo del producto. Si desviamos recursos de desarrollo para registrar la fundamentación corremos el riesgo de retrasar la demostración del prototipo y enfrentar la cancelación del proyecto. En este caso no registramos la fundamentación, debido a que el beneficio de esta inversión sería mínimo.

En términos más generales, distinguimos cuatro niveles de captura de la fundamentación:

- **Captura de fundamentación no explícita.** Los recursos se gastan sólo en el desarrollo. La documentación se enfoca sólo en los modelos del sistema. La información de la fundamentación está presente únicamente en la memoria de los desarrolladores y en los registros de comunicación, como los mensajes de correo electrónico, memorandos y faxes.
- **Reconstrucción de la fundamentación.** Se gastan recursos en la recuperación de la fundamentación del diseño durante el esfuerzo de documentación. El criterio de diseño y la motivación que está detrás de las decisiones arquitectónicas principales se integra con los modelos del sistema correspondientes. Las alternativas y argumentaciones descartadas no se capturan en forma explícita.
- **Captura de fundamentación.** Se hace un gran esfuerzo en la captura de la fundamentación conforme se toman las decisiones. La información sobre la fundamentación se documenta como un modelo separado y con referencias cruzadas hacia los demás documentos. Por ejemplo, la motivación para el modelo de análisis de requerimientos se captura en el Documento de fundamentación del análisis de requerimientos (RARD, por sus siglas en inglés), complementando al Documento de análisis de requerimientos (RAD, por sus siglas en inglés). En forma similar, la motivación para el diseño del sistema se captura en el Documento de fundamentación del diseño del sistema (SDRD, por sus siglas en inglés).
- **Integración de la fundamentación.** El modelo de fundamentación llega a ser el modelo central que usan los desarrolladores. La fundamentación producida durante diferentes fases se integra en una base de información viva y consultable. Los cambios al sistema de información suceden primero en la base de información como una discusión seguida por una o más decisiones. Los modelos del sistema representan la suma de las decisiones capturadas en la base de información.

En los dos primeros niveles de captura de la fundamentación, *Captura de fundamentación no explícita* y *Reconstrucción de la fundamentación*, nos apoyamos en la memoria de los desarrolladores para capturar y almacenar la fundamentación. En los dos últimos niveles, *Captura de fundamentación* e *Integración de la fundamentación*, invertimos recursos para la construcción de una memoria corporativa que es independiente de los desarrolladores. El compromiso entre estos dos extremos es la inversión de recursos durante las primeras fases del desarrollo. En este capítulo nos enfocamos en los dos últimos niveles de captura de fundamentación.

Además de los beneficios a largo plazo, el mantenimiento de la fundamentación también tiene efectos positivos a corto plazo: hacer explícita la fundamentación de una decisión nos permite comprender mejor el criterio que siguen los demás. También nos motiva a tomar decisiones racionales en vez de emocionales. Cuando menos, nos ayuda a distinguir cuáles decisiones se evaluaron con cuidado y cuáles se tomaron bajo presión y a la carrera.

Los modelos de fundamentación representan un cuerpo de información más grande y que cambia con mayor rapidez que los modelos del sistema. Esto introduce problemas relacionados con la complejidad y el cambio, como hemos visto antes. Por tanto, podemos aplicar las mismas técnicas de modelado para manejar la complejidad y el cambio. Luego describimos la manera en que representamos las razones con modelos de asuntos.

8.3 Conceptos de la fundamentación

En esta sección describimos los modelos de problemas, la representación que usamos para la fundamentación. El modelado de problemas se basa en la suposición de que el diseño sucede como una actividad dialéctica durante la cual los desarrolladores resuelven un problema argumentando las ventajas y desventajas de diferentes alternativas. Luego podemos capturar la fundamentación modelando el argumento que conduce a las decisiones de desarrollo. Representamos:

- Una pregunta o problema de diseño como un nodo de problema (sección 8.3.2).
- Soluciones alternas al problema como nodos de propuesta (sección 8.3.3).
- Ventajas y desventajas de diferentes alternativas usando nodos de argumento (sección 8.3.4).
- Decisiones que tomamos para resolver un problema como un nodo de resolución (sección 8.3.5).

En la sección 8.3.7 analizamos varias representaciones de problemas que tienen trascendencia histórica. Pero primero hablemos del control de tráfico centralizado, el dominio para los ejemplos de este capítulo.

8.3.1 Control de tráfico centralizado

Los sistemas de control de tráfico centralizados (CTC, por sus siglas en inglés) permiten que los despachadores de trenes supervisen y dirijan los trenes desde lejos. Las vías de tren están divididas en circuitos de vía contiguos que representan la unidad más pequeña que puede supervisar un despachador. Las señales y otros dispositivos aseguran que, a lo sumo, sólo un tren pueda ocupar un circuito de vía en cualquier momento. Cuando un tren entra a un circuito de vía, un sensor detecta su presencia y aparece la identificación del tren en el monitor del despachador. El despachador opera cambios de vía para dirigir los trenes. El sistema permite que un despachador planee una ruta completa alineando una secuencia de cambios de vía en la posición correspondiente. Al conjunto de circuitos de vía controlados por un solo despachador se le llama sección de vía.

La figura 8-1 es una versión simplificada de una interfaz de usuario de un CTC. Los circuitos de vía están representados por líneas. Los cambios de vía están representados por la intersección de tres líneas. Las señales están representadas por iconos que indican si una señal está abierta (es decir, permitiendo el paso del tren) o cerrada (es decir, impidiendo el paso del tren). Los cambios de vía, trenes y señales están numerados para su referencia en los comandos emitidos por el despachador. En la figura 8-1 las señales están numeradas `S1` a `S4`, los cambios de vía están numerados `CV1` y `CV2`, y los trenes están numerados `T1291` y `T1515`. Las computadoras que están cerca de la vía, llamadas estaciones del camino, aseguran que el estado de un grupo de cambios de vía y señales no presenten un riesgo para la seguridad. Por ejemplo, una estación del camino que controla los dispositivos de la figura 8-1 asegura que las señales opuestas `S1` y `S2` no puedan estar abiertas al mismo tiempo. Las estaciones del camino están diseñadas de tal forma que el estado del dispositivo que controlan sea seguro en caso de falla. A tal equipo se le llama a prueba de fallas. Los sistemas CTC se comunican con las estaciones del camino para modificar el estado de las vías cuando despachan trenes. Los sistemas CTC son, por lo general, altamente funcionales, pero no necesitan ser a prueba de fallas, tomando en cuenta que la seguridad de los trenes está garantizada por las estaciones del camino.

En los años sesenta los sistemas CTC tenían un tablero de desplegado personalizado que contenía focos que mostraban el estado de los circuitos de vía. Los cambios de vía y señales estaban controlados por medio de un tablero de entrada con muchos botones oprimibles e inte-

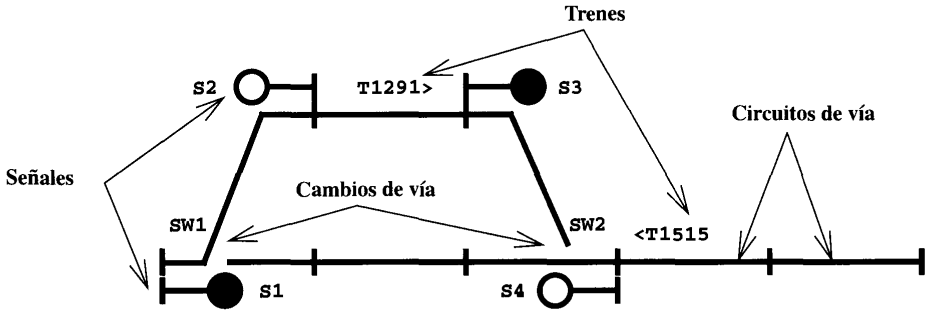


Figura 8-1 Un ejemplo de un desplegado de sección de vía CTC (simplificado para este ejemplo).

rruptores de dos posiciones. En los setenta los CRT reemplazaron a los tableros personalizados y proporcionaron a los despachadores información más detallada en menos espacio. En fechas más recientes se han introducido los sistemas de control de tráfico basados en estaciones de trabajo, proporcionando la posibilidad de una interfaz de usuario más refinada para los despachadores y la capacidad para distribuir el procesamiento entre varias computadoras.

Los sistemas de control de tráfico centralizados necesitan tener una disponibilidad muy alta. Aunque los sistemas de control de tráfico no son esenciales para la vida (la seguridad está asegurada por las estaciones del camino), una falla del sistema puede dar lugar a una gran desorganización del tráfico en las vías controladas, dando como resultado pérdidas económicas considerables. En consecuencia, la transición hacia nuevas tecnologías, como pasar de una mainframe a un ambiente de estaciones de trabajo o de una interfaz textual a una interfaz de usuario gráfica, necesita evaluarse con cuidado y hacerse mucho más despacio que para otros sistemas. El control de tráfico es un dominio en el cual la captura de la fundamentación es esencial y, por tanto, sirve como base para los ejemplos de este capítulo.

Tratemos a continuación la manera en que se usan los modelos de problemas para representar la fundamentación.

8.3.2 Definición de la cuestión: problemas

Un **problema** representa una dificultad concreta, como un requerimiento, un diseño o un inconveniente administrativo. *¿Qué tan pronto se le debe notificar a un despachador el retraso de un tren? ¿Cómo deben guardarse los datos persistentes? ¿Cuál tecnología presenta el mayor riesgo?* Con mucha frecuencia los problemas representan dificultades que no tienen una solución correcta única y que no pueden resolverse de manera algorítmica. Los problemas se resuelven, por lo general, mediante discusiones y negociaciones.

En UML representamos a los problemas con instancias de la clase `Problema`. Los `Problema` tienen un atributo `tema` que resume al problema, un atributo `descripción` que describe el problema con mayor detalle y hace referencia a material de apoyo, y un atributo `estado` que indica si el problema ha sido resuelto o no. El `estado` de un problema es **abierto** si no se ha resuelto y **cerrado** en caso contrario. Un problema cerrado puede volverse a abrir si se necesita revisarlo. Por convención damos un nombre corto a cada problema, como `¿retraso tren?:Problema` para hacer referencia a él. Por ejemplo, la figura 8-2 muestra los tres problemas que dimos como ejemplo en el párrafo anterior.

¿retraso tren?:Problema	¿Qué tan pronto debe notificarse al despachador el retraso de un tren?
¿almacenamiento? :Problema	¿Cómo deben guardarse los datos persistentes?
¿riesgo tecnología? :Problema	¿Qué tecnología presenta el mayor riesgo?

Figura 8-2 Un ejemplo de problemas (diagrama de objetos UML).

Los problemas que se plantean durante el desarrollo a menudo están relacionados. Por ejemplo, los problemas pueden descomponerse en **subproblemas** más pequeños. *¿Cuáles son los requerimientos de tiempo de respuesta en el sistema de control de tráfico?* incluye *¿Qué tan pronto se debe notificar a un despachador el retraso de un tren?* El desarrollo del sistema completo puede formularse como un solo problema: *¿qué sistema de control de tráfico debemos construir?*, que puede descomponerse en varios subproblemas. Los problemas también pueden plantearse por decisiones tomadas en otros problemas. Por ejemplo, la decisión de cachear datos en un nodo local plantea el problema del mantenimiento de la consistencia entre las copias de los datos centrales y los cacheados. A tales problemas se les llama **problemas consecuentes**.

Considere el sistema de control de tráfico centralizado que describimos antes. Supongamos que estamos examinando la transición de un sistema mainframe a un sistema basado en computadoras de escritorio. En el sistema futuro cada despachador tendrá una máquina de escritorio individual que se comunica con un servidor, el cual administra la comunicación con los dispositivos de campo. Durante la discusión del diseño se plantean dos problemas de interfaz: *¿cómo deben darse los comandos al sistema?* y *¿cómo deben mostrarse los circuitos de vía ante el despachador?* La figura 8-3 muestra estos dos problemas representados con un diagrama de objetos UML.

Un problema sólo debe enfocarse en la dificultad y no en las alternativas posibles para resolverlo. Una convención que favorece esto es redactar los problemas como preguntas. Para reforzar este concepto, también incluimos signos de interrogación en el nombre del problema. La información acerca de las alternativas posibles para resolver un problema se capturan mediante propuestas, las cuales trataremos a continuación.

¿entrada?:Problema	¿Cómo deben darse los comandos al sistema?
¿desplegado?:Problema	¿Cómo deben mostrarse los circuitos de vía ante el despachador?

Figura 8-3 Problemas de interfaz CTC (diagrama de objetos UML).

8.3.3 Exploración del espacio de solución: propuestas

Una **propuesta** representa una respuesta candidata para un problema. *No es necesario notificarle al despachador* es una propuesta para el asunto *¿qué tan pronto se le debe notificar a un despachador el retraso de un tren?* Una propuesta no necesita ser una respuesta válida o buena para el asunto que trata de resolver. Esto permite que los desarrolladores exploren en forma amplia el espacio de solución. Con frecuencia, en la lluvia de ideas la propuesta de una solución imperfecta hace que se generen nuevas ideas y soluciones que, tal vez, de otra forma no se hubieran pensado. Pueden traslaparse diferentes propuestas que tratan el mismo tema. Por ejemplo, las propuestas sobre el tema *¿cómo guardar los datos persistentes?* podrían incluir *Usar una base de datos relacional* y *Usar una base de datos relacional para los datos estructurados y archivos planos para las imágenes.* Las propuestas se usan para representar la solución al problema y también para las alternativas descartadas.

Una propuesta puede tratar uno o varios problemas. Por ejemplo, *Usar una arquitectura modelo/vista/despachador* puede tratar *¿Cómo separar los objetos de interfaz de los objetos de entidad?* y *¿Cómo mantener consistencia a través de varias vistas?* Las propuestas también pueden activar nuevos problemas. Por ejemplo, en respuesta al problema *¿Cómo minimizar las fugas de memoria?*, la propuesta *Usar recolección de basura* puede activar el problema consecuente *¿Cómo minimizar la degradación del tiempo de respuesta debida a la administración de la memoria?* Cuando tratamos un problema necesitamos asegurarnos que también se traten todos los problemas consecuentes asociados con las propuestas seleccionadas.

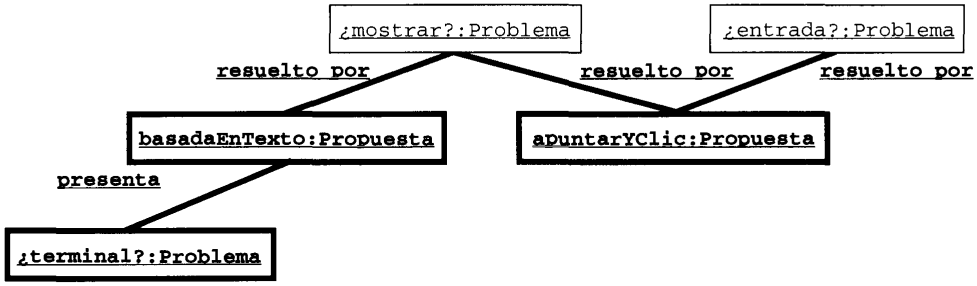
En UML representamos las propuestas como instancias de la clase `Propuesta`. Las `Propuesta`, al igual que los `Problema`, tienen un atributo `tema` y otro `descripción`. Por convención, a las propuestas les damos un nombre corto y las redactamos como enunciados que comienzan con un verbo. Las `Propuesta` están relacionadas con los `Problema` que resuelven con una asociación `resuelto por`. Los `Problema` están relacionados con las `Propuesta` que activan con una asociación `plantea`.

Cuando se discuten los problemas de interfaz, en nuestro sistema de control de tráfico centralizado consideramos dos propuestas, una interfaz `apuntarYClic`, que permite que los circuitos de vías se representen en forma gráfica, y una interfaz `basadaEnTexto` en la cual las secciones de vía se representan con caracteres especiales. La propuesta `basadaEnTexto` plantea un problema consecuente respecto a cuál emulación de terminal usar. La figura 8-4 muestra la adición de las dos propuestas y el problema consecuente.

Una propuesta sólo debe contener información relacionada con la solución, no sobre su valor, ventajas y desventajas. Los criterios y argumentos que describimos a continuación se usan para este propósito.

8.3.4 Evaluación del espacio de solución: criterios y argumentos

Un **criterio** es una cualidad deseable que deben tener las propuestas que resuelven un problema específico. Los objetivos de diseño, como el *tiempo de respuesta* o la *confiabilidad*, son criterios usados para resolver problemas de diseño. Los objetivos de administración, como el *costo mínimo* o *riesgo mínimo*, son criterios que se usan en la solución de problemas administrativos. Un conjunto de criterios indica las dimensiones en que necesita valorarse cada propuesta. Se dice que una propuesta que satisface un criterio se **valora positivamente** en ese criterio. En forma similar, se dice que una propuesta que no satisface un criterio se **valora negativamente** en ese criterio. Los criterios pueden compartirse entre varios problemas.



apuntarYClic:Propuesta	La interfaz para el despachador puede realizarse con una interfaz de apuntar y hacer clic.
basadaEnTexto:Propuesta	La pantalla usada por el despachador puede ser de sólo texto con caracteres gráficos para representar los segmentos de vía.
¿terminal?:Problema	¿Cuál emulación de terminal deberá usarse para el desplegado?

Figura 8-4 Un ejemplo de propuestas y problema consecuente (diagrama de objetos UML). Las `Propuesta` y el `Problema` consecuente se resaltan en negritas.

En UML representamos los criterios como instancias de la clase `Criterio`. Un `Criterio`, al igual que `Problema` y `Propuesta`, tiene un atributo `tema` y uno `descripción`. El atributo `tema` siempre se redacta en forma positiva; esto es, deberá establecer la cualidad que debe maximizar la propuesta. *Rápido*, *responsivo* y *barato* son buenos atributos de `tema`. *Costo* y *tiempo* no lo son. Un `Criterio` se asocia con `Propuesta` mediante asociaciones `valoración`. Las asociaciones valoración tienen un atributo de valor que indica si la valoración es positiva o negativa, y un atributo de peso que indica la fuerza de la propuesta con respecto al criterio. Por convención, añadimos el signo "$" al final del nombre del criterio para enfatizar que los criterios son mediciones de bondad y no deben confundirse con argumentos o asuntos.

Mientras evaluamos la interfaz de nuestro sistema de control de tráfico centralizado identificamos dos criterios: *disponibilidad*, que representa al requerimiento no funcional de maximizar el tiempo en que está disponible el sistema, y *usabilidad*, que representa (en este caso) el requerimiento no funcional de minimizar el tiempo para la emisión de comandos válidos (vea la figura 8-5). Estos criterios están tomados a partir de los requerimientos no funcionales del sistema. Valoramos ambas propuestas según estos criterios. Decidimos que la interfaz de apuntar y hacer clic tiene una valoración negativa en el criterio de disponibilidad, ya que al ser más compleja que la interfaz de texto presenta una mayor probabilidad de errores. Sin embargo, decidimos que la interfaz de apuntar y hacer clic es más utilizable que la interfaz textual, debido a una selección de comandos y entrada de datos más fáciles. Observe que el conjunto de asociaciones que vinculan las propuestas y el criterio en la figura 8-5 representan un compromiso: cada propuesta maximiza uno de los dos criterios, y el asunto es decidir cuál criterio tiene una prioridad más alta.

Un **argumento** es una opinión expresada por una persona que está de acuerdo o en desacuerdo con una propuesta, un criterio o una valoración. Los argumentos capturan el debate que lleva a la exploración del espacio de solución, define la bondad de las medidas y, a final de cuen-

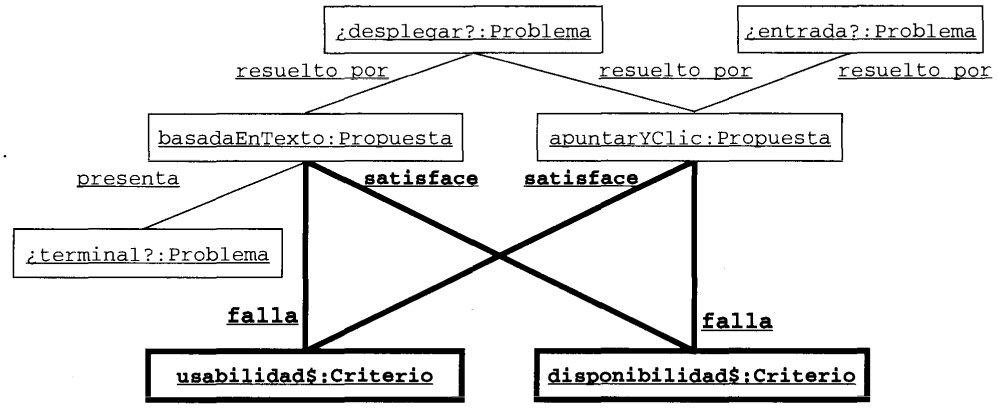


Figura 8-5 Un ejemplo de criterios y valoraciones (diagrama de objetos UML). Los `Criterio` se resaltan en negritas. Una valoración negativa está indicada por una asociación etiquetada `falla`, mientras que las valoraciones positivas están indicadas con una asociación etiquetada `satisface`.

tas, conduce a una decisión. En UML representamos los argumentos con instancias de la clase `Argumento`, que incluye los atributos `tema` y `descripción`. Los argumentos se relacionan con la entidad que deliberan con una asociación `apoyado por` o `se opone a`.

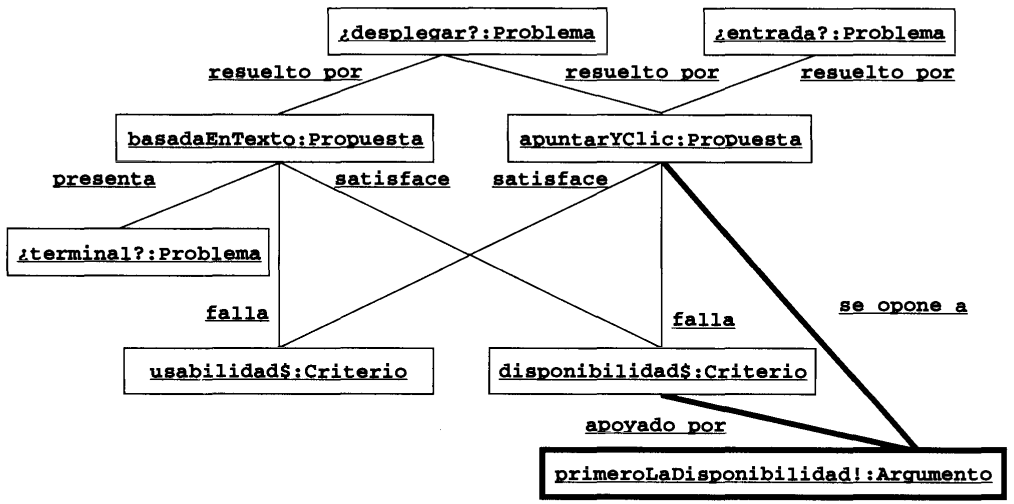
Mientras se discute la prioridad relativa de los criterios de disponibilidad y usabilidad, decidimos que cualquier beneficio del aspecto de usabilidad sería desplazado por una disponibilidad reducida del sistema. Capturamos esto creando un argumento que apoya el criterio de disponibilidad (vea la figura 8-6). Observe que un argumento puede apoyar a un nodo y oponerse a otro de manera simultánea.

Cuando seleccionamos criterios, valoramos propuestas y argumentamos acerca de ellas, evaluamos el espacio de diseño. El siguiente paso es usar esta evaluación para llegar al cierre y solucionar el problema.

8.3.5 Descomposición del espacio de solución: resoluciones

Una **resolución** representa la alternativa seleccionada para cerrar un problema. Una resolución representa una decisión que tiene impacto en uno de los modelos del sistema o en el modelo de tarea. Una resolución puede basarse en varias propuestas y resume la justificación que conduce a la decisión. En UML representamos las resoluciones con una instancia de la clase `Resolución`, que incluye atributos de `tema`, `descripción`, `justificación` y `estado`. Una `Resolución` puede relacionarse con `Propuestas` mediante asociaciones `basadaEn`. Una `Resolución` tiene exactamente una asociación `resuelve` hacia el `Problema` que resuelve.

El atributo `estado` de una `Resolución` indica si la `Resolución` todavía es relevante o no. Cuando la `Resolución` se vincula con su problema correspondiente su estado se establece como



primeroLaDisponibilidad! :Argumento

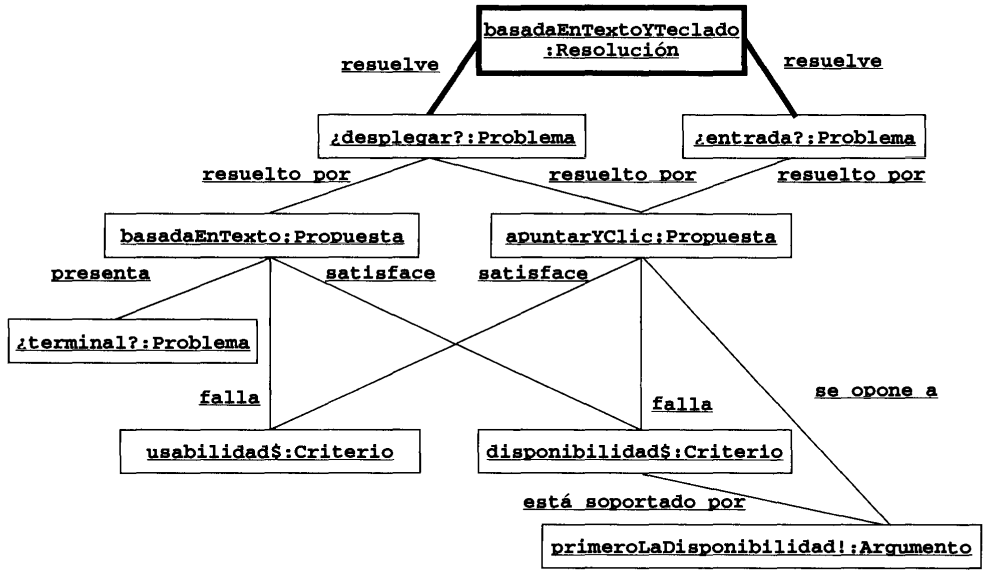
Es mucho más complejo implementar las interfaces de apuntar y hacer clic que las interfaces basadas en texto. También son más difíciles de probar, ya que la cantidad de acciones disponibles ante el despachador es mucho mayor. La interfaz de apuntar y hacer clic tiene el riesgo de introducir errores fatales en el sistema que anularían cualquier beneficio de usabilidad que pudiera proporcionar la interfaz.

Figura 8-6 Un ejemplo de argumento (diagrama de objetos UML). El `Argumento` se resalta en negritas.

`activo`, y el `estado` del `Problema` correspondiente se cambia a `cerrado`. Si se vuelve a abrir el `Problema`, el `estado` del `Problema` se cambia a `abierto` y el `estado` de la `Resolución` se cambia a `obsoleto`. Un `Problema` cerrado tiene exactamente una `Resolución` activa y cualquier cantidad de `Resolución` obsoletas.

Para finalizar el problema de la interfaz de control de tráfico, seleccionamos un desplegado basado en texto y una interfaz de teclado como base para la interfaz de usuario. Esta decisión está motivada por tratar al criterio de disponibilidad como más importante que el de usabilidad: una interfaz basada en texto dará como resultado código de interfaz de usuario más simple y más confiable a costa de un poco de usabilidad. El despachador no podrá ver tantos datos al mismo tiempo y no podrá dar comandos tan rápido como sería con una interfaz de apuntar y hacer clic. Creamos un nodo de resolución que contiene la justificación de la decisión y creamos vínculos entre la resolución y los dos problemas que resuelve (vea la figura 8-7).

La adición de una resolución a un modelo de problema concluye, efectivamente, la discusión del problema correspondiente. Ya que el desarrollo es iterativo, a veces es necesario volver a abrir un problema y volver a evaluar las alternativas competitivas. Sin embargo, al final del desarrollo la mayoría de los problemas deben estar cerrados, o listados en la documentación como problemas conocidos.



basadaEnTextoYTeclado:Resolución

Seleccionamos un desplegado basado en texto y una entrada por teclado para la interfaz de usuario de control de tráfico. La emulación de terminal debe proporcionar caracteres que permitan el trazado de circuitos de vía en modo texto.

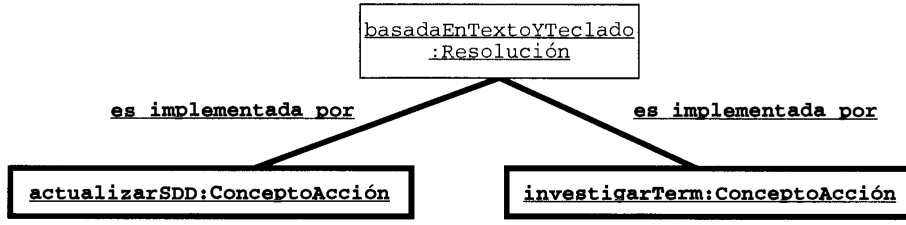
Esta decisión está motivada por la relativa simplicidad y confiabilidad de las interfaces basadas en texto, comparadas con las interfaces de apuntar y hacer clic. Estamos conscientes de que esta decisión tiene el costo de menos usabilidad, ya que se pueden presentar menos datos al despachador y la emisión de comandos será más lenta y más propensa a errores.

Figura 8-7 Un ejemplo de un problema cerrado (diagrama de objetos UML). La `Resolución` se resalta en negritas.

8.3.6 Implementación de resoluciones: conceptos de acción

Una resolución se implementa desde el punto de vista de uno o más **conceptos de acción**. A una persona se le asigna un concepto de acción, el cual es una tarea con una fecha de terminación. Los conceptos de acción no son parte de la fundamentación por sí mismos, sino que son parte del modelo de tarea (vea el capítulo 11, *Administración del proyecto*). Los conceptos de acción se describen aquí debido a que están muy integrados con el modelo de problemas.

En UML representamos un concepto de acción con una instancia de la clase `ConceptoAcción`. La clase `ConceptoAcción` tiene los atributos `tema`, `descripción`, `propietario`, `fechaTerminación` y `estado`. El `propietario` es la persona responsable de la terminación del `ConceptoAcción`. El `estado` de un `ConceptoAcción` puede ser `porHacer`, `noFactible`, `enProceso` o `realizado`. Una `Resolución` se asocia con los `ConceptoAcción` con un vínculo `esimplementadapor`. La figura 8-8 representa los `ConceptoAcción` generados después de la resolución de la figura 8-7.



actualizarSDD:ConceptoAcción	Para Alicia. Actualizar el SDD para que refleje la resolución basadaEnTextoYTeclado.
investigarTerm:ConceptoAcción	Para David. Investigar las diferentes emulaciones de terminal y sus ventajas para el desplegado de SecciónVía.

Figura 8-8 Un ejemplo de implementación de una resolución (diagrama de objetos UML). Los ConceptoAcción se resaltan en negritas.

La notación de problemas que describimos y su integración con el modelo de tarea es la notación de modelado que usamos para representar la fundamentación. En la literatura se han propuesto otros modelos de problemas para representar la fundamentación. Luego investigaremos en forma breve esos otros modelos.

8.3.7 Ejemplos de modelos y sistemas basados en problemas

La captura de la fundamentación fue propuesta originalmente por Kunz y Rittel. Desde entonces se han diseñado y evaluado diferentes modelos en el contexto de la ingeniería de software y otras disciplinas de la ingeniería. Aquí comparamos de modo breve tres de ellas, IBIS (siglas en inglés de sistema de información basado en problemas [Kunz y Rittel, 1970]), DRL (siglas en inglés de lenguaje de representación de decisiones [Lee, 1990]) y QOC (siglas en inglés de preguntas, opciones y criterios [MacLean *et al.*, 1991]).

Sistema de información basado en problemas

El IBIS incluye un modelo de problemas y un método de diseño para el tratamiento de problemas mal estructurados u horrorosos (en oposición a los dóciles). Un problema **horroroso** se define como un problema que no puede resolverse en forma algorítmica, sino que tiene que resolverse mediante discusión y debate.

El modelo de problemas IBIS (figura 8-9) tiene tres nodos (Problemas, Posiciones y Argumentos) relacionados por varios tipos de vínculos (apoya, se opone a, reemplaza, responde a, generaliza, cuestiona y sugiere). Cada Problema describe una dificultad de diseño bajo consideración. Los desarrolladores proponen soluciones al problema creando nodos Posición (similares a los nodos Propuesta que se describieron en la sección 8.3.3). Mientras se generan las alternativas los desarrolladores argumentan acerca de su valor con nodos Argumento. Los Argumento pueden apoyar una Posición u oponerse a una Posición. Observe que el mismo nodo puede aplicarse a varias posiciones. El modelo IBIS no incluyó originalmente Criterio ni Resolución.

IBIS fue soportado por una herramienta de hipertexto (gIBIS [Conklin y Burgess-Yakemovic, 1991]) y se usó para capturar la fundamentación durante reuniones frente a frente. Proporcionó las bases para la mayoría de los modelos de problemas subsiguientes, incluyendo DRL y QOC, que tratamos a continuación.

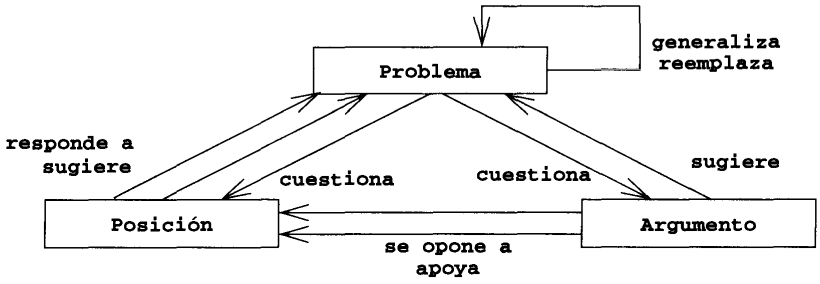


Figura 8-9 El modelo IBIS (diagrama de clase UML).

Lenguaje de representación de decisiones

El lenguaje de representación de decisiones DRL trata de capturar la **fundamentación de las decisiones** de un diseño [Lee, 1990]. La fundamentación de una decisión es definida por Lee como la representación de los elementos cualitativos de la toma de decisiones, incluyendo las alternativas que se han considerado, su evaluación, los argumentos que condujeron a esas evaluaciones y los criterios usados en esas evaluaciones. El DRL está apoyado por SYBIL, una herramienta que permite que el usuario lleve cuenta de las dependencias entre los elementos de la fundamentación cuando revisa las evaluaciones. El DRL aumenta el modelo IBIS original añadiendo nodos para capturar los `Objetivo de diseño` y los `Procedimiento`. El DRL ve la construcción de la fundamentación como una tarea comparable con el diseño del artefacto mismo. El DRL se resume en la figura 8-10. Las principales desventajas del DRL son su complejidad (siete tipos de nodos y 15 tipos de vínculos) y el esfuerzo empleado en la estructuración de la fundamentación capturada.

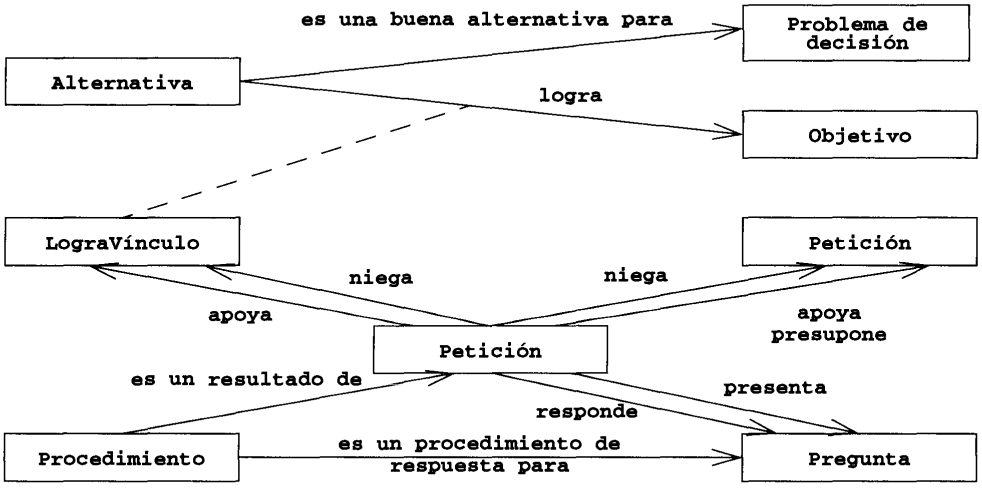


Figura 8-10 Lenguaje de representación de decisiones (diagrama de clase UML).

Preguntas, opciones y criterios (QOC)

Preguntas, opciones y criterios (QOC) es otro aumento al IBIS. Las `Pregunta` representan problemas de diseño a resolver (`Problema` en los modelos de problemas que presentamos). `Opción` son las respuestas posibles a las `Pregunta` (`Propuesta` en nuestro modelo). Las `Opción` pueden activar otras `Pregunta consecuente`. Las `Opción` se valoran en forma positiva y negativa respecto a `Criterio`, que son medidas de bondad relativas definidas por los desarrolladores. También, los `Argumento` pueden apoyar o retar cualquier `Pregunta`, `Opción`, `Criterio` o relación entre ellos. Los `Argumento` también pueden apoyar y retar a otro `Argumento`. La figura 8-11 muestra el modelo QOC.

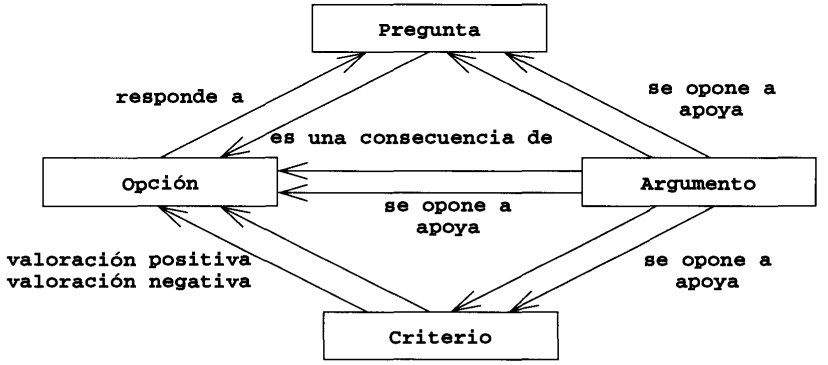


Figura 8-11 Modelo preguntas, opciones, criterios (diagrama de clase UML).

QOC e IBIS difieren en el nivel de proceso. Por un lado, el propósito de IBIS ha sido capturar la argumentación de diseño conforme sucede (por ejemplo, se uso gIBIS para capturar la información generada durante las reuniones de diseño). Por otro lado, las estructuras QOC se construyen como un acto de reflexión sobre el estado actual del diseño. Esta separación conceptual de las fases de construcción y argumentación del proceso de diseño enfatiza la elaboración y estructuración sistemática de la fundamentación, en vez de capturarlas como un efecto secundario de la deliberación. La fundamentación, desde la perspectiva del QOC, es una descripción del espacio de diseño explorado por los desarrolladores. Desde la perspectiva de IBIS, la fundamentación es un registro histórico del análisis que conduce a un diseño específico. En la práctica, ambos enfoques pueden aplicarse para capturar suficiente fundamentación. A continuación describimos las actividades relacionadas con la captura y mantenimiento de la fundamentación.

8.4 Actividades de la fundamentación: de los problemas a las decisiones

El mantenimiento de la fundamentación ayuda a los desarrolladores a manejar el cambio. Mediante la captura de la justificación de las decisiones pueden revisar con más facilidad las decisiones importantes cuando cambian los requerimientos del usuario o el ambiente de destino. Sin embargo, para que los modelos de fundamentación sean útiles necesitan estar capturados, estructurados y tenerse un acceso fácil a ellos. En esta sección describimos estas actividades, incluyendo:

- La captura de la fundamentación durante las reuniones de diseño (sección 8.4.2).
- La revisión de los modelos de fundamentación con aclaraciones subsiguientes (sección 8.4.3).
- La captura de fundamentación adicional durante las revisiones (sección 8.4.4).
- La reconstrucción de la fundamentación que no se capturó (sección 8.4.5).

La información más crítica sobre la fundamentación se genera durante el diseño del sistema: las decisiones que se toman durante el diseño del sistema pueden tener un impacto en cada uno de los subsistemas, y su revisión es costosa, en especial cuando se hace tardíamente en el proceso de diseño. Además, por lo general es compleja la fundamentación que hay tras la descomposición en subsistemas, ya que abarca muchos asuntos diferentes, como la asignación del hardware, el almacenamiento persistente, el control de acceso, el flujo de control global y las condiciones de frontera.

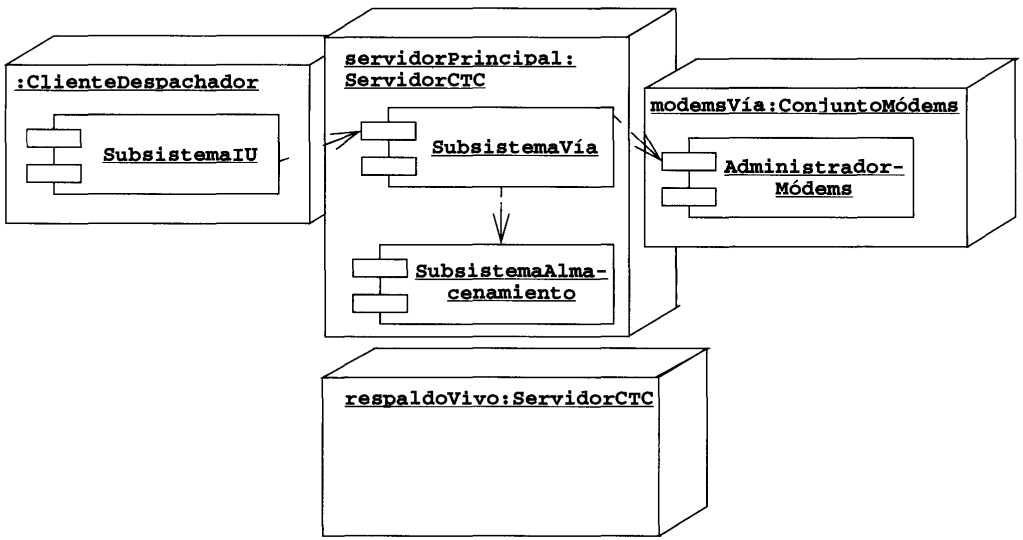
Por esta razón, en este capítulo nos enfocamos en el diseño del sistema. Sin embargo, observe que el mantenimiento de la fundamentación puede realizarse en forma similar a lo largo del desarrollo, desde la obtención de requerimientos hasta las pruebas en campo. Ilustramos las actividades de la fundamentación con problemas del diseño del sistema CTC, un sistema de control de tráfico centralizado para trenes de carga. A continuación describimos el modelo de diseño del sistema actual del CTC.

8.4.1 Diseño del sistema CTC

Considere el sistema CTC que describimos en la sección 8.3.1. Estamos en el proceso de reingeniería de un sistema heredado, reemplazando una computadora mainframe con una red de estaciones de trabajo. También estamos mejorando el sistema con cosas como la adición de control de acceso y un mayor enfoque en la seguridad y la usabilidad. Estamos en medio del diseño del sistema. Hasta ahora, a partir de los requerimientos no funcionales hemos identificado varios objetivos de diseño (ordenados en forma descendente de acuerdo con la prioridad):

- **Disponibilidad.** El sistema debe fallar menos de una vez al mes y recobrarse por completo en diez minutos después de una falla.
- **Seguridad.** Ninguna entidad que esté fuera del cuarto de control debe poder tener acceso al estado de las vías controladas o manipular ninguno de sus dispositivos.
- **Usabilidad.** Una vez que esté entrenado, un despachador no debe dar más de dos comandos erróneos al día.

Asignamos un nodo cliente por despachador. Dos nodos servidores redundantes mantienen el estado global del sistema (vea la figura 8-12). Los servidores también son responsables del almacenamiento persistente. Los datos se guardan en archivos planos que pueden copiarse fuera de línea e importarse hacia una base de datos para procesamiento fuera de línea. La comunicación con los dispositivos en las vías se realiza mediante módems manejados por una máquina dedicada. Un middleware soporta dos tipos de comunicaciones entre subsistemas: invocación de método para el manejo de peticiones y notificación de cambio de estado para informar a los subsistemas de los cambios de estado. Cada subsistema se suscribe a los eventos en que está interesado. En el presente, debemos tratar los problemas de control de acceso y definir los mecanismos que impiden que los despachadores manipulen el trabajo de los demás despachadores. En las secciones siguientes describimos la manera en que se debate y resuelve el asunto del control de acceso mientras se captura su fundamentación.



ClienteDespachador A cada despachador se le asigna un nodo `ClienteDespachador` que ejecuta la interfaz de usuario del sistema.

ServidorCTC Un `ServidorCTC` es responsable del mantenimiento del estado del sistema. Transmite comandos del despachador hacia los dispositivos de campo y recibe información de campo por medio del nodo `ConjuntoMódems`. Un `ServidorCTC` también es responsable del almacenamiento del estado persistente (por ejemplo, direcciones de dispositivos, nombres de dispositivos, las asignaciones de los despachadores, horarios de trenes). Se usan dos `ServidorCTC`, uno principal y un respaldo vivo, para incrementar la disponibilidad.

ConjuntoMódems El `ConjuntoMódems` administra los módems que se utilizan para la comunicación con los dispositivos de campo.

AdministradorMódems El `AdministradorMódems` es responsable de la conexión con los dispositivos de campo y de la transmisión de los comandos de campo.

SubsistemaAlmacenamiento El `SubsistemaAlmacenamiento` es responsable del mantenimiento del estado persistente.

SubsistemaVía El `SubsistemaVía` es responsable del mantenimiento del estado de las vías, conforme se reciben desde el campo las noticias de los cambios de estado, y de la emisión de comandos para los dispositivos mediante el `AdministradorMódems`, con base en los comandos en el nivel usuario recibidos del `SubsistemaIU`.

SubsistemaIU El `SubsistemaIU` es responsable de la recepción de comandos y del desplegado del estado de las vías ante el despachador. El `SubsistemaIU` controla la validez de los comandos del despachador antes de pasarlos al `ServidorCTC`.

Figura 8-12 Descomposición en subsistemas del CTC (diagrama de organización UML). El estado del sistema es mantenido por el `servidorPrincipal`. Un `respaldoVivo` del servidor principal está en espera por si falla el `servidorPrincipal`. El `servidorPrincipal` envía comandos y recibe transiciones de estado de las vías por medio del `conjuntoMódems`.

8.4.2 Captura de la fundamentación en las reuniones

Las reuniones permiten que los desarrolladores presenten, negocien y resuelvan problemas frente a frente. La presencia física de los desarrolladores respectivos involucrados en la discusión es importante al añadir los beneficios de la comunicación no verbal: permite que las personas valoren las posiciones relativas de cada cual y los compromisos que están dispuestos a aceptar. En forma alterna, es difícil la negociación y toma de decisiones mediante correo electrónico, por ejemplo, ya que pueden suceder tergiversaciones. Por tanto, las reuniones frente a frente son un punto de inicio natural para la captura de la fundamentación.

En el capítulo 3, *Comunicación de proyectos*, describimos procedimientos para la organización y captura de reuniones con minutas y agendas. Una agenda enviada antes de la reunión describe el estado y los puntos a tratar. La reunión se registra en minutas que se distribuyen poco después de la reunión. Usando los conceptos de modelado de problemas que describimos en la sección 8.3, escribimos una agenda desde el punto de vista de los *problemas* que necesitan discutirse y resolverse. Establecemos el objetivo de la reunión como llegar a una solución de estos problemas y cualquier subproblema relacionado que se plantee en la discusión. Estructuramos las minutas de la reunión en función de *propuestas* que se exploraron durante la reunión, *criterios* en los que se estuvo de acuerdo y *argumentos* que usamos para apoyar o rechazar propuestas. Capturamos las decisiones como *resoluciones* y *conceptos de acción* que establecen resoluciones. Durante la reunión revisamos el estado desde el punto de vista de los conceptos de acción que se produjeron en las reuniones anteriores.

Por ejemplo, considere el problema del control de acceso del sistema `CTC`. Necesitamos organizar una reunión del equipo de arquitectura, que incluye a los desarrolladores responsables del `SubsistemaIU`, el `SubsistemaSeguimiento` y el `ServicioNotificación`. Alicia, la moderadora del equipo de arquitectura, envía la agenda que se muestra en la figura 8-13.

Durante la reunión revisamos el concepto de acción (CA[1]: *investigar el modelo de control de acceso del middleware*) generado en la reunión de arquitectura anterior. El middleware proporciona bloques básicos para la autentificación y cifrado, pero no introduce ninguna otra restricción en el modelo de acceso. Los problemas `PR[1]` y `PR[2]` se resuelven con rapidez con conocimiento del dominio: un despachador puede ver todas las `SecciónVía`, pero sólo puede manipular los dispositivos de su `SecciónVía`. Sin embargo, el problema `PR[3]` (*¿cómo debe integrarse el control de acceso con las* `SecciónVía` *y* `ServicioNotificación`*?*) es más difícil e inicia un debate.

AGENDA: Integración del control de acceso y la notificación

Cuándo y dónde	**Papel**
Fecha: 13/09	**Moderador principal**: Alicia
Inicio: 4:30 p.m.	**Tomador de tiempo**: David
Terminación: 5:30 p.m.	**Secretario de actas**: Eduardo
Lugar: Edificio de trenes	**Salón**: 3420

1. Propósito

Se ha terminado la primera revisión de la correspondencia entre hardware y software y el diseño del almacenamiento persistente. Necesita definirse el modelo de control de acceso y su integración con los subsistemas actuales, como el `ServicioNotificación` y el `SubsistemaVía`.

2. Resultado deseado

Resolver los problemas acerca de la integración del control de acceso con la notificación.

3. Compartir la información [tiempo asignado: 15 minutos]

`CA[1]`: David: Investigar el modelo de control de acceso proporcionado por el middleware.

4. Discusión [tiempo asignado: 35 minutos]

`PR[1]`: ¿Un despachador puede ver las `SecciónVía` de los demás despachadores?

`PR[2]`: ¿Un despachador puede modificar las `SecciónVía` de otro despachador?

`PR[3]`: ¿Cómo debe integrarse el control de acceso con las `SecciónVía` y el `ServicioNotificación`?

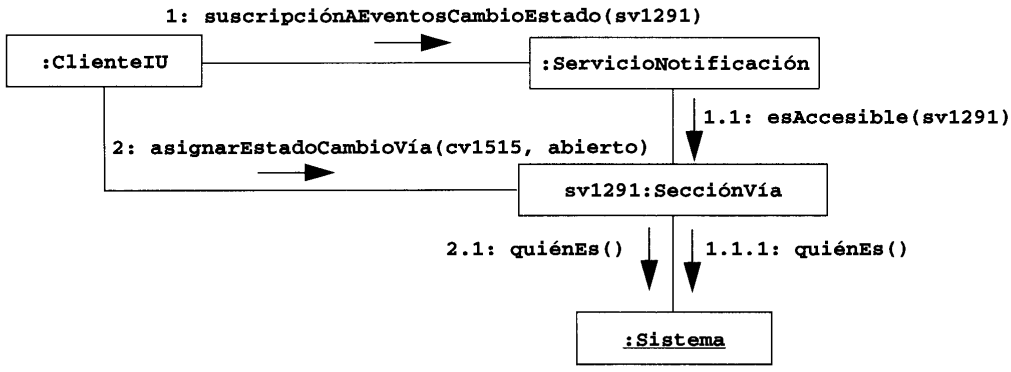
5. Cierre [tiempo asignado: 5 minutos]

Revisar y asignar nuevos conceptos de acción.

Crítica de la reunión.

Figura 8-13 Agenda para la discusión del control de acceso del CTC.

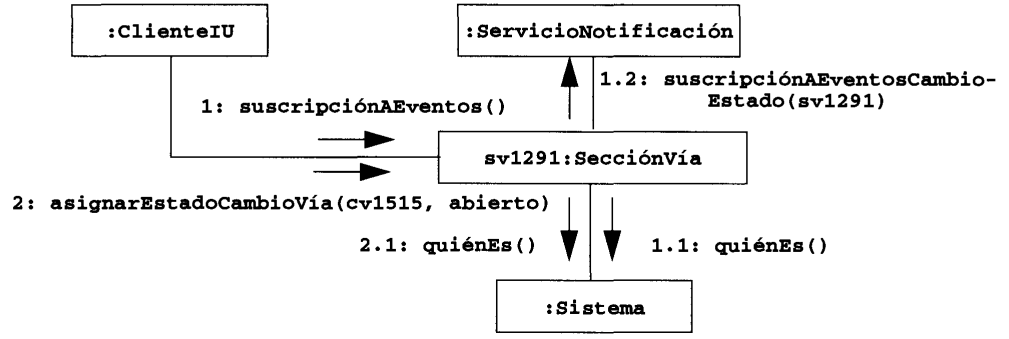
David, el desarrollador responsable del `ServicioNotificación` propone que se integre el control de acceso con la `SecciónVía` (vea la figura 8-14). La `SecciónVía` mantendría una lista de acceso de los `Despachador` que pueden examinar o modificar la `SecciónVía` dada. Los eventos también se organizarían por `SecciónVía`. Para que se le notifique acerca de los eventos en una `SecciónVía`, un subsistema necesitaría suscribirse a la `SecciónVía` del `ServicioNotificación`. El `ServicioNotificación` revisaría luego con la `SecciónVía` dada para ver si el despachador actual tiene, al menos, acceso de lectura.



ServicioNotificación	El `ServicioNotificación` transmite los cambios de estado de una `SecciónVía`. Para recibir noticias del `ServicioNotificación`, un subsistema necesita suscribirse a una `SecciónVía`. Sólo los subsistemas que tienen acceso a una `SecciónVía` dada pueden suscribirse a los eventos generados por una `SecciónVía`. El acceso está determinado por la llamada a la operación `esAccesible()` de la `SecciónVía`.
Sistema	El `Sistema` es responsable del seguimiento con seguridad de quién es el despachador actual con base en la información proporcionada por el `ClienteIU`. La `SecciónVía` examina las credenciales del despachador actual con la operación `quiénEs()`.
SecciónVía	Una `SecciónVía` consiste en un conjunto de `CircuitoVía` contiguos y sus `Dispositivo` asociados. El acceso se controla en el nivel de `SecciónVía` con una lista de acceso.
ClienteIU	El `ClienteIU` es responsable del desplegado de `SecciónVía` y de la entrada de comandos para cambiar el estado de `SecciónVía`.

Figura 8-14 Propuesta `P[1]`: el acceso es controlado por el objeto `SecciónVía` con una lista de acceso. El `ServicioNotificación` consulta a la `SecciónVía` para determinar si un subsistema puede recibir noticias acerca de una `SecciónVía` dada (diagrama de colaboración UML).

Alicia, la desarrolladora responsable del `SubsistemaVía`, que incluye la clase `SecciónVía`, propone que se invierta la dependencia entre `SecciónVía` y `ServicioNotificación` (vea la figura 8-15). En esta propuesta, el `ClienteIU` sólo podría interactuar con la clase `SecciónVía`, incluso cuando se suscribe a los eventos. El `ClienteIU` llamaría al método `SuscribirseAEventos()` de `SecciónVía`, el cual realizaría las revisiones de control de acceso y luego llamaría a `SuscribirseAEstadoCambioEventos()` en el `ServicioNotificación`. Entonces el `ClienteIU` no tendría acceso directo al `ServicioNotificación`. Esto tiene la ventaja de centralizar todas las operaciones protegidas en una clase y centralizar las revisiones de control de acceso. Además, en ese caso la `SecciónVía` podría desuscribir a los `ClienteIU` cuando se modifique la lista de acceso.

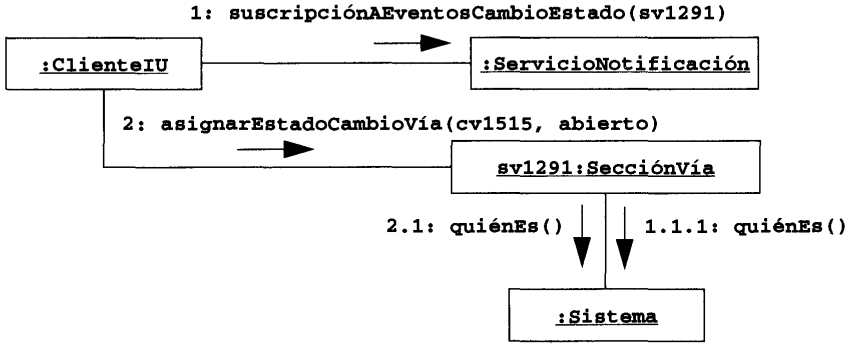


ServicioNotificación	El `ServicioNotificación` transmite los cambios de estado de una `SecciónVía`. Para recibir noticias del `ServicioNotificación`, un subsistema necesita suscribirse a una `SecciónVía`. *El ServicioNotificación sólo es accesible para la clase SecciónVía, la cual controla el acceso.*
SecciónVía	Una `SecciónVía` consiste de un conjunto de `CircuitoVía` contiguos y sus `Dispositivo` asociados. El acceso se controla en el nivel `SecciónVía`. *Para recibir noticias de cambios de estado, un subsistema necesita llamar a la operación suscribirAEventos() de la clase SecciónVía. La SecciónVía revisa el acceso antes de llamar al método suscripciónAEventosSecciónVía() del ServicioNotificación.*

Figura 8-15 Propuesta `P[2]`: El `ClienteIU` se suscribe a los eventos de la sección de vía mediante la operación `suscribirAEventos()` de `SecciónVía`. La `SecciónVía` revisa el acceso y luego llama a la operación `suscribirAEventosSecciónVía()` de `ServicioNotificación`. El `ServicioNotificación` no es accesible para la clase `ClienteIU` (diagrama de colaboración UML, las diferencias con respecto a la figura 8-14 se muestran en *cursivas*).

Eduardo observa que a cada uno de los despachadores se le permite ver las `SecciónVía` de los demás despachadores y, por tanto, sólo necesitan controlarse las modificaciones al estado. Suponiendo que todas las modificaciones se realizan por medio de invocaciones a método y que el `ServicioNotificación` sólo se usa para la difusión de los cambios, no es necesario que el `ServicioNotificación` esté integrado con el control de acceso. En este caso se podría usar un refinamiento de la propuesta inicial de David (vea la figura 8-16).

El equipo de arquitectura decide usar la propuesta de Eduardo debido a su simplicidad. Eduardo produce las minutas de reunión cronológicas que se muestran en la figura 8-17.



ServicioNotificación El ServicioNotificación transmite los cambios de estado de una SecciónVía. Para recibir noticias del ServicioNotificación, un subsistema necesita suscribirse a una SecciónVía. ~~*Sólo los subsistemas que tienen acceso a una SecciónVía dada pueden suscribirse a los eventos generados por una SecciónVía. El acceso está determinado por la llamada a la operación esAccesible() de la SecciónVía.*~~

Figura 8-16 Propuesta P[3]: El acceso a las operaciones que modifican a SecciónVía está controlado por el objeto SecciónVía con una lista de acceso. El ServicioNotificación no necesita ser parte del control de acceso, debido a que cada uno de los despachadores puede ver los cambios de estado (diagrama de colaboración UML, lo eliminado de la figura 8-14 se resalta con ~~*cursivas tachadas*~~).

MINUTA CRONOLÓGICA: Integración del control de acceso y la notificación

Cuándo y dónde	**Papel**
Fecha: 13/09	**Moderador principal**: Alicia
Inicio: 4:30 p.m.	**Tomador de tiempo**: David
Terminación: 6:00 p.m.	**Secretario de actas**: Eduardo
Lugar: Edificio de trenes	**Salón**: 3420

1. Propósito

Se ha terminado la primera revisión de la correspondencia entre hardware y software y el diseño del almacenamiento persistente. Necesita definirse el modelo de control de acceso y su integración con los subsistemas actuales, y en particular necesitan definirse el ServicioNotificación y el SubsistemaVía.

2. Resultado deseado

Resolver los asuntos acerca de la integración del control de acceso con la notificación.

3. Compartir la información

CA[1]: David: Investigar el modelo de control de acceso proporcionado por el middleware.

Estado: El middleware soporta una autentificación y cifrado fuertes. No introduce ninguna restricción en el modelo de acceso. Se puede implementar cualquier política de acceso en el lado servidor.

Figura 8-17 Minutas cronológicas de la discusión del control de acceso del CTC.

4. Discusión

PR[1]: ¿Un despachador puede ver las SecciónVía de los demás despachadores?

Zoe: Sí.

Eduardo: En la especificación del CTC.

PR[2]: ¿Un despachador puede modificar las SecciónVía de otro despachador?

Zoe: No. Sólo el despachador asignado a la SecciónVía puede manipular los dispositivos de la sección. Hay que tomar en cuenta que el despachador puede reasignarse en forma dinámica.

Eduardo: También en la especificación del CTC.

PR[3]: ¿Cómo debe integrarse el control de acceso con las SecciónVía y el ServicioNotificación?

David: La SecciónVía mantiene una lista de acceso. El ServicioNotificación pregunta a la SecciónVía quién tiene acceso.

Alicia: Es probable que debamos invertir la dependencia entre SecciónVía y ServicioNotificación. En vez de ello, el ClienteIU solicita la suscripción a la SecciónVía, y ésta revisa el acceso y luego llama al ServicioNotificación. De esta forma todos los métodos protegidos están en un solo lugar.

David: De esta forma la SecciónVía también puede cancelar la suscripción a los despachadores cuando se les quita el acceso.

Eduardo: Oye, no se necesita control de acceso en el ServicioNotificación: los despachadores pueden ver todas las SecciónVía. Mientras no se use el SistemaNotificación para cambiar el estado de SecciónVía no hay necesidad de restringir las suscripciones.

Alicia: Pero si se piensa en el control de acceso sobre la notificación sería más general.

Eduardo: Pero más complejo. Separemos simplemente el control de acceso y la notificación en este momento y revisemos el problema si cambian los requerimientos.

Alicia: Está bien. Me encargaré de revisar la API del SubsistemaVía.

5. Cierre

CA[2]: Alicia: Diseñar el control de acceso para el SubsistemaVía con base en la autentificación y cifrado proporcionados por el middleware.

Figura 8-17 Minutas cronológicas de la discusión del control de acceso del CTC.

Eduardo produce las minutas de la figura 8-17 insertando en la agenda la discusión que es relevante para los diferentes asuntos. Sin embargo, la discusión se registra como una lista cronológica de enunciados hechos por los participantes. La mayoía de estos enunciados mezcla la presentación de una alternativa con la argumentación en contra de otra alternativa. Para aclarar las minutas, Eduardo las reestructura después de la reunión con modelos de problemas (figura 8-18).

Los resultados importantes de la reunión de control de acceso son:

- Los Despachador pueden ver todas las SecciónVía, pero sólo modifican las que tienen asignadas.
- Se usa una lista de acceso asociada con las SecciónVía para el control de acceso.
- El ServicioNotificación no está integrado con el control de acceso, debido a que los cambios de estado pueden ser vistos por cualquier Despachador.

MINUTA ESTRUCTURADA: Integración del control de acceso y la notificación

Cuándo y dónde		**Papel**
Fecha:	13/09	**Moderador principal**: Alicia
Inicio:	4:30 p.m.	**Tomador de tiempo**: David
Terminación:	6:00 p.m.	**Secretario de actas**: Eduardo
Lugar:	Edificio de trenes	**Salón**: 3420

1. Propósito

Se ha terminado la primera revisión de la correspondencia entre hardware y software y el diseño del almacenamiento persistente. Necesita definirse el modelo de control de acceso y su integración con los subsistemas actuales, y en particular necesitan definirse el `ServicioNotificación` y el `SubsistemaVía`.

2. Resultado deseado

Resolver los problemas acerca de la integración del control de acceso con la notificación.

3. Compartir la información

`CA[1]`: David: Investigar el modelo de control de acceso proporcionado por el middleware.

Estado: El middleware soporta una autentificación y cifrado fuertes. No introduce ninguna restricción en el modelo de acceso. Puede implementarse cualquier política de acceso en el lado servidor.

4. Discusión

`PR[1]`: ¿Un despachador puede ver las `SecciónVía` de los demás despachadores?

`R[1]`: Sí (derivado de una especificación del CTC y confirmado por Zoe, una usuaria de pruebas).

`PR[2]`: ¿Un despachador puede modificar las `SecciónVía` de otro despachador?

`R[2]`: No. Sólo el despachador asignado a la `SecciónVía` puede manipular los dispositivos de la sección. Hay que tomar en cuenta que el despachador puede reasignarse en forma dinámica (derivado de una especificación del CTC y confirmado por Zoe).

`PR[3]`: ¿Cómo debe integrarse el control de acceso con las `SecciónVía` y el `ServicioNotificación`?

`P[3.1]`: La `SecciónVía` mantiene una lista de acceso de quienes pueden examinar o modificar el estado de la `SecciónVía`. Para suscribirse a los eventos un subsistema envía una petición al `ServicioNotificación`, el cual a su vez envía una petición a la `SecciónVía` para revisar el acceso.

`P[3.2]`: Las `SecciónVía` alojan todas las operaciones protegidas. El `ClienteIU` solicita la suscripción a los eventos de la `SecciónVía` enviando una petición a la `SecciónVía`, y ésta revisa el acceso y luego envía una petición al `ServicioNotificación`. `AR[3.1]` para `P[3.2]`: El control de acceso y las operaciones protegidas están centralizadas en una sola clase.

`P[3.3]`: No es necesario restringir el acceso a la suscripción de eventos. El `ClienteIU` solicita la suscripción de manera directa al `ServicioNotificación`. El `ServicioNotificación` no necesita revisar el acceso.

`AR[3.2]` para `P[3.3]`: Los despachadores pueden ver el estado de cualquier `SecciónVía` (vea `R[1]`).

`AR[3.3]` para `P[3.3]`: Simplicidad.

`R[3]`: `P[3.3]`. Vea el concepto de acción `CA[2]`.

5. Cierre

`CA[2]`: Alicia: Diseñar el control de acceso para el `SubsistemaVía` con base en la autentificación y cifrado proporcionados por el middleware y la resolución `R[3]` tratada en esta minuta.

Figura 8-18 Minutas estructuradas de la discusión de control de acceso de CTC.

Enfocándonos en el modelo de problemas, también hemos capturado que:

- Se investigó la integración del `ServicioNotificación` con el control de acceso.
- La centralización de todos los métodos protegidos en la clase `SecciónVía` fue un principio aceptable.

Estas dos últimas partes de la información es información de fundamentación y, por lo general, no se considera importante. Sin embargo, éste es el tipo de información que el secretario de actas captura y estructura para facilitar cambios futuros.

8.4.3 Captura asíncrona de la fundamentación

Las discusiones de las reuniones se apoyan en información del contexto. Cuando se inicia la reunión, la mayoría de los participantes ya tienen una cantidad considerable de información acerca del sistema, su propósito pretendido y su diseño. El moderador de la reunión se enfoca, por lo general, en un pequeño conjunto de problemas que necesitan resolverse. Por ejemplo, en la reunión que presentamos en la sección anterior, todos los participantes sabían el propósito y funcionalidad del sistema CTC, sus objetivos de diseño y la descomposición actual en subsistemas. La minuta de esta reunión sólo registra los problemas que están a discusión y, por tanto, no contiene mucha o ninguna información antecedente. Por desgracia, esta información se pierde con el tiempo y las minutas de reunión se hacen obsoletas con rapidez.

Podemos usar el modelado de problemas para resolver esta dificultad. En el capítulo 3, *Comunicación de proyectos*, describimos el uso de groupware, como grupos de noticias o Lotus Notes, para apoyar la comunicación asíncrona. Al integrar la preparación y el registro de la reunión con la comunicación asíncrona podemos capturar información contextual adicional.

En el ejemplo del CTC, supongamos que María, la desarrolladora responsable del `SubsistemaIU`, no pudo asistir a la reunión de control de acceso. Ella lee la agenda y la minuta de la reunión, las cuales se colocaron en el grupo de noticias dedicado al equipo de arquitectura. Aunque comprende el resultado de la reunión, la discusión acerca del `ServicioNotificación` requiere aclaraciones: el argumento `AR[3.2]` para la propuesta `P[3.3]` dice que, debido a que los `Despachador` pueden ver cada una de las `SecciónVía`, todos los eventos pueden ser visibles y, por tanto, no hay necesidad de controlar el acceso a los eventos. Esto implica que el `ServicioNotificación` se usa sólo para notificar los cambios de estado a los demás subsistemas. En otras palabras, `SecciónVía` no cambia su estado a consecuencia de los eventos generados por los demás subsistemas. María quiere confirmar que esta suposición es correcta y, en consecuencia, plantea un asunto al grupo de noticias (figura 8-19). También propone que no se permita que el `ServicioVía` se suscriba a cualquier evento para asegurar un control de acceso adecuado.

El seguimiento de las minutas de reunión permite que los desarrolladores capturen más del contexto que rodea al diseño. En consecuencia, se capturan más razones e información más clara. El uso del mismo modelo de problemas, tanto para las reuniones como para las discusiones en línea, nos permite integrar toda la información de la fundamentación. Aunque esto puede hacerse con una tecnología mínima, como los grupos de noticias, la representación del modelo de problemas, las agendas y minutas de reunión y los mensajes relacionados pueden integrarse en una

```
Grupo de noticias: ctc.arquitectura.discusión
Tema:                                                                Fecha:
PR[1]: ¿Un despachador puede ver las SecciónVía de los demás despachadores? 14/09
PR[2]: ¿Un despachador puede modificar las SecciónVía de otro despachador? 14/09
PR[3]: ¿Cómo debe implementarse el control de acceso con las SecciónVía
       y el ServicioNotificación?                                    14/09
       P[3.1]: Las SecciónVía mantienen una lista de acceso.         14/09
       P[3.2]: Las SecciónVía tienen operaciones de suscripción.     14/09
               +AR[3.1]:Extensibilidad.                              14/09
               +AR[3.2]:Centraliza todas las operaciones protegidas. 14/09
       P[3.3]: El ServicioNotificación no es parte del acceso.       14/09
               +AR[3.3]:Los despachadores pueden ver todas las SecciónVía. 14/09
               +AR[3.4]:Simplicidad.                                 14/09
---
De: María
Grupo de noticias: ctc.arquitectura.discusión
Tema: Problema consecuente: ¿No debe usarse la notificación para las peticiones?
Fecha: martes 15 de septiembre, 13:12:48 -0400

PR[4] respondiendo a AR[3.3]: para las listas de acceso frente a capacidades
> Los despachadores pueden ver todas las SecciónVía y, por tanto, deben poder
> ver todos los eventos.
Esto asume que la SecciónVía no depende de los eventos para cambiar su estado y
que los eventos sólo se usan para informar a los demás subsistemas acerca de los
cambios de estado. Para efectos de robustez, ¿debemos impedir que el ServicioVía
se suscriba a algún evento?
```

Figura 8-19 Ejemplo de un asunto consecuente enviado en forma asíncrona (envío a grupo de noticias). María, una desarrolladora que no asistió a la reunión, solicita aclaraciones. Esto conduce al envío de un asunto adicional y la captura de más razones.

herramienta groupware, como una base de datos Lotus Notes personalizada o una base de asuntos multiusuario alojada en un sitio Web (vea el ejemplo en la figura 8-20).

Una vez que se instituyen procedimientos para la organización y registro de la fundamentación en las reuniones, y se les expande con groupware, podemos capturar gran cantidad de la fundamentación. El siguiente reto es mantener actualizada esta información conforme suceden los cambios.

8.4.4 Captura de la fundamentación cuando se tratan los cambios

Los modelos de razones nos ayudan a manejar el cambio. Por desgracia, la fundamentación misma está sujeta a cambios cuando revisamos las decisiones. Cuando diseñamos una solución en respuesta a un cambio de requerimientos, por ejemplo, vemos la fundamentación anterior para valorar cuáles decisiones necesitan revisarse y diseñar un cambio. No sólo necesitamos capturar la fundamentación para el cambio y su solución, sino que también necesitamos relacionarlas con las razones anteriores.

Por ejemplo, supongamos que en el sistema CTC cambian los requerimientos del control de acceso. Antes se permitía que los `Despachador` vieran todas las `SecciónVía`. El cliente nos informa que, a menos que se especifique con anterioridad, los `Despachador` sólo podrán ver las `SecciónVía` colindantes. En respuesta a este cambio necesitamos modificar el diseño del control

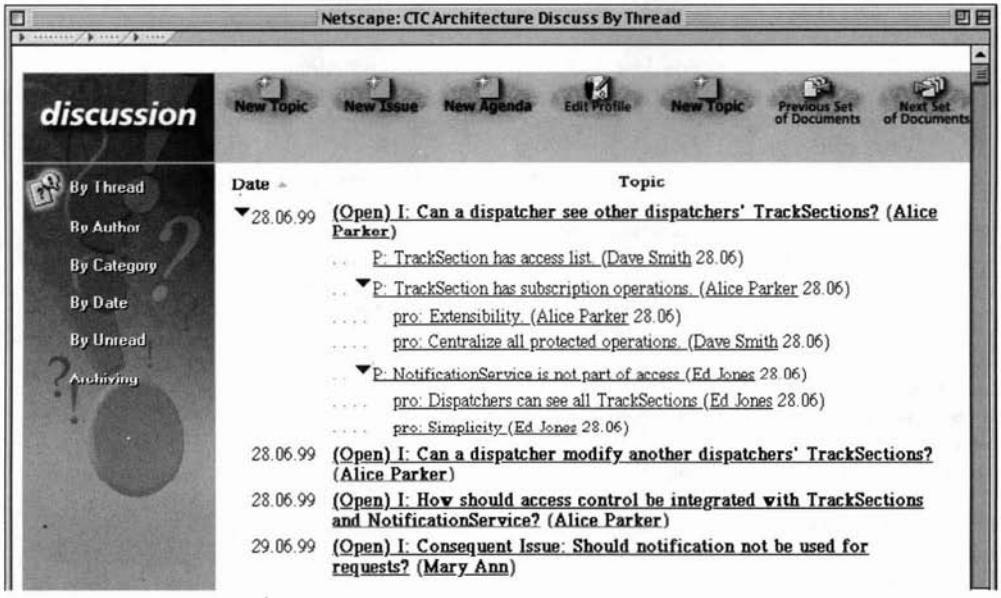


Figura 8-20 Un ejemplo de base de datos de problemas (plantilla de base de datos IBIS LN en Domino Lotus Notes). Los desarrolladores pueden tener acceso y colocar problemas, propuestas, argumentos y resoluciones con formularios Web.

de acceso y organizar una reunión con el equipo de arquitectura. En particular, necesitamos buscar la fundamentación anterior asociada con el control de acceso. Alicia, la moderadora principal del equipo de arquitectura, plantea la agenda que se muestra en la figura 8-21.

Durante la reunión David presenta la fundamentación tratada en reuniones anteriores y en el grupo de noticias de arquitectura. El equipo de arquitectura observa que ya no es válida la suposición de que todos los subsistemas pueden ver los eventos: al despachador sólo se le debe permitir que vea los eventos relacionados con las `SecciónVía` colindantes. La propuesta P[2, 14/09] (vea la figura 8-15) parece ser la mejor solución bajo los nuevos requerimientos, ya que todas las operaciones protegidas pueden centralizarse en la clase `SecciónVía`. Por desgracia, la implementación ya ha avanzado y los desarrolladores quieren minimizar los cambios al código. En vez de ello, Alicia propone que se seleccione la propuesta P[1, 14/09) (vea la figura 8-14): el `ClienteIU` permanece sin cambio, ya que no necesitan cambiar las interfaces de las clases `SecciónVía` y `ServicioNotificación`. Sólo necesita cambiar `ServicioNotificación` en forma tal que envíe peticiones a `SecciónVía` para revisar el acceso del despachador actual. Para revocar los privilegios del despachador cuando cambie una lista de acceso, la `SecciónVía` envía una petición al `ServicioNotificación` para que desuscriba al `Despachador`. Esto introduce una dependencia circular entre `SecciónVía` y `ServicioNotificación`, pero minimiza las modificaciones al código existente.

El equipo de arquitectura selecciona esta solución. Eduardo produce las minutas estructuradas que se muestran en la figura 8-22 (las minutas cronológicas no se muestran por brevedad).

AGENDA: Revisión del control de acceso, los despachadores sólo pueden acceder a las vías contiguas.

Cuándo y dónde	**Papel**
Fecha: 13/10	**Moderador principal**: Alicia
Inicio: 4:30 p.m.	**Tomador de tiempo**: David
Terminación: 5:30 p.m.	**Secretario de actas**: Eduardo
Lugar: Edificio de señales	**Salón**: 2300

1. Propósito

El cliente solicitó que los despachadores puedan tener acceso a las `SecciónVía` contiguas.

2. Resultado deseado

Resolver los problemas del control de acceso relacionados con este cambio de requerimientos.

3. Compartir la información [tiempo asignado: 15 minutos]

`CA[1]:` David: Recuperar la fundamentación del control de acceso.

4. Discusión [tiempo asignado: 35 minutos]

`PR[1]:` ¿Cómo debe revisarse el control de acceso con base en los requerimientos de vías contiguas?

5. Cierre [tiempo asignado: 5 minutos]

Revisar y asignar nuevos conceptos de acción.

Crítica de la reunión.

Figura 8-21 Agenda para la revisión del control de acceso del CTC.

MINUTA ESTRUCTURADA: Revisión del control de acceso, los despachadores sólo pueden tener acceso a las vías contiguas.

Cuándo y dónde	**Papel**
Fecha: 13/10	**Moderador principal**: Alicia
Inicio: 4:30 p.m.	**Tomador de tiempo**: David
Terminación: 5:30 p.m.	**Secretario de actas**: Eduardo
Lugar: Edificio de señales	**Salón**: 2300

1. Propósito

El cliente solicitó que los despachadores puedan tener acceso a las `SecciónVía` contiguas.

2. Resultado deseado

Resolver los problemas del control de acceso relacionados con este cambio de requerimientos.

3. Compartir la información

`CA[1]:` David: Recuperar la fundamentación del control de acceso.

Resultado: se recuperaron los problemas `PR[1, 13/09]` y `PR[2, 15/09]:`

PR[1, 13/09]: ¿Cómo debe integrarse el control de acceso con las SecciónVía y el ServicioNotificación? (minuta del 14/09).

Figura 8-22 Minutas estructuradas para la revisión del control de acceso del CTC. Las *cursivas* indican la fundamentación que se recuperó para el propósito de esta reunión.

P[3.1]: *La SecciónVía mantiene una lista de acceso sobre quiénes pueden examinar o modificar el estado de la SecciónVía. Para suscribirse a los eventos, un subsistema envía una petición al ServicioNotificación, el cual a su vez envía una petición a la SecciónVía para revisar el acceso.*

P[3.2]: *Las SecciónVía alojan todas las operaciones protegidas. El ClienteIU solicita la suscripción a los eventos de la SecciónVía enviando una petición a la SecciónVía y ésta revisa el acceso y luego envía una petición al ServicioNotificación.*

AR[3.1] para P[3.2]: Extensibilidad.

AR[3.2] para P[3.2]: El control de acceso y las operaciones protegidas están centralizadas en una sola clase.

P[3.3]: *No es necesario restringir el acceso a la suscripción de eventos. El ClienteIU solicita la suscripción en forma directa al ServicioNotificación. El ServicioNotificación no necesita revisar el acceso.*

AR[3.3] para P[3.3]: Los despachadores pueden examinar el estado de cualquier SecciónVía (vea R[1]).

AR[3.4] para P[3.3]: Simplicidad.

R[3]: P[3.3]. Vea el concepto de acción CA[2].

PR[2, 15/09]: ¿No deberá usarse la notificación para las peticiones? (del envío de María a las noticias el 15/09).

R[2]: *La notificación sólo deberá usarse para informar los cambios de estado. Las SecciónVía, y más generalmente el SubsistemaVía, no deberá cambiar su estado con base en eventos.*

4. Discusión

`PR[1]`: ¿Cómo debe revisarse el control de acceso con base en los requerimientos de vías contiguas?

`P[1.1]`: Las operaciones protegidas, incluyendo la suscripción, centralizadas en `SecciónVía` como en `P[3.2, 13/09]`.

`A[1.1]` contra `P[1.1]`: Esto requiere que todos los subsistemas se suscriban a los eventos de notificación a modificarse, debido a que la operación de suscripción se mueve del `ServicioNotificación` hacia la `SecciónVía`.

`P[1.2]`: El `ServicioNotificación` envía peticiones a `SecciónVía` para revisar el acceso. `SecciónVía` envía peticiones a `ServicioNotificación` para desuscribir a los despachadores a los que se les ha revocado el acceso. `P[3.1, 13/09]`

`AR[1.2]` para `P[1.2]`: Cambios mínimos a la implementación existente.

`AR[1.3]` contra `P[1.2]`: Dependencia circular.

`R[1]`: `P[1.2]`, vea `CA[2]` y `CA[3]`.

5. Cierre

`CA[2]`: Alicia: Cambiar `SecciónVía` para que desuscriba a los despachadores cuando se revocan sus derechos.

`CA[3]`: David: Modificar `ServicioNotificación` para revisar el acceso con `SecciónVía` cuando se suscriba un nuevo subsistema.

Figura 8-22 Minutas estructuradas para la revisión del control de acceso del CTC. Las *cursivas* indican la fundamentación que se recuperó para el propósito de esta reunión.

Las minutas que se muestran en la figura 8-22 sirven para dos propósitos: para registrar la fundamentación del nuevo cambio y para relacionarlo con las fundamentaciones anteriores. Esto se logra citando las fundamentaciones anteriores que se usaron para revisar la decisión del control de

acceso. Además, estas nuevas minutas se envían al grupo de noticias de arquitectura y son discutidas por los demás desarrolladores que no pudieron asistir a la reunión, completando de esta forma el ciclo de registro y aclaración de la información de la fundamentación. En el caso en que se use groupware, la nueva fundamentación puede relacionarse con las fundamentaciones anteriores con un hipervínculo, facilitando a los desarrolladores la navegación hacia la información relacionada.

Observe que, aunque se use una base de problemas para mantener y dar seguimiento a los problemas abiertos, esta base de información puede crecer con rapidez hacia un gran caos no estructurado. Además, algunos problemas no se registran, ya que no todos los problemas se discuten en reuniones. Muchos se discuten y resuelven de manera informal en charlas de pasillo. Por tanto, es necesario reconstruir las fundamentaciones faltantes del sistema e integrarlas con las fundamentaciones anteriores. Trataremos esto en la siguiente sección.

8.4.5 Reconstrucción de las fundamentaciones

La reconstrucción de las fundamentaciones es un método diferente para la captura de las fundamentaciones del sistema. En vez de capturar las decisiones y su justificación conforme suceden, las fundamentaciones se reconstruyen en forma sistemática a partir del modelo del sistema, el registro de comunicaciones y la memoria de los desarrolladores. Con este método se capturan y estructuran de manera más sistemática las fundamentaciones. Se invierten menos recursos durante las primeras fases del proceso, permitiendo, por tanto, que los desarrolladores lleguen más rápido a una solución. Además, la separación de la actividad de diseño con respecto a la captura de la fundamentación permite que los desarrolladores regresen y critiquen sus diseños en forma más objetiva. Sin embargo, la reconstrucción de las fundamentaciones se enfoca en la solución seleccionada y no captura las alternativas descartadas y su discusión. Por ejemplo, supongamos que no capturamos la fundamentación del control de acceso en el CTC y que la única información que tenemos es la del modelo de diseño del sistema (figura 8-23).

[. . .]

4. Control de acceso

El acceso en el CTC está controlado en el nivel de las `SecciónVía`: El `Despachador` que está asignado a una `SecciónVía` puede modificar su estado; esto es, abrir y cerrar señales y cambios de vía y modificar otros dispositivos. Además, el `Despachador` puede examinar el estado de las `SecciónVía` contiguas sin modificarles su estado. Esto es necesario para que el `Despachador` observe los `Tren` que están a punto de entrar en la `SecciónVía` controlada.

El control de acceso se implementa con una lista de acceso mantenida por la `SecciónVía`. La lista de acceso contiene la identidad del `Despachador` que puede modificar la `SecciónVía` (es decir, escritor) y la identidad del `Despachador` que puede ver el estado de la `SecciónVía` (es decir, lector). Para efectos de generalización, la lista de acceso se implementa de tal forma que puede incluir a varios lectores y varios escritores. La `SecciónVía` revisa la lista de acceso en cada operación que modifica o consulta el estado de la `SecciónVía`.

Cuando los subsistemas se suscriben a los eventos, el `ServicioNotificación` envía una petición a la `SecciónVía` para revisar el acceso. La `SecciónVía` envía una petición al `ServicioNotificación` para cancelar la suscripción a los despachadores que se les revoca el acceso.

El diagrama de colaboración de la figura 8-14 muestra esta solución.

[. . .]

Figura 8-23 Fragmento del documento del diseño del sistema, sección del control de acceso.

PR[1]: **¿Cómo debe integrarse el control de acceso de SecciónVía con la notificación?**

El acceso en el CTC está controlado en el nivel de las SecciónVía: El Despachador que está asignado a una SecciónVía puede modificar su estado; esto es, abrir y cerrar señales y cambios de vía y modificar otros dispositivos. Además, el Despachador puede examinar el estado de las SecciónVía contiguas sin modificarles su estado. Esto es necesario para que el Despachador observe los Tren que están a punto de entrar en la SecciónVía controlada.

P[1]: **La clase SecciónVía controla todas las modificaciones de estado y el acceso a la suscripción de notificación.** El control de acceso se implementa con una lista de acceso en SecciónVía. La clase SecciónVía revisa el acceso de quien llama para cada operación que examina o modifica el estado. En particular, el que llama se suscribe a la notificación de eventos llamando a métodos de SecciónVía, que a su vez pasa la petición al ServicioNotificación cuando se otorga el acceso. Esta solución se ilustra en la figura 8-15.	A favor: • Solución central: todos los métodos protegidos relacionados con la SecciónVía están en un solo lugar.
P[2]: **La clase SecciónVía controla todas las modificaciones de estado, el ServicioNotificación controla la suscripción.** Es similar a P[1], excepto que quien llama solicita la suscripción a eventos en forma directa al ServicioNotificación, que revisa el acceso con la SecciónVía antes de otorgar la suscripción. Esta solución se ilustra en la figura 8-14.	A favor: • El acceso es independiente de la interfaz: las interfaces de ServicioNotificación y SecciónVía son iguales, como si no hubiera control de acceso (argumento heredado). En contra: • Dependencia circular entre ServicioNotificación y SecciónVía: la SecciónVía llama a operaciones de ServicioNotificación para generar eventos, y los métodos de suscripción del ServicioNotificación llaman a operaciones de la SecciónVía para revisar el acceso.

R[1]

P[2]. P[1] habría sido una mejor solución, pero el control de acceso no se aplicaba a la notificación. Para minimizar la reconstrucción del código y el diseño se seleccionó P[2].

Figura 8-24 La fundamentación reconstruida para la notificación del problema del control de acceso del CTC.

Queremos recuperar la fundamentación del diseño del sistema para revisarla y documentarla. Decidimos organizar cada problema como una tabla con dos columnas, la izquierda para las propuestas y la derecha para sus argumentos correspondientes. En la figura 8-24 recuperamos la fundamentación para la integración del control de acceso con notificación. Identificamos dos soluciones posibles: P[1], en la cual la clase SecciónVía exporta todas las operaciones cuyo acceso está controlado, incluyendo la suscripción a las notificaciones, y P[2], en la cual el ServicioNotificación delega la revisión del control de acceso a la SecciónVía. Luego, en la

columna derecha enumeramos las ventajas y desventajas de cada solución y en la parte inferior de la tabla resumimos la justificación de la decisión como una resolución.

Cuesta menos trabajo capturar una fundamentación reconstruida, como la de la figura 8-24, que las actividades que describimos con anterioridad. Sin embargo, es más difícil capturar las alternativas descartadas y las razones para tales elecciones, en especial cuando las decisiones se revisan a lo largo del tiempo. En la figura 8-24 la resolución establece que no se seleccionó la mejor propuesta y pudimos recordar las razones para esta decisión no óptima (es decir, que ya se había terminado una buena cantidad de código antes de esta decisión y queríamos minimizar la reelaboración del código). En forma alternativa, la reconstrucción de la fundamentación es una herramienta efectiva para la revisión por la identificación de decisiones que son inconsistentes con los objetivos de diseño del proyecto. Además, aunque las decisiones revisadas no se examinen en una etapa posterior del proyecto, este conocimiento puede beneficiar a los nuevos desarrolladores asignados al proyecto o a los desarrolladores que revisen el sistema en iteraciones posteriores.

El balance entre la captura, el mantenimiento y la reconstrucción de la fundamentación difiere para cada proyecto y es necesario administrarlo con cuidado. Es relativamente frecuente ver que los esfuerzos de captura de fundamentaciones acumulan enormes cantidades de información que es inútil o que no es accesible con facilidad para los desarrolladores que deben beneficiarse con esa información. A continuación nos enfocamos en los asuntos de administración.

8.5 Administración de la fundamentación

En esta sección describimos los problemas relacionados con las actividades de administración de la fundamentación. El registro de las justificaciones para las decisiones de diseño se ve, con frecuencia, como una intrusión de la administración en el trabajo de los desarrolladores y, por tanto, las técnicas de la fundamentación encuentran resistencia por parte de los desarrolladores y a menudo degeneran en un proceso burocrático. Las técnicas de la fundamentación necesitan administrarse con cuidado para que sean útiles. En esta sección describimos la manera de:

- Escribir documentos acerca de la fundamentación (sección 8.5.1).
- Asignar responsabilidades para la captura y mantenimiento de los modelos de fundamentación (sección 8.5.2).
- Comunicar acerca de los modelos de fundamentación (sección 8.5.3).
- Usar los problemas para negociar (sección 8.5.4).
- Resolver conflictos (sección 8.5.5).

Igual que antes, continuamos enfocándonos en la actividad de diseño del sistema. Sin embargo, observe que estas técnicas pueden aplicarse de manera uniforme a todo el desarrollo.

8.5.1 Documentación de la fundamentación

Cuando la fundamentación está capturada y documentada de manera explícita, queda mejor descrita en documentos separados de los documentos del modelo del sistema. Por ejemplo, la fundamentación que hay detrás de las decisiones del análisis de requerimientos se acredita en el documento de fundamentación del análisis de requerimientos (RARD, por sus siglas en inglés), que complementa al documento de análisis de requerimientos (RAD, por sus siglas

en inglés). En forma similar, la fundamentación que hay detrás de las decisiones del diseño del sistema se acredita en el documento de fundamentación del diseño del sistema (SDRD, por sus siglas en inglés), que complementa al documento de diseño del sistema (SDD, por sus siglas en inglés). En esta sección nos enfocamos en el SDRD, ya que el diseño del sistema es la actividad que más se beneficia con la captura de la fundamentación. La figura 8-25 es un ejemplo de una plantilla para SDRD.

La audiencia del SDRD es la misma que la del SDD: el SDRD lo usan los desarrolladores cuando revisan las decisiones, los revisores cuando revisan el sistema y los nuevos desarrolladores cuando se les asigna al proyecto. Las actividades específicas para las que está orientado el documento de fundamentación se describen en la primera sección del documento. Un documento que se enfoca en la justificación del sistema para los revisores puede contener sólo las propuestas y argumentos relevantes para las resoluciones seleccionadas. Un documento que captura la mayor parte posible del contexto del diseño puede contener, además, todas las alternativas descartadas y sus evaluaciones. La primera sección también repite los objetivos de diseño que se seleccionaron al inicio del diseño del sistema. Estos representan los criterios que usan los desarrolladores para evaluar soluciones alternas.

Las siguientes dos secciones se componen de una lista de problemas, formateados de la misma manera que el problema del control de acceso que describimos con anterioridad en la figura 8-24. La lista de problemas puede constituir la justificación sistemática del diseño, o ser tan sólo una colección de problemas capturados en el curso del diseño del sistema. Los problemas pueden relacionarse entre sí con subproblemas y referencias a problemas consecuentes. Por último, en este documento pueden insertarse apuntadores hacia el SDD en las secciones relevantes para facilitar la navegación.

Documento de fundamentación del diseño del sistema.

1. Introducción
   - 1.1 Propósito del documento
   - 1.2 Objetivos del diseño
   - 1.3 Definiciones, siglas y abreviaturas
   - 1.4 Referencias
   - 1.5 Panorama
2. Fundamentación para la arquitectura de software actual
3. Fundamentación para la arquitectura de software propuesta
   - 3.1 Panorama
   - 3.2 Fundamentación para la descomposición en subsistemas
   - 3.3 Fundamentación para la correspondencia entre hardware y software
   - 3.4 Fundamentación para la administración de datos persistentes
   - 3.5 Fundamentación para el control del acceso y la seguridad
   - 3.6 Fundamentación para el control del software global
   - 3.7 Fundamentación para las condiciones de frontera

Glosario

Figura 8-25 Un ejemplo de plantilla para el SDRD.

La segunda sección del SDRD describe la fundamentación para el sistema que se está reemplazando. Si no hay sistema anterior, esta sección puede sustituirse con la fundamentación para sistemas similares. El propósito de esta sección es hacer explícitas las fundamentaciones anteriores que se han explorado y los problemas que deben observar los desarrolladores.

La tercera sección del SDRD describe la fundamentación del nuevo sistema. En forma paralela a la estructura del SDD, esta sección se divide en siete subsecciones:

- *Panorama* presenta una vista a ojo de pájaro de esta sección, incluyendo un resumen de los problemas más críticos que se trataron durante el diseño del sistema.
- *Fundamentación para la descomposición en subsistemas* justifica la descomposición en subsistemas seleccionada. ¿Cómo minimiza el acoplamiento? ¿Cómo incrementa la coherencia? ¿Cómo satisface los objetivos de diseño que se establecieron en la primera sección? Desde un punto de vista más general, aquí se enumera cualquier problema que haya tenido un impacto en la descomposición en subsistemas.
- *Fundamentación para la correspondencia entre hardware y software* justifica la configuración de hardware seleccionada, la asignación de subsistemas a los nodos y los problemas del código heredado.
- *Fundamentación para la administración de datos persistentes* justifica la selección del mecanismo de almacenamiento de datos. ¿Por qué se seleccionaron archivos planos o base de datos relacional o base de datos orientada a objetos? ¿Cuáles criterios de diseño condujeron a esta decisión? ¿Cuáles otros problemas debe manejar el componente de almacenamiento?
- *Fundamentación para el control de acceso y la seguridad* justifica la implementación del control de acceso seleccionado. ¿Por qué se seleccionaron las listas de acceso o las capacidades? ¿Cómo se integra el control de acceso con otros sistemas para la distribución de información, como el middleware o el subsistema de notificación? El problema del control de acceso para el CTC se incluiría en esta sección.
- *Fundamentación para el control del software global* justifica el mecanismo de control seleccionado. ¿Cuáles componentes heredados restringieron el control del software? ¿Hubo componentes heredados incompatibles? ¿Cómo se manejó esto?
- *Fundamentación para las condiciones de frontera* justifica cada condición de frontera y su manejo. ¿Por qué revisa el sistema la consistencia de la base de datos al arranque? ¿Por qué el sistema da un aviso a los `Despachador` 10 minutos antes de apagarse (en vez de 20 minutos o dos horas)?

El SDRD deberá escribirse al mismo tiempo que el SDD y revisarse cada vez que se revise el SDD. Esto asegura que ambos documentos sean consistentes y motiva a los desarrolladores para que hagan explícita la fundamentación. El SDRD no sólo deberá revisarse cuando se cambia el diseño, sino también cuando se encuentra que falta fundamentación (por ejemplo, durante las revisiones).

8.5.2 Asignación de responsabilidades

La asignación de responsabilidades para la captura y el mantenimiento de la fundamentación es la decisión administrativa más crítica para que sean útiles los modelos de fundamentación. El mantenimiento de la fundamentación puede percibirse con facilidad como un proceso burocrático

intruso mediante el cual los desarrolladores necesitan justificar todas las decisiones. En vez de ello, una pequeña cantidad de personas que tenga acceso a la información de todos los desarrolladores, como los borradores de documentos de diseño y grupos de noticias de los desarrolladores, debe mantener los modelos de fundamentación. Este pequeño grupo de personas, al convertirse en el historiador del sistema, llega a ser útil para los demás desarrolladores cuando proporciona información sobre la fundamentación y, por tanto, crea un incentivo en los desarrolladores para que les proporcionen información. A continuación se tienen los papeles principales relacionados con el mantenimiento del modelo de razones:

- El **secretario de actas** registra la fundamentación en las reuniones. Esto incluye el registro cronológico de declaraciones durante las reuniones y su reestructuración con los problemas después de la reunión (vea la sección 8.4.2).
- El **editor de fundamentación** recolecta y organiza la información relacionada con la fundamentación. Esto incluye la obtención de las minutas de reunión del secretario de actas, de los reportes de evaluación de prototipos y tecnología de los desarrolladores y de los borradores de todos los modelos del sistema y documentos de diseño de los escritores técnicos. El editor de fundamentación impone una sobrecarga mínima sobre los desarrolladores y escritores, realizando el papel de estructuración e indexado. Los desarrolladores no necesitan proporcionar información estructurada como los modelos de problemas, sino que el editor de fundamentación construye un índice de todos los problemas.
- El **revisor** examina las fundamentaciones capturadas por el editor de fundamentación e identifica brechas que necesiten reconstruirse. Luego el revisor recolecta la información relevante de los registros de comunicaciones y, si es necesario, de los desarrolladores. Éste no debe ser un papel administrativo o de aseguramiento de la calidad: el revisor debe beneficiar en forma directa a los desarrolladores para que pueda recopilar información válida. Este papel puede combinarse con el de editor de fundamentación.

El tamaño del proyecto determina la cantidad de secretarios de actas, editores de fundamentación y revisores. Puede usarse la siguiente heurística para la asignación de estos papeles:

- **Un secretario de actas por equipo.** Las reuniones se organizan, por lo general, por equipo de subsistema o por equipo de funcionalidad cruzada. Un desarrollador de cada equipo puede funcionar como secretario de actas, distribuyendo por tanto este papel consumidor de tiempo a lo largo de todo el proyecto.
- **Un editor de fundamentación por proyecto.** El papel de editor de fundamentación para un proyecto es un papel de tiempo completo. A diferencia del papel del secretario de actas, que puede ser rotatorio, el papel del editor de fundamentación requiere consistencia y debe asignarse a una sola persona. En proyectos pequeños puede asignarse este papel al arquitecto del sistema (vea el capítulo 6, *Diseño del sistema*).
- **Incrementar la cantidad de revisores después de la entrega.** Cuando el sistema se entrega y disminuye la cantidad de desarrolladores necesarios de manera directa para el proyecto, deberán asignarse algunos desarrolladores al papel de revisores para recuperar y organizar la mayor cantidad de información posible. La información de la fundamentación todavía es recuperable de la memoria de los desarrolladores, pero desaparece con rapidez cuando los desarrolladores pasan a otros proyectos.

8.5.3 Heurística para la comunicación acerca de la fundamentación

Una gran parte de la comunicación *es* información sobre la fundamentación, ya que los argumentos son, por definición, la fundamentación (vea la sección 8.2). Los desarrolladores argumentan acerca de los objetivos del diseño, si un problema es relevante o no, el beneficio de varias soluciones y su evaluación. La fundamentación constituye un cuerpo de información grande y complejo, por lo general más grande que el sistema mismo. Además, con frecuencia la comunicación sucede en foros pequeños, por ejemplo, en una reunión de equipo o una conversación junto a la cafetera. El reto de la comunicación acerca de la fundamentación es hacer accesible esta información a todas las partes involucradas sin causar una sobrecarga de información. En este capítulo nos enfocamos en técnicas para la captura y estructuración de la fundamentación, como el uso de un modelo de problemas en las minutas, conversaciones complementarias y documentación de la fundamentación. Además, pueden usarse las siguientes heurísticas para incrementar la estructura de la fundamentación y facilitar la navegación por ellas:

- **Nombre los problemas en forma consistente.** Los problemas deberán nombrarse en forma consistente y única en todas las minutas, grupos de noticias, mensajes de correo electrónico y documentos. Los problemas pueden tener un número (por ejemplo, 1291) y un nombre corto (por ejemplo, "el problema del acceso y notificación") para facilitar su referencia.
- **Centralice los problemas.** Aunque los problemas se discutirán en muchos contextos, motive que un contexto (por ejemplo, un grupo de noticias o una base de problemas) sea el depósito central de problemas. Esta base de problemas debe mantenerla el editor de fundamentación, pero puede usarla y ampliarla cualquier desarrollador. Esto permite que los desarrolladores busquen información con rapidez.
- **Haga referencias cruzadas de problemas y elementos del sistema.** La mayoría de los problemas se aplican a elementos específicos de los modelos del sistema (por ejemplo, un caso de uso, un objeto, un subsistema). Saber a cuál elemento del modelo se aplica un problema específico es directo. Sin embargo, saber cuáles problemas se aplican a un elemento del modelo específico es una cuestión mucho más difícil. Para facilitar este tipo de consulta, los problemas deberán asociarse al elemento del modelo aplicable cuando se plantean los problemas.
- **Administre el cambio.** La fundamentación evoluciona conforme lo hacen los modelos. Por tanto, la administración de la configuración debe aplicarse en forma consistente a la fundamentación y los documentos, así como se aplica a los modelos del sistema.

La captura y estructuración de la fundamentación no sólo mejora la comunicación acerca de la fundamentación, sino que también facilita la comunicación acerca de los modelos del sistema. La integración de la fundamentación y la información del sistema permite que los desarrolladores mantengan en mejor forma ambos tipos de información.

8.5.4 Modelado y negociación de problemas

Las decisiones más importantes en el desarrollo son resultado de la negociación. Diferentes partes que representan intereses diferentes, y a veces conflictivos, llegan a un consenso sobre algún aspecto del sistema. El análisis de requerimientos incluye la negociación de la funcionalidad con el cliente. El diseño del sistema incluye la negociación de las interfaces de subsistemas entre desarrolladores. La integración incluye la resolución de conflictos entre desarrolladores. Usamos el modelado de problemas para representar la información intercambiada durante esas negociaciones para capturar la fundamentación. También podemos usar el modelado de problemas para facilitar las negociaciones.

La negociación tradicional, que consiste en negociar posiciones, con frecuencia consume tiempo y es ineficiente, en especial cuando las partes negociadoras sostienen posiciones incompatibles. Se desperdician esfuerzos en la defensa de la posición personal, citando todas sus ventajas, mientras que la parte opositora desperdicia esfuerzos denigrando la posición del otro citando todas sus desventajas. La negociación avanza a pequeños pasos hacia un consenso o termina en una solución arbitraria sobre el asunto negociable. Además, esto puede suceder aun cuando las partes negociadoras tengan intereses compatibles: cuando se defienden posiciones las personas tienen mucho mayor problema para evolucionar o cambiar su posición sin pérdida de credibilidad. El método de negociación Harvard [Fischer *et al.*, 1991] aborda estos puntos retirando el foco de las posiciones. A continuación presentamos varios puntos importantes del método Harvard redactados desde el punto de vista del modelado de problemas:

- **Separe a los desarrolladores de las propuestas.** Los desarrolladores pueden desperdiciar muchos recursos desarrollando una propuesta específica (es decir, una posición), a un punto tal que la crítica de la propuesta se toma como una crítica personal hacia el desarrollador. Se debe separar a los desarrolladores y a las propuestas para facilitar la evolución o rechazo de una propuesta. Esto puede lograrse haciendo que varios desarrolladores trabajen en la misma propuesta o haciendo que todas las partes involucradas participen en el desarrollo de todas las propuestas. La separación del trabajo de diseño y la implementación puede facilitar aún más esta distinción. Al asegurarse que la negociación se dé antes que la implementación, y antes de que se hayan comprometido recursos considerables, los desarrolladores pueden hacer que las propuestas evolucionen hacia cosas con las que todos puedan vivir.
- **Enfóquese en los criterios y no en las propuestas.** Como se dijo antes, la mayoría de los argumentos pueden trazarse hasta los criterios, a menudo implícitos, que se usan para la evaluación. Una vez que se tiene en su lugar un conjunto de criterios aceptados, la evaluación y selección de propuestas es mucho menos controvertida. Además, los criterios están mucho menos sujetos a cambios que los demás factores del producto. Llegar a un acuerdo pronto sobre los criterios también facilita la revisión de las decisiones.
- **Tome en cuenta todos los criterios en vez de maximizar uno solo.** Diferentes criterios reflejan los intereses de partes diferentes. Los criterios de desempeño están motivados, por lo general, por preocupaciones de usabilidad. Los criterios de modificabilidad están motivados por preocupaciones de mantenimiento. Aunque algún criterio se considere con mayor prioridad que los demás, la optimización únicamente de estos criterios de alta prioridad pone en riesgo que se dejen fuera de la negociación a una o más partes.

La visión del desarrollo como una negociación responde a los aspectos sociales del desarrollo. Los desarrolladores son personas que, además de sus opiniones técnicas, pueden tener una perspectiva emocional sobre soluciones diferentes. Esto puede influir en sus relaciones con los demás desarrolladores (y a veces interferir con ellas) conforme se presentan los conflictos. El uso del modelado de problemas para capturar la fundamentación y dirigir las decisiones puede integrar y mejorar los aspectos técnicos y sociales del desarrollo.

8.5.5 Estrategias para la resolución de conflictos

En ocasiones los participantes en el proyecto no obtienen un consenso mediante la negociación. En tales casos, es esencial que se cuente con estrategias para la resolución de conflictos a fin de manejar la situación. Las peores decisiones de diseño son las que no se toman a causa de falta de consenso o por la ausencia de estrategias para la resolución de conflictos. Esto retrasa las decisiones esenciales hasta muy tarde en el desarrollo, dando como resultado altos costos de rediseño y registro.

Son posibles muchas estrategias diferentes para la resolución de conflictos. Por ejemplo, considere las cinco estrategias siguientes:

- **La mayoría gana.** En caso de conflicto, un voto mayoritario puede eliminar el bloqueo y resolver la decisión. Varias herramientas de colaboración permiten que los usuarios asignen peso a diferentes argumentos en el modelo de problemas y, por tanto, calculen con una fórmula aritmética cuál propuesta debe seleccionarse [Purvis *et al.*, 1996]. Esto asume que las opiniones de cada uno de los participantes tiene la misma importancia y que, desde el punto de vista estadístico, el grupo toma decisiones correctas.
- **El propietario tiene la última palabra.** En esta estrategia, el propietario de un asunto (la persona que lo planteó) es responsable de decidir el resultado. Esto supone que el propietario tiene el interés más grande en el asunto.
- **La gerencia siempre está en lo correcto.** Una estrategia alternativa es apoyarse en la jerarquía de la organización. Si un grupo no puede llegar al consenso, el gerente del grupo impone una decisión basado en los argumentos. Esto asume que el gerente es capaz de comprender los argumentos y tomar los compromisos adecuados.
- **Los expertos siempre tienen la razón.** En esta estrategia un experto externo, ajeno al debate, valora la situación y aconseja el mejor curso de acción. Por ejemplo, durante el análisis de requerimientos se puede entrevistar a un usuario de pruebas para evaluar las propuestas diferentes sobre un problema. Por desgracia, tal experto tiene un conocimiento limitado de las demás decisiones del sistema o, de manera más general, del contexto del diseño.
- **El tiempo decide.** Conforme un problema se deja sin resolver, el tiempo llega a ser una presión y obliga a una decisión. Además, los problemas controvertidos pueden llegar a ser más fáciles de resolver conforme se toman otras decisiones y se definen otros aspectos del sistema. El peligro de esta estrategia es que conduce a decisiones que optimizan criterios a corto plazo (como la facilidad de la implementación) y no toman en cuenta los criterios a largo plazo (como la modificabilidad y el mantenimiento).

Las estrategias *La mayoría gana* y *El propietario tiene la última palabra* no funcionan bien. Ambas producen resultados inconsistentes (multiplicidad de tomadores de decisiones) y decisiones que no están bien apoyadas por el resto de los participantes. Las estrategias *La gerencia siempre está en lo correcto* y *Los expertos siempre tienen la razón* conducen a mejores decisiones técnicas y a mejor consenso cuando el gerente y el experto tienen suficiente conocimiento. La estrategia *El tiempo decide* es una retirada, aunque una que puede dar como resultado una costosa repetición del trabajo.

En la práctica, primero tratamos de llegar a un consenso, y en el caso de la falta de consenso, recurrimos a una estrategia de experto o gerente. Si falla la estrategia del experto o gerente dejamos que el tiempo decida o lo resolvemos por voto obligatorio de la mayoría.

8.6 Ejercicios

1. A continuación se tiene un fragmento de un documento de sistema para un sistema de administración de accidentes. Es una descripción, en lenguaje natural, de la fundamentación para que una base de datos relacional sea el almacenamiento permanente. Modele esta fundamentación con problemas, propuestas, argumentos, criterios y resoluciones, como se definió en la sección 8.3.

> Un asunto fundamental en el diseño de bases de datos fue la realización del motor de la base de datos. Los requerimientos no funcionales iniciales del subsistema de base de datos insisten en el uso de una base de datos orientada a objetos para la máquina subyacente. Otras opciones posibles incluyen el uso de una base de datos relacional, un sistema de archivos o una combinación de las otras opciones. Una base de datos orientada a objetos tiene la ventaja de poder manejar relaciones de datos complejas y se apega por completo a las palabras de moda. Por otro lado, las bases de datos OO pueden ser demasiado lentas para grandes volúmenes de datos o accesos muy frecuentes. Además, los productos existentes no se integran bien con CORBA, debido a que el protocolo no soporta características específicas del lenguaje de programación, como las asociaciones Java. El uso de una base de datos relacional proporciona un motor más robusto con características de desempeño más altas y una gran experiencia y herramientas para trabajar con ella. Además, el modelo de datos relacional se integra muy bien con CORBA. Por lo que se refiere a las desventajas, este modelo no soporta con facilidad las relaciones de datos complejas. La tercera opción se propuso para manejar tipos de datos específicos que se escriben una vez y se leen con poca frecuencia. Este tipo de datos (incluyendo lecturas de sensor y salidas de control) tiene menos relaciones con poca complejidad y debe archivarse durante largos periodos de tiempo. Los archivos proporcionan una solución de archivado fácil y pueden manejar grandes cantidades de datos. En forma alternativa, habrá que escribir cualquier código a partir de cero, incluyendo la serialización del acceso. Decidimos usar sólo una base de datos relacional, basados en el requerimiento de usar CORBA y a la luz de la simplicidad relativa de las relaciones entre los datos persistentes del sistema.

2. En la sección 8.3 examinamos un problema relacionado con el control del acceso y la notificación en el sistema CTC. Seleccione un problema similar que pueda suceder en el desarrollo y el CTC y cólmelo con propuestas, criterios y argumentos relevantes, justificando una resolución. Los ejemplos de estos problemas incluyen:
   - ¿Cómo puede mantenerse la consistencia entre `servidorPrincipal` y `respaldoVivo`?
   - ¿Cómo debe detectarse la falla del `servidorPrincipal` e implementarse el cambio subsiguiente al `respaldoVivo`?
3. Está desarrollando una herramienta CASE usando UML como notación principal y está considerando la integración de la fundamentación en la herramienta. Describa la manera en que un desarrollador puede asociar problemas a diferentes elementos del modelo. Trace un diagrama de clase sobre el modelo de problemas y sus asociaciones con los elementos del modelo.
4. Considere la misma herramienta CASE del ejercicio 3. Está considerando la generación de un documento de fundamentación a partir del modelo. Describa la correspondencia entre clases, problemas y las secciones del documento de fundamentación.
5. Está integrando un sistema para el reporte de errores con una herramienta de administración de configuración para dar seguimiento a los reportes de errores, resolución de errores, petición de características y mejoras. Está considerando un modelo de problemas para la

integración de estas herramientas. Trace un diagrama de clase del modelo de problemas, la discusión correspondiente, la administración de la configuración y los elementos para el reporte de errores.

Referencias

[Boehm *et al.*, 1995] B. Boehm, P. Bose, E. Horowitz y M.J. Lee, "Software requirements negotiation and renegotiation aids: A theory-W based spiral approach", *Proceedings of the ICSE-17*, Seattle, 1995.

[Buckingham Shum y Hammond, 1994] S. Buckingham Shum y N. Hammond, "Argumentation-based design rationale: What use at what cost?" *International Journal of Human-Computer Studies*, Vol. 40, págs. 603–652, 1994.

[Conklin y Burgess-Yakemovic, 1991] J. Conklin y K. C. Burgess-Yakemovic, "A process-oriented approach to design rationale", *Human-Computer Interaction*, Vol. 6, págs. 357–391, 1991.

[Dutoit *et al.*, 1996] A. H. Dutoit, B. Bruegge y R. F. Coyne, "The use of an issue-based model in a team-based software engineering course", *Conference Proceedings of Software Engineering: Education and Practice* (SEEP'96). Dunedin, NZ, enero de 1996.

[Fischer *et al.*, 1991] R. Fisher, W. Ury y B. Patton, *Getting to Yes: Negotiating Agreement Without Giving In*, 2a ed. Penguin Books, Nueva York, 1991.

[Kunz y Rittel, 1970] W. Kunz y H. Rittel, "Issues as elements of information systems", *Working Paper No. 131*, Institut für Grundlagen der Plannung, Universität Stuttgart, Alemania, 1970.

[Lee, 1990] J. Lee, "A qualitative decision management system", *Artificial Intelligence at MIT: Expanding Frontiers*. P.H Winston & S. Shellard (eds.) MIT Press, Cambridge, MA, Vol. 1, págs. 104–133, 1990.

[Lee, 1997] J. Lee, "Design rationale systems: Understanding the issues", *IEEE Expert*, mayo-junio de 1997.

[MacLean *et al.*, 1991] A. MacLean, R. M. Young, V. Bellotti y T. Moran, "Questions, options, and criteria: Elements of design space analysis", *Human-Computer Interaction*, Vol. 6, págs. 201–250, 1991.

[Moran y Carroll, 1996] T. P. Moran & J. M. Carroll (eds.), *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Potts *et al.*, 1988] C. Potts y G. Bruns, "Recording the reasons for design decisions", *Proceedings of the 10th International Conference on Software Engineering*, págs. 418–427, 1988.

[Potts *et al.*, 1994] C. Potts, K. Takahashi y A. I. Anton, "Inquiry-based requirements analysis", *IEEE Software*, Vol. 11, no. 2, págs. 21–32, 1994.

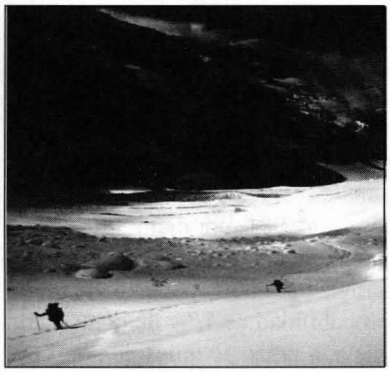
[Potts, 1996] C. Potts, "Supporting software design: Integrating design methods and design rationale", *Design Rationale: Concepts, Techniques, and Use*. T.P. Moran & J.M. Carroll (eds.) Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Purvis *et al.*, 1996] M. Purvis, M. Purvis y P. Jones, "A group collaboration tool for software engineering projects", *Conference proceedings of Software Engineering: Education and Practice (SEEP'96)*. Dunedin, NZ, enero de 1996.

[Shipman *et al.*, 1997] F.M. Shipman III y R.J. McCall, "Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication", *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, Vol. 11, no. 2, 1997.

9

9



Pruebas

Se ha concluido el software.
Ahora sólo estamos tratando de que funcione.
— Declaración hecha en una revisión conjunta del programa ejecutivo STARS E-8A FSD

Las pruebas son el proceso de encontrar diferencias entre el comportamiento esperado, especificado por los modelos del sistema, y el comportamiento observado del sistema. Las pruebas unitarias encuentran diferencias entre el modelo de diseño de objetos y sus componentes correspondientes. Las pruebas estructurales encuentran diferencias entre el modelo del diseño del sistema y un subconjunto de subsistemas integrados. Las pruebas funcionales encuentran diferencias entre el modelo de caso de uso y el sistema. Por último, las pruebas de desempeño encuentran diferencias entre los requerimientos no funcionales y el desempeño real del sistema. Cuando se encuentran diferencias, los desarrolladores identifican el defecto que causa la falla observada y modifican el sistema para corregirlo. En otros casos se identifica al modelo como causa de la diferencia y se actualiza éste para que refleje el estado del sistema.

Desde el punto de vista del modelado, las pruebas son un intento de falsificación del sistema con respecto a los modelos del sistema. El objetivo es diseñar pruebas que manifiesten los defectos que hay en el sistema y que revelen los problemas. Esta actividad es contraria a todas las demás actividades que describimos en los capítulos anteriores: análisis, diseño, implementación, comunicación y negociación son actividades constructivas. Sin embargo, las pruebas están orientadas al quebrantamiento del sistema. En consecuencia, las pruebas las realizan, por lo general, desarrolladores que no estuvieron involucrados en la construcción del sistema.

En este capítulo, primero motivamos la importancia de las pruebas. Proporcionamos una visión de conjunto de las actividades de las pruebas, describimos con mayor detalle los conceptos de defecto, error, falla y prueba, y luego describimos las actividades de prueba que dan como resultado el plan, diseño y ejecución de las mismas. Concluimos este capítulo tratando problemas administrativos relacionados con las pruebas.

9.1 Introducción

Las pruebas son el proceso de análisis de un sistema, o componente de un sistema, para detectar las diferencias entre el comportamiento especificado (requerido) y el observado (existente). Por desgracia, es imposible probar por completo un sistema no trivial. En primer lugar, las pruebas no son determinantes. En segundo, es necesario realizar las pruebas bajo restricciones de tiempo y presupuesto. En consecuencia, los sistemas se entregan sin estar probados por completo, lo que conduce a defectos que son descubiertos por los usuarios finales.

El primer lanzamiento del transbordador espacial Columbia en 1981, por ejemplo, se canceló a causa de un problema que no se detectó durante el desarrollo. Se encontró que el problema fue provocado por un cambio hecho dos años antes por un programador, quien, por error, cambió un factor de retraso de 50 a 80 milisegundos. Esto añadió la probabilidad de 1/67 de que fallara el lanzamiento de cualquier transbordador espacial. Por desgracia, a pesar de miles de horas de pruebas después de que se hizo el cambio, no se descubrió el defecto en la fase de pruebas. Durante el lanzamiento real el defecto causó un problema de sincronización en las cinco computadoras a bordo del transbordador que dio lugar a la decisión de abortar el lanzamiento. El siguiente es un fragmento de un artículo de Richard Feynman que describe los retos de las pruebas del transbordador espacial.

> En un total de casi 250,000 segundos de operación, los motores habían fallado en forma seria tal vez 16 veces. Los ingenieros pusieron mucha atención en esas fallas y trataron de remediarlas lo más pronto posible. Esto se hace mediante estudios de prueba en aparatos especiales diseñados en forma experimental para la falla en cuestión, mediante una inspección cuidadosa del motor buscando pistas sugerentes (como grietas) y mediante estudios y análisis considerables...
>
> La forma usual en que se prueban tales motores (para aeronaves civiles o militares) puede llamarse prueba de componentes del sistema o de abajo hacia arriba. Primero necesitan comprenderse por completo las propiedades y limitaciones de los materiales que se usan (para las hojas de las turbinas, por ejemplo), y comienzan las pruebas en aparatos experimentales para determinarlas. Con este conocimiento se diseñan y prueban de manera individual partes componentes más grandes (como los cojinetes). Conforme se observan deficiencias y errores de diseño se corrigen y verifican con más pruebas. Debido a que se prueba un componente a la vez, estas pruebas y modificaciones no son excesivamente caras. Por último, hay muchas probabilidades de que las modificaciones al motor para resolver las últimas dificultades no sean muy difíciles de realizar, porque la mayoría de los problemas serios ya se han descubierto y resuelto en las etapas anteriores, menos caras, del proceso.
>
> El motor principal del transbordador espacial se manejó de manera diferente, podríamos decir que de arriba hacia abajo. El motor se diseñó y ensambló de una vez con relativamente muy pocos estudios preliminares detallados del material y sus componentes. Luego, cuando se encuentran problemas en los cojinetes, hojas de turbina, tubos de enfriamiento, etc., es más difícil y caro descubrir la causa y hacer cambios. Por ejemplo, se encontraron grietas en las hojas de turbina de la turbobomba de oxígeno a alta presión. ¿Son causadas por fallas en el material, por el efecto de la atmósfera de oxígeno sobre las propiedades del material, por los esfuerzos térmicos del arranque o el apagado, por la vibración y esfuerzo de mantenerse funcionando o principalmente por alguna resonancia a determinadas velocidades, etcétera? ¿Qué tanto puede funcionar desde que se inicia la grieta hasta la falla y qué tanto depende del nivel de potencia? El uso del motor terminado como banco de prueba para resolver estas preguntas es excesivamente caro. No quiere perderse un motor completo para saber cuándo y cómo sucede una falla.

No obstante, es esencial un conocimiento preciso de esta información para adquirir confianza en la confiabilidad del motor que se usa. Sin una comprensión detallada no puede lograrse la confianza. Una desventaja adicional del método de arriba hacia abajo es que, si se consigue la comprensión de un defecto, puede ser imposible implementar una corrección simple, como una nueva forma para la carcasa de la turbina, sin rediseñar todo el motor.

El motor principal del transbordador espacial es una máquina extraordinaria. Tiene una relación peso-potencia mucho mayor que cualquier motor anterior. Está construido al límite, o más allá, de experiencias de ingeniería anteriores. Por tanto, como era de esperarse, aparecieron muchos tipos diferentes de imperfecciones y dificultades. Por desgracia, debido a que se construyó de arriba hacia abajo, son difíciles de encontrar y componer. No se obtuvo el propósito de diseño de un tiempo de vida de encendidos equivalentes a 55 misiones (27,000 segundos de operación, ya sea en una misión de 500 segundos o en un banco de pruebas). El motor ahora requiere mantenimiento muy frecuente y el reemplazo de partes importantes, como las turbobombas, cojinetes, carcasas de hojas metálicas, etc. La turbobomba de combustible de alta presión tiene que reemplazarse cada tres o cuatro misiones equivalentes (aunque puede ser que para ahora ya se haya corregido) y la turbobomba de oxígeno de alta presión cada cinco o seis. Esto es, cuando mucho, 10% de la especificación original.

El artículo de Feynman[1] nos da una idea de los problemas asociados con la prueba de sistemas complejos. Aunque el transbordador espacial es un sistema de hardware y software complejo en extremo, los retos de las pruebas son los mismos para cualquier sistema complejo.

A menudo se han visto las pruebas como un trabajo que pueden realizar los principiantes. Los gerentes querrán asignar a los nuevos miembros al equipo de pruebas, debido a que las personas con experiencia detestan las pruebas o porque se les necesita para los trabajos más importantes de análisis y diseño. Por desgracia, tal actitud conduce a muchos problemas. Para probar un sistema en forma efectiva, quien hace la prueba debe tener una comprensión detallada del sistema completo, que va desde las decisiones de requerimientos hasta las del diseño del sistema y los problemas de implementación. Quien hace la prueba también debe tener conocimientos sobre las técnicas de pruebas, y aplicarlas en forma eficaz y eficiente para cumplir con las restricciones de tiempo, presupuesto y calidad.

Este capítulo está organizado de la manera siguiente: en la sección 9.2 revisamos las pruebas y mostramos su relación con otros métodos para el aseguramiento de la calidad. En la sección 9.3 definimos con mayor detalle los elementos del modelo relacionados con las pruebas, incluyendo los defectos, su manifestación y su relación con las pruebas. En la sección 9.4 describimos las actividades de prueba que se encuentran en el proceso de desarrollo. Primero describimos las pruebas unitarias, que se enfocan en la localización de fallas en un solo componente. Luego describimos las pruebas de integración y del sistema, que se enfocan en la localización de fallas en combinaciones de componentes y en el sistema completo, respectivamente. También

1. Feynman [Feynman, 1988] escribió el artículo cuando era miembro de la comisión presidencial que investigaba la explosión del transbordador espacial Challenger en enero de 1985. Se rastreó la causa del accidente hasta una erosión de los anillos O en los cohetes secundarios sólidos. Además de los problemas de pruebas del motor principal del transbordador y los cohetes secundarios sólidos, el artículo menciona el fenómeno del cambio gradual de los criterios de aceptación de pruebas y los problemas resultantes de la falta de comunicación entre la administración y los desarrolladores que se encuentra, por lo general, en las organizaciones jerárquicas.

describimos las actividades de prueba que se enfocan en los requerimientos no funcionales, como las pruebas de desempeño y las pruebas de fortaleza. Concluimos la sección 9.4 con las actividades de prueba de campo y de instalación. En la sección 9.5 tratamos los problemas administrativos relacionados con las pruebas.

9.2 Un panorama de las pruebas

La **confiabilidad** es una medida del éxito con que el comportamiento de un sistema se apega a la especificación de su comportamiento. La **confiabilidad del software** es la probabilidad de que un sistema de software no causará la falla del sistema durante un tiempo especificado bajo condiciones especificadas [IEEE Std. 982-1989]. La **falla** es cualquier desviación del comportamiento observado con respecto al especificado. Un **error** significa que el sistema está en un estado tal que el procesamiento adicional del sistema conducirá a una falla, lo cual causa que el sistema se desvíe del comportamiento pretendido. Un **defecto** es la causa mecánica o algorítmica de un error. El objetivo de las pruebas es maximizar la cantidad de defectos descubiertos, lo cual luego permite que los desarrolladores los corrijan e incrementen la confiabilidad del sistema.

Definimos las pruebas como el intento sistemático de *localizar errores en forma planeada* en el software implementado. Esta definición contrasta con otra usada por lo común que dice que "las pruebas son el proceso de demostrar que *no hay errores presentes*". La distinción entre estas dos definiciones es importante. Nuestra definición no significa que simplemente demostramos que el programa hace lo que pretende hacer. El objetivo explícito de las pruebas es demostrar la presencia de defectos. Nuestra definición implica que usted, el desarrollador, está deseoso de desmantelar cosas. La mayoría de las actividades del proceso de desarrollo son constructivas: durante el análisis, diseño e implementación los objetos y relaciones se identifican, refinan y se hacen corresponder con un ambiente de computadora.

Las pruebas requieren una manera de pensar diferente en la que los desarrolladores tratan de detectar defectos en el sistema; esto es, diferencias entre la realidad y los requerimientos. A muchos desarrolladores se les dificulta. Una razón es la forma en que se usa la palabra "éxito" durante las pruebas. Muchos gerentes de proyecto dicen que un caso de prueba es "exitoso" si no encuentran ningún error; esto es, usan la segunda definición de las pruebas durante el desarrollo. Sin embargo, debido a que "exitoso" denota un logro y "no exitoso" significa algo indeseable, no deben usarse así estas palabras durante las pruebas.

En este capítulo tratamos a las pruebas como una actividad basada en la falsificación de los modelos del sistema, la cual se basa en la falsificación de Popper de las teorías científicas [Popper, 1992]. De acuerdo con Popper, cuando se prueba una teoría científica, el objetivo es diseñar experimentos que falsifiquen la teoría subyacente. Si los experimentos no pueden romper la teoría, se fortalece nuestra confianza en ella y se le adopta (hasta que al final se le falsifica). En forma similar, en la prueba de software el objetivo es identificar errores en el sistema de software ("falsificar la teoría"). Si ninguna de las pruebas puede falsificar el comportamiento del sistema de software con respecto a los requerimientos, está listo para entregarse.

En otras palabras, un sistema de software se lanza cuando los intentos de falsificación ("pruebas") han incrementado la confiabilidad del software, asegurando una mayor confianza en que el sistema de software haga lo que se supone que debe hacer.

9.2.1 Técnicas de control de calidad

Hay muchas técnicas para incrementar la confiabilidad de un sistema de software. La figura 9-1 se enfoca en tres clases de técnicas para evitar los defectos, detectarlos y tolerarlos. Las *técnicas para evitar defectos* tratan de detectar errores en forma estadística; esto es, sin apoyarse en la ejecución de ninguno de los modelos del sistema, en particular el modelo de código. Las *técnicas para la detección de defectos*, como la depuración y las pruebas, son experimentos no controlados y controlados, respectivamente, que se usan durante el proceso de desarrollo para identificar errores y encontrar los defectos subyacentes antes de lanzarse el sistema. Las *técnicas de tolerancia de defectos* suponen que puede lanzarse un sistema con errores y que las fallas del sistema pueden manejarse recuperándose de ellas en el momento de la ejecución.

9.2.2 Técnicas para evitar defectos

La evitación de defectos trata de impedir la ocurrencia de errores y fallas encontrando defectos en el sistema antes de lanzarlo. Las técnicas para evitar defectos incluyen:

- Desarrollo de metodologías
- Administración de la configuración
- Técnicas de verificación
- Revisiones

El **desarrollo de metodologías** evita los defectos proporcionando técnicas que minimizan la introducción de defectos en los modelos del sistema y en el código. Tales técnicas incluyen la representación, sin ambigüedades, de los requerimientos, el uso de abstracción y encapsulamiento de datos, la minimización del acoplamiento entre subsistemas y la maximización de la coherencia entre subsistemas, la definición temprana de las interfaces de subsistemas y la captura de la información de las razones para las actividades de mantenimiento. En los capítulos 4 a 7 describimos estas técnicas. La suposición de la mayoría de estas técnicas es que el sistema contiene menos defectos si se maneja la complejidad con enfoques basados en modelos.

La **administración de la configuración** evita los defectos causados por cambios sin disciplina en los modelos del sistema. Por ejemplo, es un error común cambiar una interfaz de subsistema sin notificarlo a todos los desarrolladores de los componentes que lo llaman. La administración de la configuración puede asegurar que si los modelos de análisis y el código se están volviendo inconsistentes entre sí se les notificará a los analistas e implementadores. La suposición que hay tras la administración de la configuración es que el sistema contiene mucho menos defectos si se controla el cambio.

La **verificación** trata de encontrar defectos antes de cualquier ejecución del sistema. La verificación es posible en casos específicos, como la verificación del kernel de un sistema operativo pequeño [Walker *et al.*, 1980]. Sin embargo, la verificación tiene sus límites. No está en un estado lo bastante maduro como para que pueda aplicarse para asegurar la calidad de grandes sistemas complejos. Además, supone que los requerimientos son correctos, lo cual rara vez sucede: supongamos que necesitamos verificar una operación y que conocemos las condiciones previas y posteriores de la operación. Si podemos verificar que la operación proporciona una transformación tal que antes de la ejecución de la operación la condición previa es cierta y que después de la

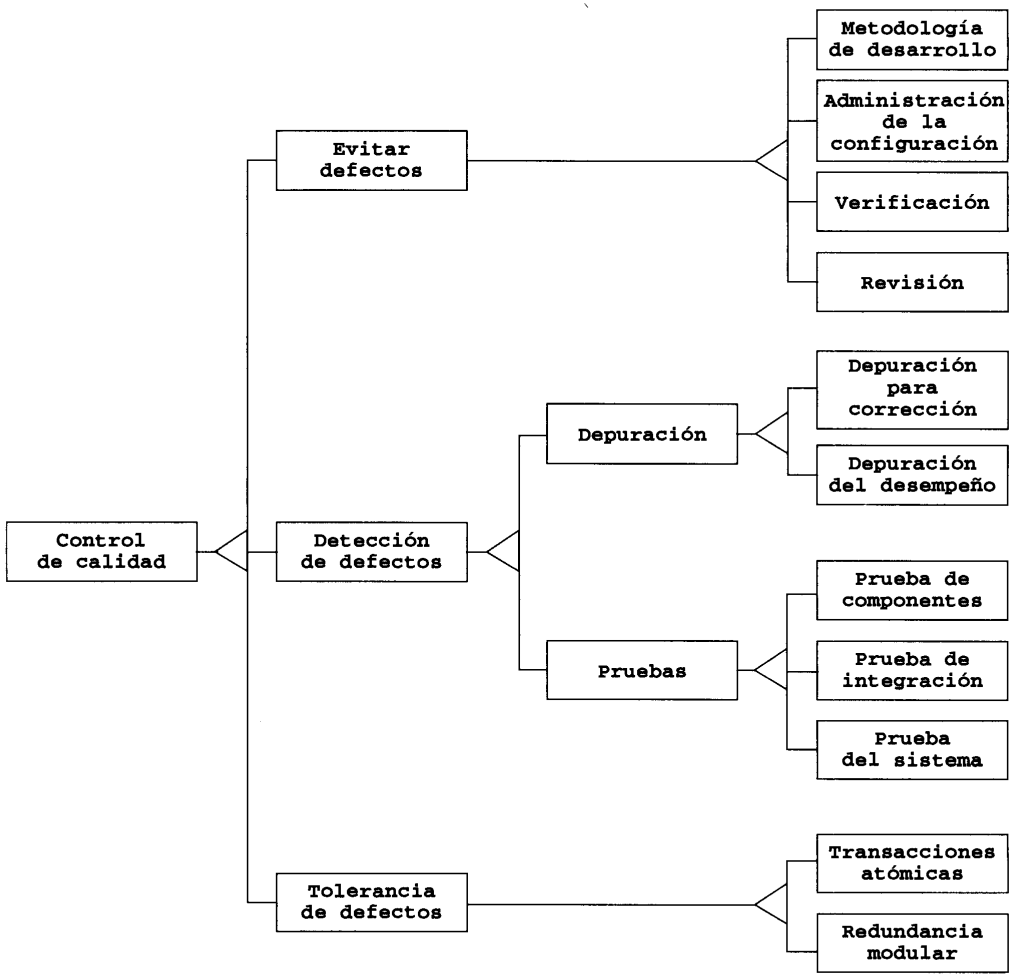


Figura 9-1 Taxonomía de las actividades del control de calidad (diagrama de clase UML).

ejecución se mantiene la condición posterior, entonces la operación está implementada en forma correcta; esto es, está verificada. Sin embargo, a veces olvidamos partes importantes de la condición previa o de la posterior. Por ejemplo, puede ser que las condiciones posteriores no establezcan que el tren debe permanecer en el suelo mientras pasa de la vía 1 a la vía 2. Por último, la verificación sólo aborda los defectos algorítmicos: no puede manejar los defectos mecánicos de la máquina virtual. Para la verificación debemos suponer que la máquina virtual sobre la que ejecuta el sistema está ejecutando en forma correcta y no cambia durante la prueba.

Una **revisión** es la inspección manual de algunos o todos los aspectos del sistema sin ejecutar, de hecho, el sistema. Hay dos tipos de revisiones: ensayo e inspección. En un **ensayo** de

código el desarrollador presenta de modo informal la API, el código y la documentación asociada de los componentes ante el equipo de revisión. Éste hace comentarios sobre la correspondencia entre los diseños de análisis y objetos y el código usando casos de uso y escenarios de la fase de análisis. Una **inspección** es similar a un ensayo, pero la presentación de la unidad es formal. De hecho, en una inspección de código no se permite que el desarrollador presente los artefactos (modelo, código y documentación). Esto lo hace el equipo de revisión, el cual es responsable de la revisión de la interfaz y el código del componente contra los requerimientos. También revisa que los algoritmos sean eficientes con respecto a los requerimientos no funcionales. Por último, revisa los comentarios acerca del código y los compara con el código mismo para encontrar comentarios imprecisos e incompletos. El desarrollador sólo está presente cuando la revisión necesita aclaraciones acerca de la definición y uso de las estructuras de datos o algoritmos.

Los revisores de código han probado ser muy efectivos en la detección de errores. En algunos experimentos, hasta 85% de todos los defectos identificados se han encontrado en las revisiones [Fagan, 1976], [Jones, 1977].

9.2.3 Técnicas para la detección de defectos

Las técnicas para la detección de defectos ayudan a encontrar defectos en los sistemas pero no tratan de recuperar las fallas que causan. En general, las técnicas para la detección de defectos se aplican durante el desarrollo, pero en algunos casos también se usan después del lanzamiento del sistema. Las cajas negras de los aviones que registran los últimos minutos de un vuelo antes de estrellarse son un ejemplo de una técnica para detección de defectos. Distinguimos dos tipos de técnicas para la detección de defectos: la depuración y las pruebas.

La **depuración** asume que los defectos pueden encontrarse iniciando a partir de una falla no planeada. El desarrollador mueve al sistema a través de una sucesión de estados hasta que a final de cuentas llega al estado erróneo y lo identifica. Una vez que se ha identificado este estado es necesario determinar el defecto algorítmico o mecánico que lo causa. Hay dos tipos de depuración: el objetivo de la **depuración para corrección** es encontrar cualquier desviación entre los requerimientos no funcionales observados y los especificados. La **depuración del desempeño** trata la desviación entre los requerimientos no funcionales observados y los especificados, como el tiempo de respuesta.

La **prueba** es una técnica de detección de defectos que trata de crear fallas o errores en forma planeada. Esto permite que el desarrollador detecte fallas en el sistema antes de que sea lanzado al cliente. Observe que esta definición de prueba implica que una prueba satisfactoria es la que identifica errores. Usaremos esta definición a lo largo de las fases del desarrollo. Otra definición de la prueba que se usa con frecuencia es que "demuestra que no se presentan errores". Usaremos esta definición sólo después del desarrollo del sistema, cuando tratemos de demostrar que el sistema entregado satisface los requerimientos funcionales y los no funcionales.

Si usáramos esta definición todo el tiempo tenderíamos a seleccionar datos de prueba que tuvieran una probabilidad muy baja de causar que fallara el programa. Por otro lado, si el objetivo es demostrar que un programa tiene errores, tendemos a buscar datos de prueba con una alta probabilidad de localización de errores. La característica de un buen modelo de prueba es que contiene casos de prueba que identifican errores. Debe probarse cada entrada que se le pueda dar al sistema, ya que de no hacerlo hay probabilidades de que no se detecten los

defectos. Por desgracia, tal enfoque requiere tiempos de prueba largos en extremo, incluso para sistemas pequeños.

La figura 9-2 muestra un resumen de las actividades de prueba:

- Las **pruebas unitarias** tratan de encontrar defectos en los objetos participantes o en los subsistemas, o en ambos, con respecto a los casos de uso del modelo de casos de uso.
- Las **pruebas de integración** son las actividades relacionadas con la localización de defectos cuando se prueban juntos componentes que se han probado de manera individual; por ejemplo, los subsistemas descritos en la descomposición en subsistemas, mientras se ejecutan los casos de uso y escenarios del RAD.
    - Las **pruebas de estructura del sistema** son la culminación de las pruebas de integración que involucran a todos los componentes del sistema. Las pruebas de integración y de estructura explotan el conocimiento del SDD usando una estrategia descrita en el Plan de pruebas (TP, por sus siglas en inglés).
- Las **pruebas de sistema** prueban todos los componentes juntos, vistos como un solo sistema, para identificar errores con respecto a los escenarios del enunciado del problema y a los objetivos de los requerimientos y del diseño identificados en el análisis y el diseño del sistema, respectivamente:
    - Las **pruebas funcionales** prueban los requerimientos del RAD y, si se tiene disponible, del manual de usuario.
    - Las **pruebas del desempeño** revisan los requerimientos no funcionales y objetivos de diseño adicionales del SDD. Observe que los desarrolladores realizan las pruebas funcionales y de desempeño.
    - Las **pruebas de aceptación** y de **instalación** revisan los requerimientos contra el acuerdo del proyecto y deben ser realizadas por el cliente, con apoyo de los desarrolladores si es necesario.

9.2.4 Técnicas para la tolerancia de defectos

Si no podemos impedir los errores debemos aceptar el hecho de que el sistema lanzado contendrá defectos que darán lugar a fallas. La tolerancia de defectos es la recuperación de una falla mientras el sistema está ejecutando. Permite que un sistema se recupere de la falla de un componente pasando la información del estado erróneo de regreso a los componentes que llamaron, suponiendo que alguno de ellos sepa qué hacer en este caso. Los sistemas de bases de datos proporcionan transacciones atómicas para recuperarse de una falla durante una secuencia de acciones que necesitan hacerse todas o ninguna. La redundancia modular está sustentada en la suposición de que las fallas del sistema se basan, por lo general, en fallas de componentes. Los sistemas con redundancia modular se construyen asignando más de un componente para que realice la misma operación. Las cinco computadoras a bordo del transbordador espacial son un ejemplo de un sistema modular redundante. El sistema puede continuar aunque falle un componente, debido a que los demás componentes todavía están realizando la funcionalidad requerida. El tratamiento de las técnicas de tolerancia de defectos es importante para los sistemas muy confiables, pero rebasa el alcance de este libro. Una cobertura excelente del tema la proporcionan [Siewiorek y Swarz, 1992].

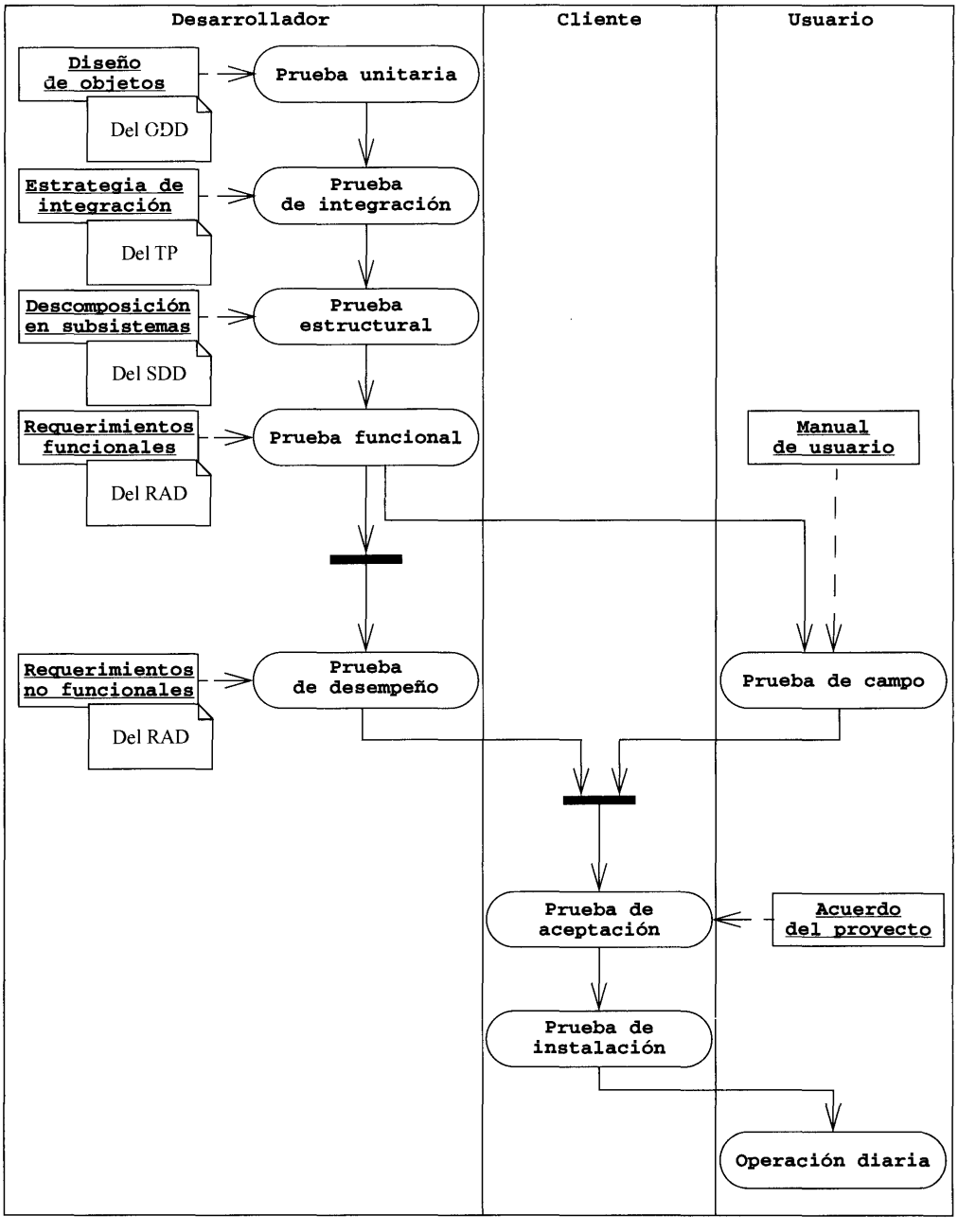


Figura 9-2 Actividades de prueba y sus productos de trabajo relacionados (diagrama de actividad UML). Los carriles indican quién ejecuta la prueba.

9.3 Concepto de las pruebas

En esta sección presentamos los elementos del modelo usado durante las pruebas (figura 9-3):

- Un **componente** es una parte del sistema que puede aislarse para la prueba. Un componente puede ser un objeto, un grupo de objetos o uno o más subsistemas.
- Un **defecto** es un error de diseño o codificación que puede causar un comportamiento anormal de un componente.
- Un **error** es la manifestación de un defecto durante la ejecución del sistema.
- Una **falla** es una desviación entre la especificación de un componente y su comportamiento. Una falla es producida por uno o más errores.
- Un **caso de prueba** es un conjunto de entradas y resultados esperados que ejercita a un componente con el propósito de causar fallas y detectar defectos.
- Un **stub de prueba** es una implementación parcial de componentes de los cuales depende el componente probado. Un **manejador de pruebas** es una implementación parcial de un componente que depende del componente probado. Los stubs y manejadores de pruebas permiten que los componentes se aíslen del resto del sistema para las pruebas.
- Una **corrección** es un cambio a un componente. El propósito de una corrección es reparar un defecto. Tome en cuenta que la corrección puede introducir nuevos defectos [Brooks, 1975].

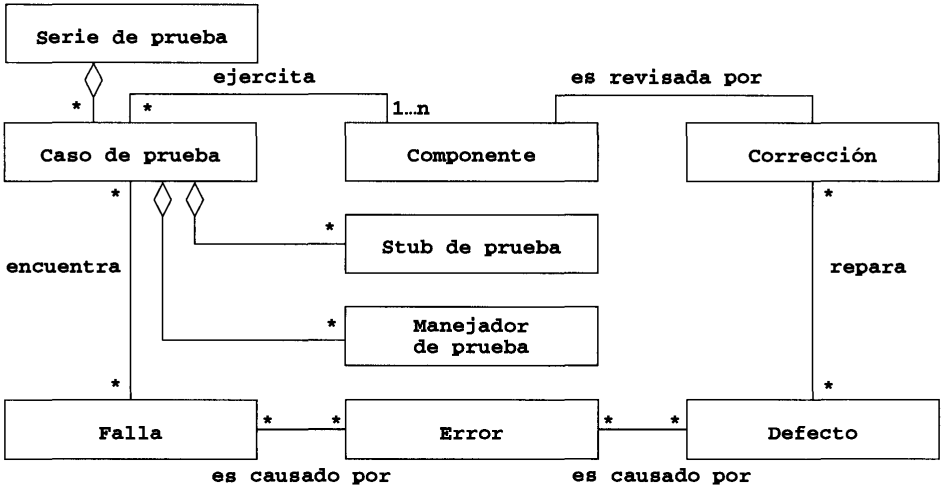


Figura 9-3 Elementos del modelo usado durante las pruebas (diagrama de clase UML).

9.3.1 Defectos, errores y fallas

Con la comprensión inicial de los términos de las definiciones de la sección 9.3, veamos la figura 9-4.

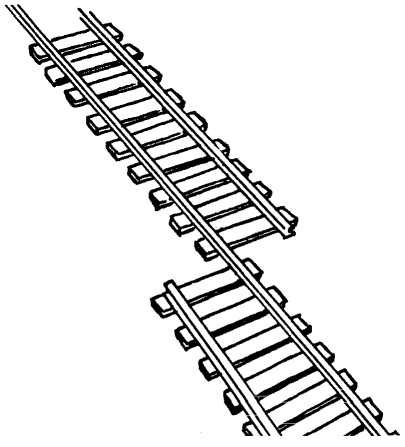


Figura 9-4 Un ejemplo de un defecto.

¿Qué vemos? La figura 9-4 muestra un par de vías que no están alineadas. Si imaginamos un tren corriendo sobre la vía, éste se descarrilaría (falla). Sin embargo, la figura no presenta, de hecho, una falla ni un error ni un defecto. No muestra una falla debido a que no se ha especificado el comportamiento esperado y tampoco hay un comportamiento observado. La figura 9-4 tampoco muestra un error, debido a que eso significaría que el sistema está en un estado cuyo procesamiento adicional conducirá a una falla. Aquí sólo vemos vías y no se muestra ningún tren en movimiento. Para hablar sobre errores y fallas necesitamos comparar el comportamiento deseado (descrito en el caso de uso en el RAD) con el comportamiento observado (descrito en el caso de prueba). Supongamos que tenemos un caso de uso con un tren moviéndose desde la vía superior izquierda hacia la inferior derecha (figura 9-5).

Nombre del caso de uso	`ManejarTren`
Actor participante	`OperadorTren`
Condición inicial	El `OperadorTren` oprime el botón "ArrancarTren" en el panel de control.
Flujo de eventos	1. El tren comienza a moverse sobre la vía 1. 2. El tren pasa a la vía 2.
Condición final	El tren está corriendo sobre la vía 2.
Requerimientos especiales	Ninguno.

Figura 9-5 Caso de uso `ManejarTren` que especifica el comportamiento esperado del tren.

Identificador del caso de prueba	`ManejarTren`
Ubicación de la prueba	`http://www12.in.tum.de/TrainSystem/test-cases/test1`
Característica a probar	Operación continua del motor durante 5 segundos.
Criterio pasa o falla de la característica	La prueba pasa si el tren se mueve durante 5 segundos y cubre la longitud de al menos dos vías.
Medios de control	1. Se llama al método `ArrancarTren()` mediante un manejador de prueba `ArrancarTren` (contenido en el mismo directorio que la prueba `ManejarTren`).
Datos	2. La dirección del viaje y su duración se leen desde un archivo de entrada `http://www12.in.tum.de/TrainSystem/test-cases/input` 3. Si la depuración está puesta a CIERTO el caso de prueba enviará los siguientes mensajes de sistema "Entra a vía n, sale de vía n" para cada n, donde n es el número de la vía actual.
Procedimiento de prueba	La prueba se inicia haciendo doble clic en el caso de prueba en la ubicación especificada. La prueba ejecutará sin intervención adicional hasta que termine. La prueba no deberá durar más de 7 segundos.
Requerimientos especiales	Se necesita el *stub* de prueba `Motor` para la implementación de la vía.

Figura 9-6 El caso de prueba `ManejarTren` para el caso de uso descrito en la figura 9-5.

Luego podemos continuar para derivar un caso de prueba que mueva al tren desde el estado descrito en la condición inicial del caso de uso hasta un estado en donde se descarrile, precisamente cuando salga de la vía superior (figura 9-6).

En otras palabras, cuando ejecutemos este caso de prueba podremos demostrar que el sistema contiene un defecto. Tome en cuenta que el estado actual de la figura 9-7 es erróneo, pero no muestra una falla.

La falta de alineación de las vías puede ser resultado de una mala comunicación entre los equipos de desarrollo (cada vía tiene que ser colocada en su posición por un equipo) o debido a una implementación errónea de la especificación por parte de alguno de los equipos (figura 9-8). Estos son ejemplos de defectos algorítmicos. Es probable que usted ya esté familiarizado con muchos otros defectos algorítmicos que se introducen durante la fase de implementación. Por ejemplo, "salir muy pronto del ciclo", "salir muy tarde del ciclo", "probar la condición equivocada", "olvidar la inicialización de una variable" son todos defectos algorítmicos específicos de la implementación. Los defectos algorítmicos también pueden suceder durante el análisis y diseño del sistema. Los problemas de tensiones y sobrecargas, por ejemplo, son defectos algorítmicos específicos del diseño que conducen a fallas cuando las estructuras de datos se llenan más allá de su capacidad especificada. Los errores de producción y desempeño son posibles cuando un sistema no se desempeña a la velocidad especificada por los requerimientos no funcionales.

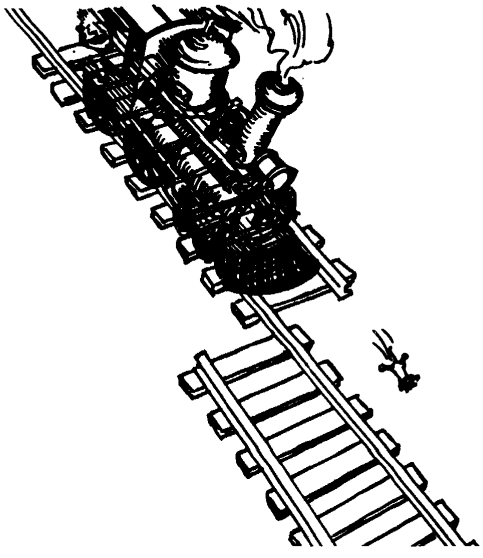


Figura 9-7 Un ejemplo de un error.

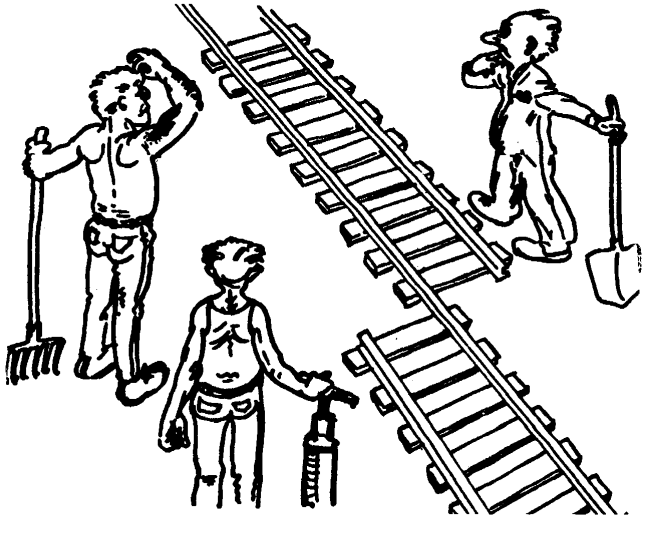


Figura 9-8 Un defecto puede tener una causa algorítmica.

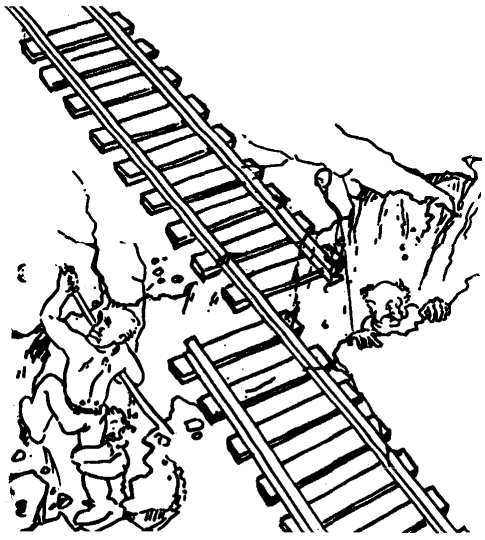


Figura 9-9 Un defecto puede tener una causa mecánica, como un sismo.

Aunque las vías estuvieran implementadas de acuerdo con la especificación del RAD, podrían desalinearse durante la operación diaria, por ejemplo, si hubiera un sismo que moviera el suelo subyacente (figura 9-9).

Un error en la máquina virtual de un sistema de software es otro ejemplo de una falla mecánica: aunque los desarrolladores hubieran hecho la implementación correcta, es decir, hubieran hecho en forma correcta la correspondencia entre el modelo de objetos y el código, el comportamiento observado todavía puede desviarse con respecto al especificado. En los proyectos de ingeniería concurrentes, por ejemplo, donde se desarrolla el hardware en paralelo con el software, no podemos hacer siempre la suposición de que la máquina virtual ejecuta como se especificó. Otros ejemplos de fallas mecánicas son las fallas de corriente. Observe la relatividad de los términos "defecto" y "falla" con respecto a un componente particular del sistema: la falla en un componente del sistema (el sistema de corriente eléctrica) es el defecto mecánico que puede conducir a una falla en otro componente del sistema (el sistema de software).

9.3.2 Casos de prueba

Un **caso de prueba** es un conjunto de datos de entrada y resultados esperados que ejercitan a un componente con el propósito de causar fallas y detectar defectos. Un caso de prueba tiene cinco atributos: `nombre`, `ubicación`, `entrada`, `oráculo` y `bitácora` (tabla 9-1). El `nombre` del caso de prueba permite que quien hace la prueba distinga entre casos de prueba diferentes. Una heurística para nombrar a los casos de prueba es derivar el nombre a partir del requerimiento que se está probando o del componente que está a prueba. Por ejemplo, si se está probando un caso de uso `Depósito()`, tal vez quiera nombrar al caso de prueba *`Prueba_Depósito`*. Si un caso de prueba involucra a dos componentes A y B, un buen nombre sería *`Prueba_AB`*. El atributo

Tabla 9-1 Atributos de la clase `CasoPrueba`.

Atributos	Descripción
`nombre`	Nombre del caso de prueba
`ubicación`	Nombre de ruta completo del ejecutable
`entrada`	Datos de entrada o comandos
`oráculo`	Resultados esperados de la prueba contra los que se compara la salida de la prueba
`bitácora`	Salida producida por la prueba

`ubicación` describe dónde puede encontrarse al caso de prueba. Debe ser un nombre de ruta o un URL hacia el ejecutable del programa de prueba y sus datos de entrada.

La `entrada` describe el conjunto de datos de entrada o comandos a dar por el actor del caso de prueba (que puede ser la persona que hace la prueba o un manejador de la prueba). El comportamiento esperado del caso de prueba es la secuencia de datos de salida o comandos que debe producir una ejecución correcta de la prueba. El comportamiento esperado lo describe el atributo `oráculo`. La `bitácora` es un conjunto de correlaciones con indicaciones de tiempo del comportamiento observado contra el esperado para varias corridas de prueba.

Una vez que se han identificado y descrito los casos de prueba se identifican las relaciones entre ellos. Se usan la agregación y las asociaciones `precede` para describir las relaciones entre los casos de prueba. La agregación se usa cuando un caso de prueba puede descomponerse en un conjunto de subpruebas. Dos casos de prueba están relacionados con la asociación `precede` cuando un caso de prueba debe preceder a otro.

La figura 9-10 muestra un modelo de prueba donde `PruebaA` debe preceder a `PruebaB` y `PruebaC`. Por ejemplo, `PruebaA` consiste en `PruebaA1` y `PruebaA2`, lo que significa que una vez que se han realizado `PruebaA1` y `PruebaA2` ya se ha probado `PruebaA` y no hay una prueba separada para `PruebaA`. Un buen modelo de prueba tiene la menor cantidad de asociaciones posibles, debido a que las pruebas que no están asociadas entre ellas pueden ejecutarse en forma independiente. Esto permite que quien hace la prueba la agilice, si se dispone de los recursos necesarios para la

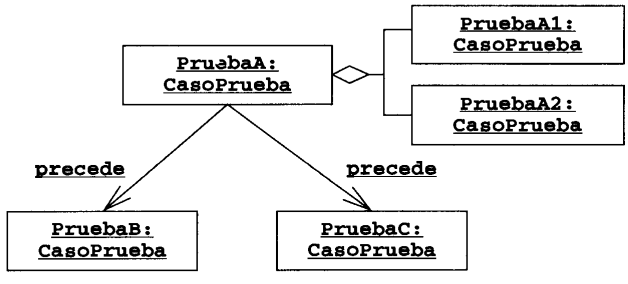


Figura 9-10 Modelo de prueba con casos de prueba. La `PruebaA` consiste en dos pruebas, `PruebaA1` y `PruebaA2`. `PruebaB` y `PruebaC` pueden probarse de manera independiente, pero sólo después de que se haya realizado `PruebaA`.

prueba. En la figura 9-10, `PruebaB` y `PruebaC` pueden probarse en paralelo, debido a que no hay ninguna relación entre ellas.

Los casos de prueba se clasifican en pruebas de caja negra y de caja blanca, dependiendo del aspecto del modelo del sistema que se prueba. Las **pruebas de caja negra** se enfocan en el comportamiento de entrada/salida del componente. Las pruebas de caja negra no manejan los aspectos internos del componente ni el comportamiento o estructura de los componentes. Las **pruebas de caja blanca** se enfocan en la estructura interna del componente. Una prueba de caja blanca se asegura que, sin importar el comportamiento de entrada/salida particular, se pruebe cada estado del modelo dinámico del objeto y cada interacción entre los objetos. En consecuencia, las pruebas de caja blanca van más allá que las de caja negra. De hecho, la mayoría de las pruebas de caja blanca requieren datos de entrada que no pueden derivarse sólo a partir de una descripción de los requerimientos funcionales. Las pruebas unitarias combinan ambas técnicas: las de caja negra para probar la funcionalidad del componente y las de caja blanca para probar los aspectos estructurales y dinámicos del componente.

9.3.3 Stubs y manejadores de pruebas

La ejecución de casos de prueba sobre componentes solos o combinaciones de componentes requiere que el componente a probar se aísle del resto del sistema. Los manejadores y *stubs* de prueba se usan para sustituir las partes faltantes del sistema. Un **manejador de prueba** simula la parte del sistema que llama al componente a probar. Un manejador de prueba pasa al componente las entradas de prueba identificadas en el análisis del caso de prueba y muestra los resultados.

Un ***stub* de prueba** simula a los componentes que son llamados por el componente a probar. El *stub* de prueba debe proporcionar la misma API que el método del componente simulado, y debe regresar un valor que se apegue al tipo de resultado regresado por el tipo de firma del método. Tome en cuenta que debe haber congruencia entre las interfaces de todos los componentes. Si cambia la interfaz de un componente también deben cambiar los manejadores y *stubs* de prueba correspondientes.

Puede usarse el patrón Puente para la implementación de los *stubs* de prueba. La figura 9-11 muestra un patrón Puente donde un subsistema `Interfaz de usuario` accede al subsistema `Base de datos` que todavía no está listo para la prueba. Al separar la interfaz de `Base de datos`

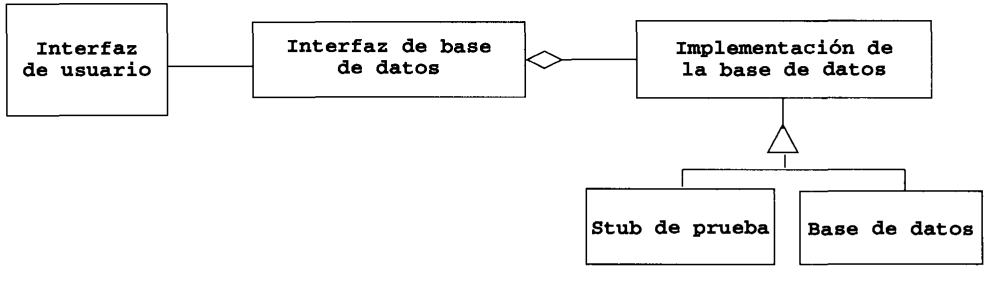


Figura 9-11 Uso del patrón de diseño Puente para hacer la interfaz de un componente que todavía no se ha terminado, o que no se conoce o no se tiene disponible durante la prueba de otro componente (diagrama de clase UML).

de la implementación de `Base de datos` podemos proporcionar primero un *stub* de prueba que responda a las peticiones y después reemplazarlo con la implementación real del subsistema `Base de datos`. Observe que el uso del patrón Puente permite el reemplazo del *stub* por el componente llamado una vez que está probado sin tener que volver a compilar el componente que llama.

La implementación de los *stubs* de prueba es una tarea no trivial. No es suficiente escribir un *stub* de prueba que tan sólo escriba un mensaje que diga que se llamó al *stub* de prueba. En la mayoría de las situaciones, cuando un componente A llama al componente B, A está esperando que B realice algún trabajo, el que luego es regresado como un conjunto de parámetros de resultado. Si el *stub* de prueba no simula este comportamiento fallará A, no a causa de un defecto de A sino debido a que el *stub* de prueba no simula en forma correcta a B.

Pero no siempre es suficiente proporcionar un valor de retorno. Por ejemplo, si un *stub* de prueba regresa siempre el mismo valor, puede ser que no regrese el valor esperado por el componente que llama en un escenario particular. Esto puede producir resultados confusos y hasta puede dar lugar a la falla del componente que llama, aunque esté implementado en forma correcta. A menudo hay un compromiso entre implementar *stubs* de prueba precisos y sustituir los *stubs* de prueba por el componente real.

9.3.4 Correcciones

Una vez que se han ejecutado las pruebas y se han detectado las fallas, los desarrolladores cambian el componente para eliminar los defectos sospechosos. Una **corrección** es un cambio a un componente con el propósito de reparar un defecto. Las correcciones pueden ir desde una simple modificación de un solo componente hasta el rediseño completo de una estructura de datos o un subsistema. En todos los casos existe una probabilidad alta de que el desarrollador introduzca nuevos defectos en el componente revisado [Brooks, 1975]. Pueden usarse varias técnicas para manejar estos defectos:

- El *seguimiento del problema* incluye la documentación de cada falla, error y defecto detectado, su corrección y las revisiones de los componentes involucrados en el cambio. Junto con la administración de la configuración, el seguimiento de problemas permite que los desarrolladores estrechen la búsqueda de nuevos defectos. En el capítulo 10, *Administración de la configuración del software*, describimos con mayor detalle la administración de la configuración.
- Las *pruebas de regresión* incluyen volver a ejecutar todas las pruebas anteriores después del cambio. Esto asegura que no se haya afectado la funcionalidad que operaba antes de la corrección. Las pruebas de regresión son importantes en los métodos orientados a objetos, los cuales piden un proceso de desarrollo iterativo. Esto requiere que las pruebas se inicien pronto y que las series de prueba se mantengan conforme evoluciona el diseño. Por desgracia, las pruebas de regresión son costosas, en especial cuando parte de las pruebas no está automatizada. En la siguiente sección describimos con mayor detalle las pruebas de regresión.
- El *mantenimiento de la fundamentación* incluye la documentación de la fundamentación de los cambios y su relación con la fundamentación del componente revisado. El mantenimiento de la fundamentación permite que los desarrolladores eviten la introducción de nuevos defectos inspeccionando las suposiciones que se usaron para construir el componente. En el capítulo 8, *Administración de la fundamentación*, describimos el mantenimiento de la fundamentación.

A continuación describimos con mayor detalle las actividades de las pruebas que conducen a la creación de los casos de prueba, su ejecución y el desarrollo de correcciones.

9.4 Actividades de las pruebas

En esta sección describimos las actividades de las pruebas, las cuales incluyen:

- **Inspección de componentes,** que encuentran defectos en componentes individuales mediante la inspección manual de su código fuente.
- **Pruebas unitarias,** que encuentran defectos aislando un componente individual usando *stubs y* manejadores de prueba y ejercitando al componente usando un caso de prueba.
- **Pruebas de integración,** que encuentran defectos integrando varios componentes.
- **Pruebas del sistema,** que se enfocan en el sistema completo, sus requerimientos funcionales y no funcionales y su ambiente de destino.

9.4.1 Inspección de componentes

Las inspecciones encuentran defectos en un componente revisando su código fuente en una reunión formal. Las inspecciones pueden realizarse antes o después de la prueba unitaria. El primer proceso de inspección estructurado fue el método de inspección de Michael Fagan [Fagan, 1976]. La inspección la realiza un equipo de desarrolladores, incluyendo al autor del componente, un moderador que facilita el proceso y uno o más revisores que encuentran defectos en el componente. El método de inspección de Fagan consta de cinco pasos:

- **Panorama.** El autor del componente presenta en forma breve el propósito y alcance del componente y los objetivos de la inspección.
- **Preparación.** Los revisores se familiarizan con la implementación del componente.
- **Reunión de inspección.** Un lector parafrasea el código fuente del componente y el equipo de inspección plantea problemas relacionados con el componente. Un moderador mantiene la reunión en el tema.
- **Reparación.** El autor revisa el componente.
- **Seguimiento.** El moderador revisa la calidad de la reparación y determina si es necesario volver a inspeccionar el componente.

Los pasos críticos de este proceso son la fase de preparación y la reunión de inspección. Durante la fase de preparación los revisores se familiarizan con el código fuente y todavía no se enfocan en la localización de defectos. Durante la reunión de inspección el lector parafrasea el código fuente; esto es, lee cada enunciado de código y explica lo que debe hacer el enunciado. Entonces los revisores plantean asuntos si creen que hay un defecto. La mayor parte del tiempo se dedica a debatir si está presente o no un defecto, pero en este momento no se exploran las soluciones para reparar el defecto. Durante la fase de panorama de la inspección el autor establece los objetivos de la inspección. Además de encontrar defectos, también puede pedirse a los revisores que busquen desviaciones con respecto a los estándares de codificación o ineficiencias.

Las inspecciones de Fagan se perciben, por lo general, como consumidoras de tiempo debido a lo largo de la fase de preparación y la reunión de inspección. La efectividad de una revisión también depende de la preparación de los revisores. David Parnas propuso un proceso de inspección revisado, la revisión del diseño activa, en donde no hay reunión de inspección que incluya a todos los miembros del equipo de inspección [Parnas y Weiss, 1985]. En vez de ello, se les pide a los revisores que encuentren fallas durante la fase de preparación. Al final de la fase de preparación cada revisor llena un cuestionario que prueba su comprensión del componente. Luego el autor se reúne en forma individual con cada revisor para recolectar la retroalimentación sobre el componente.

Las inspecciones de Fagan y las revisiones de diseño activas han mostrado ser más efectivas que las pruebas para descubrir defectos. Tanto las pruebas como las inspecciones se usan en proyectos donde es crítica la seguridad, ya que tienden a encontrar diferentes tipos de defectos.

9.4.2 Prueba unitaria

La prueba unitaria se enfoca en los bloques de construcción del sistema de software; esto es, los objetos y subsistemas. Hay tres motivaciones tras el enfoque en los componentes. Primero, la prueba unitaria reduce la complejidad de las actividades de prueba generales, permitiéndonos enfocarnos en unidades más pequeñas del sistema. Segundo, la prueba unitaria facilita resaltar y corregir defectos, ya que están involucrados pocos componentes. Tercero, la prueba unitaria permite el paralelismo en las actividades de prueba; esto es, cada componente puede probarse en forma independiente de los demás.

Los candidatos específicos para las pruebas unitarias se escogen del modelo de objetos y la descomposición en subsistemas del sistema. En principio, deben probarse todos los objetos desarrollados durante el proceso de desarrollo, pero no siempre es posible por razones de tiempo y presupuesto. El conjunto mínimo de objetos a probar debe ser el de aquellos que participan en los casos de uso. Los subsistemas deben probarse después de que se han probado de manera individual cada uno de los objetos y clases que están en el subsistema.

Los subsistemas existentes, que se reutilizan o compran, deben tratarse como componentes con estructura interna desconocida. Esto se aplica, en particular, a subsistemas comerciales, en los que no se conoce la estructura interna o no está a disposición del desarrollador.

Se han inventado muchas técnicas para la prueba unitaria. A continuación describimos las más importantes: prueba de equivalencia, prueba de frontera, prueba de ruta y prueba basada en estado.

Prueba de equivalencia

La **prueba de equivalencia** es una técnica de prueba de caja negra que minimiza la cantidad de casos de prueba. Las entradas posibles se reparten en clases de equivalencia y se selecciona un caso de prueba para cada clase. La suposición de la prueba de equivalencia es que los sistemas se comportan, por lo general, en forma similar para todos los miembros de una clase. Para probar el comportamiento asociado con una clase de equivalencia sólo necesitamos probar a un miembro de la clase. La prueba de equivalencia consiste en dos pasos: identificación de las clases de equivalencia y selección de las entradas de prueba. Se usan los siguientes criterios para determinar las clases de equivalencia.

- **Cobertura.** Cada entrada posible pertenece a una de las clases de equivalencia.
- **Inconexión.** Ninguna entrada pertenece a más de una clase de equivalencia.
- **Representación.** Si la ejecución muestra un error cuando se usa como entrada un miembro particular de una clase de equivalencia, el mismo error puede detectarse usando como entrada a cualquier otro miembro de la clase.

Para cada clase de equivalencia se seleccionan, al menos, dos partes de datos: una entrada típica, que ejercita el caso común, y una entrada inválida que ejercita las capacidades del componente para el manejo de excepciones. Después de que se han identificado todas las clases de equivalencia tiene que identificarse una entrada de prueba para cada clase que cubra la clase de equivalencia. Si hay una posibilidad de que no todos los elementos de la clase de equivalencia estén cubiertos por la entrada de prueba, la clase de equivalencia debe dividirse en clases de equivalencia más pequeñas y deben identificarse entradas de prueba para cada una de las nuevas clases.

Por ejemplo, considere un método que regresa la cantidad de días de un mes dando el mes y el año (vea la figura 9-12). El mes y el año se especifican como enteros. Por convención, 1 representa el mes de enero, 2 el de febrero y así en forma sucesiva. El rango de entradas válidas para el año es desde `0` hasta `intMax`.

```
class MiCalendarioGregoriano {
...
    public static int obtenerNumDíasEnMes(int mes, int año) {…}
...
}
```

Figura 9-12 Interfaz para un método que calcula el número de días de un mes dado (Java). El método `obtenerNumDíasEnMes()` toma dos parámetros, un mes y un año, especificados como enteros.

Encontramos tres clases de equivalencia para el parámetro mes: meses con 31 días (es decir, `1, 3, 5, 7, 8, 10, 12`), meses con 30 días (es decir, `4, 6, 9, 11`) y febrero, que puede tener 28 o 29 días. Los enteros no positivos y los enteros mayores a `12` son valores inválidos para los parámetros de mes. En forma similar, encontramos dos clases de equivalencia para el año: años bisiestos y no bisiestos. Por la especificación, los enteros negativos son valores inválidos para el año. Primero seleccionamos un valor válido para cada parámetro y clase de equivalencia (por ejemplo, febrero, junio, julio, 1901 y 1904). Tomando en cuenta que el valor de retorno del método `obtenerNumDíasEnMes()` depende de ambos parámetros, combinamos estos valores para probar por interacción, dando como resultado las seis clases de equivalencia que se muestran en la tabla 9-2.

Prueba de frontera

La **prueba de frontera** es un caso especial de la prueba de equivalencia y se enfoca en las condiciones de frontera de las clases de equivalencia. En vez de seleccionar cualquier elemento de la clase de equivalencia, la prueba de frontera requiere que los elementos se seleccionen de los "extremos" de la clase de equivalencia. La suposición que hay tras la prueba de frontera es

Tabla 9-2 Clases de equivalencia y entradas válidas seleccionadas para la prueba del método `obtenerNumDíasEnMes()`.

Clase de equivalencia	Valor para la entrada de mes	Valor para la entrada de año
Meses con 31 días, años no bisiestos	`7 (julio)`	`1901`
Meses con 31 días, años bisiestos	`7 (julio)`	`1904`
Meses con 30 días, años no bisiestos	`6 (junio)`	`1901`
Meses con 30 días, años bisiestos	`6 (junio)`	`1904`
Meses con 28 o 29 días, años no bisiestos	`2 (febrero)`	`1901`
Meses con 28 o 29 días, años bisiestos	`2 (febrero)`	`1904`

que los desarrolladores a menudo olvidan los casos especiales en la frontera de las clases de equivalencia (por ejemplo, 0, cadenas vacías, año 2000).

En nuestro ejemplo, el mes de febrero presenta varios casos de frontera. En general, los años que son múltiplos de 4 son años bisiestos. Sin embargo, los años que son múltiplos de 100 no son años bisiestos, a menos que también sean múltiplos de 400. Por ejemplo, 2000 es un año bisiesto, pero 1900 no lo fue. Los años 1900 y 2000 son buenos casos de frontera que debemos probar. Otros casos de frontera incluyen los meses 0 y 13, que son las fronteras de la clase de equivalencia inválida. La tabla 9-3 muestra los casos de frontera adicionales que seleccionamos para el método `obtenerNumDíasEnMes()`.

Tabla 9-3 Casos de frontera adicionales seleccionados para el método `obtenerNumDíasEnMes()`.

Clase de equivalencia	Valor para la entrada de mes	Valor para la entrada de año
Años bisiestos divisibles entre 400	`2 (febrero)`	`2000`
Años no bisiestos divisibles entre 100	`2 (febrero)`	`1900`
Meses no positivos inválidos	`0`	`1291`
Meses positivos inválidos	`13`	`1315`

Una desventaja de las pruebas de la clase de equivalencia y de frontera es que estas técnicas no exploran combinaciones de datos de entrada de prueba. En muchos casos, un programa falla debido a que una combinación de determinados valores causa el error. Las pruebas de causa-efecto atacan este problema estableciendo relaciones lógicas entre la entrada y las salidas o entradas y transformaciones. A las entradas se les llama causas y a la salida o las transformaciones se les llama efectos. La técnica se basa en la premisa de que el comportamiento de entrada/salida puede transformarse hacia una función booleana. Para detalles sobre esta técnica y otra llamada suposición del error deberá consultarse la literatura sobre las pruebas (por ejemplo, [Myers, 1979]).

Prueba de ruta

La **prueba de ruta** es una técnica de prueba de caja blanca que identifica fallas en la implementación del componente. La suposición que hay tras la prueba de ruta es que ejercitando todas las rutas posibles del código, al menos una vez, la mayoría de los defectos producirá fallas. La identificación de las rutas requiere conocimiento del código fuente y las estructuras de datos.

El punto inicial para la prueba de ruta es el diagrama de flujo. Un diagrama de flujo consiste en nodos que representan bloques ejecutables y asociaciones que representan el flujo de control. Un bloque básico está formado por varias instrucciones entre dos decisiones. Un diagrama de flujo puede construirse a partir del código de un componente estableciendo la correspondencia de las instrucciones de decisión (por ejemplo, instrucciones if, ciclos while) con las líneas de nodos. Las instrucciones entre cada punto de decisión (por ejemplo, bloque then, bloque else) se hacen corresponder con otros nodos. Las asociaciones entre cada nodo representan relaciones de precedencia. La figura 9-14 muestra una implementación de ejemplo del método `obtenerNumDíasEnMes()`. La figura 9-13 muestra el diagrama de flujo correspondiente como un diagrama de actividad UML. En este ejemplo modelamos los puntos de decisión con ramificaciones UML, los bloques con estados de acción UML y el flujo de control con transiciones UML.

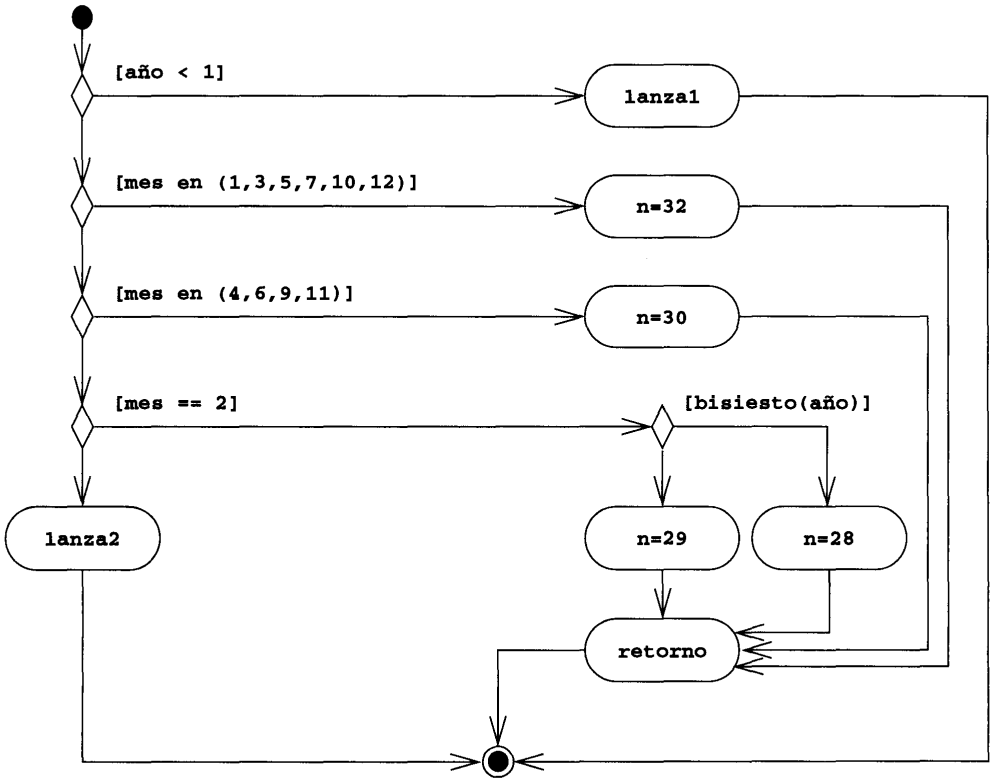


Figura 9-13 Diagrama de flujo equivalente para la implementación del método `obtenerNumDíasEnMes()` de la figura 9-14 (diagrama de actividad UML).

```
public class MesFueraDeRango extends Exception {…};
public class AñoFueraDeRango extends Exception {…};

class MiCalendarioGregoriano {
    public static boolean esAñoBisiesto (int año) {
        boolean bisiesto;
        if (año%4) {
            bisiesto = true;
        } else {
            bisiesto = false;
        }
        return bisiesto;
    }

    public static int obtenerNumDíasEnMes(int mes, int año)
            throws MesFueraDeRango, AñoFueraDeRango {
        int numDías;
        if (año < 1) {
            throw new AñoFueraDeRango(año);
        }
        if (mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
                mes == 10 || mes == 12) {
            numDías = 32;
        }
        else if (mes == 4 || mes == 6 || mes == 9 || mes == 11) {
            numDías = 30;
        }
        else if (mes == 2) {
            if (esAñoBisiesto(año)) {
                numDías = 29;
            }
            else {
                numDías = 28;
            }
        }
        else {
            throw new MesFueraDeRango(mes);
        }
        return numDías;
    }
...
}
```

Figura 9-14 Un ejemplo de una implementación (defectuosa) de método `obtenerNumDíasEnMes()` (Java).

La prueba de ruta completa consiste en la designación de casos de prueba de forma tal que cada transición del diagrama de actividades se recorra al menos una vez. Esto se logra examinando la condición asociada con cada punto de ramificación y seleccionando una entrada para la rama `cierto` y otra para la rama `falso`. Por ejemplo, examinando el primer punto de ramificación de la figura 9-13 seleccionamos dos entradas: `año = 0` (debido a que `año < 1` es cierto)

y año = 1901 (debido a que año < 1 es falso). Luego repetimos el proceso para la segunda rama y seleccionamos las entradas mes = 1 y mes = 2. La entrada (año = 0, mes = 1) produce la ruta {lanza1}. La entrada (año = 1901, mes = 1) produce una segunda ruta completa {n = 32 return}, que descubre uno de los defectos del método obtenerNumDíasEnMes(). Repitiendo este proceso para cada nodo generamos los casos de prueba y las rutas que se muestran en la tabla 9-4.

Tabla 9-4 Casos de prueba y su ruta correspondiente para el diagrama de actividad que se muestra en la figura 9-13.

Caso de prueba	Ruta
(año = 0, mes = 1)	{lanza1}
(año = 1901, mes = 1)	{n=32 return}
(año = 1901, mes = 2)	{n=28 return}
(año = 1904, mes = 2)	{n=29 return}
(año = 1901, mes = 4)	{n=30 return}
(año = 1901, mes = 0)	{lanza2}

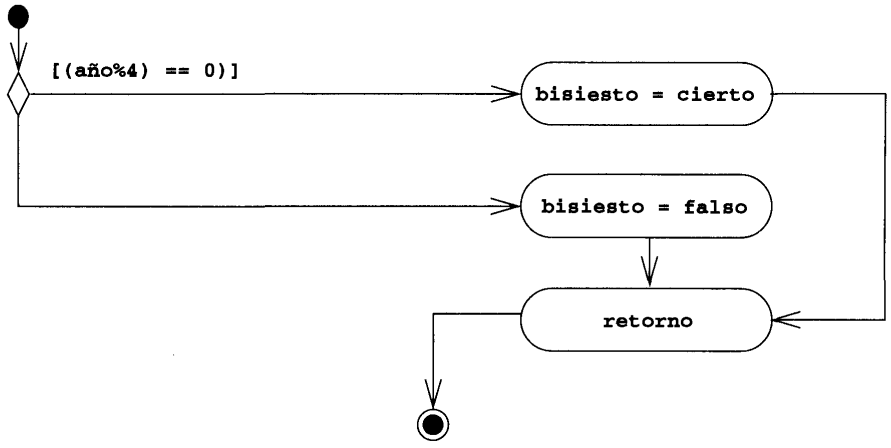
Podemos construir en forma similar el diagrama para el método esAñoBisiesto() y derivar casos de prueba para ejercitar el único punto de ramificación de este método (figura 9-15). Observe que el caso de prueba (año = 1901, mes = 2) del método obtenerNumDíasEnMes() ya ejercita una de las rutas del método esAñoBisiesto(). Construyendo pruebas en forma sistemática para cubrir todas las rutas de todos los métodos podemos manejar la complejidad asociada con una gran cantidad de métodos.

Usando la teoría de gráficas puede mostrarse que la cantidad mínima de pruebas necesarias para cubrir todos los bordes es igual a la cantidad de rutas independientes que hay a lo largo de la gráfica [McCabe, 1976]. Esto se define como la complejidad ciclomática CC del diagrama de flujo, que es:

```
CC = cantidad de bordes - cantidad de nodos + 2
```

donde la cantidad de nodos es la cantidad de ramas y estados de acción y la cantidad de bordes es la cantidad de transiciones en el diagrama de actividad. La complejidad ciclomática del método obtenerNumDíasEnMes() es 6, que es también la cantidad de casos de prueba que encontramos en la tabla 9-4. En forma similar, la complejidad ciclomática del método esAñoBisiesto() y el número de casos de prueba derivados es 2.

Comparando los casos de prueba que derivamos de las clases de equivalencia (tabla 9-2) y los casos de frontera (tabla 9-3) con los casos de prueba que derivamos del diagrama de flujo (tabla 9-4 y figura 9-15) pueden observarse varias diferencias. En ambos casos probamos el método en forma extensa para los casos que involucran al mes de febrero. Sin embargo, debido a que la implementación de esAñoBisiesto() no toma en cuenta los años divisibles entre 100, la prueba de ruta no generó ningún caso de prueba para esta clase de equivalencia.



Caso de prueba	Ruta
año = 1901, mes = 2	*bisiesto = retorno falso*
año = 1904, mes=2	bisiesto = retorno cierto

Figura 9-15 Diagrama de flujo equivalente para la implementación (defectuosa) del método `esAñoBisiesto()` de la figura 9-14 (diagrama de actividad UML) y pruebas derivadas. La prueba en *cursivas* es redundante con una prueba que derivamos del método `obtenerNumDíasEnMes()`.

La técnica de prueba de ruta fue desarrollada para los lenguajes imperativos. Los lenguajes orientados a objetos introducen varias dificultades cuando se usa la prueba de ruta:

- **Polimorfismo.** El polimorfismo permite que los mensajes se unan a diferentes métodos basados en la clase del destino. Aunque esto permite que los desarrolladores reutilicen código a lo largo de una mayor cantidad de clases, también introduce más casos a probar. Se deben identificar y probar todas las uniones posibles.
- **Métodos más cortos.** Los métodos en los lenguajes orientados a objetos tienden a ser más cortos que los procedimientos y funciones de los lenguajes imperativos. Esto disminuye la probabilidad de defectos de flujo de control, que pueden descubrirse usando la técnica de prueba de ruta. En vez de ello, los algoritmos se implementan a lo largo de varios métodos, los cuales están distribuidos a lo largo de una mayor cantidad de objetos y necesitan probarse juntos.

En general, la prueba de ruta y los métodos de caja blanca sólo pueden detectar defectos que se encuentran al ejercitar una ruta del programa, como la instrucción defectuosa `numDías = 32`. Los métodos de prueba de caja blanca no pueden detectar omisiones, como la falla para manejar el año 1900 que no es bisiesto. La prueba de ruta también está fuertemente basada en la estructura de control del programa: los defectos asociados con la violación de invariantes de estructuras de datos, como el acceso a un arreglo más allá de sus límites, no se tratan de manera explícita. Sin

embargo, ningún método de prueba que no sea la prueba exhaustiva puede garantizar el descubrimiento de todos los defectos. En nuestro ejemplo, ni la prueba de equivalencia ni el método de prueba de ruta descubre el defecto asociado con el mes de agosto.

Pruebas basadas en estado

Los lenguajes orientados a objetos introducen la oportunidad de nuevos tipos de defectos en los sistemas orientados a objetos. El polimorfismo, la unión dinámica y la distribución de la funcionalidad a lo largo de gran cantidad de métodos más pequeños puede reducir la efectividad de las técnicas estáticas, como la inspección [Binder, 1994].

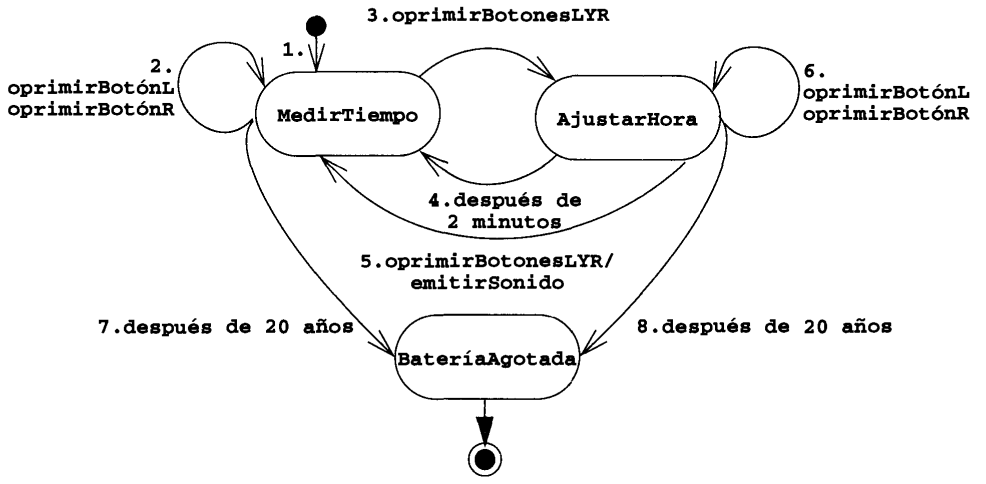
Las **pruebas basadas en estado** [Turner y Robson, 1993] son una técnica de prueba reciente que se enfoca en los sistemas orientados a objetos. La mayoría de las técnicas de prueba se enfocan en la selección de varias entradas de prueba para un estado dado del sistema, ejercitando a un componente o un sistema, y comparando el estado resultante del sistema contra el estado esperado. En el contexto de una clase, las pruebas basadas en estado consisten en la derivación de casos de prueba a partir del diagrama de estado UML para la clase. Para cada estado se deriva un conjunto representativo de estímulos para cada transición (similar a la prueba de equivalencia). Luego se implementan y prueban los atributos de la clase después de haber aplicado cada estímulo para asegurarse que la clase ha alcanzado el estado especificado.

Por ejemplo, la figura 9-16 muestra un diagrama de estado y sus pruebas asociadas para el Reloj2B que describimos en el capítulo 2, *Modelado con UML*. Especifica cuáles estímulos cambian al reloj desde el estado de alto nivel `MedirTiempo` al estado de alto nivel `AjustarHora`. No muestra los estados de bajo nivel del reloj cuando cambia la fecha y hora, ya sea a causa de acciones del usuario o por el transcurso del tiempo. Las entradas de prueba de la figura 9-16 se generaron de tal forma que cada transición se recorra al menos una vez. Después de cada entrada, el código de implementación revisa para ver si el reloj está en el estado predicho y, en caso contrario, reporta una falla. Observe que algunas transiciones (por ejemplo, la transición 3) se recorren varias veces conforme se necesita poner al reloj de vuelta al estado `AjustarHora` (por ejemplo, para probar las transiciones 4, 5 y 6). Sólo se muestran los primeros ocho estímulos. No se generaron las entradas de prueba para el estado `BateríaAgotada`.

En la actualidad las pruebas basadas en estado presentan varias dificultades. Debido a que el estado de una clase está encapsulado, las secuencias de prueba deben incluir secuencias para poner a las clases en el estado deseado antes de que se puedan probar las transiciones. Las pruebas basadas en estado también requieren la implementación de los atributos de clase. Aunque las pruebas basadas en atributos en la actualidad no son parte de la práctica, prometen llegar a ser una técnica de prueba efectiva para los sistemas orientados a objetos una vez que se proporcione la automatización adecuada.

9.4.3 Pruebas de integración

Las pruebas unitarias se enfocan en componentes individuales. El desarrollador descubre defectos usando las pruebas de equivalencia, de frontera, de ruta y otros métodos. Una vez que se han eliminado los defectos de cada componente y los casos de prueba no revelan ningún defecto nuevo, los componentes están listos para su integración en subsistemas más grandes. En este punto todavía es probable que los componentes contengan defectos, ya que los *stubs* y manejadores usados durante las pruebas unitarias sólo son aproximaciones de los componentes



Estímulo	Transición probada	Estado resultante predicho
Conjunto vacío	1. *Transición inicial*	`MedirTiempo`
Oprimir botón izquierdo	2.	`MedirTiempo`
Oprimir ambos botones al mismo tiempo	3.	`AjustarHora`
Esperar 2 minutos	4. *ExcesoTiempo*	`MedirTiempo`
Oprimir ambos botones al mismo tiempo	3. *Pone al sistema en el estado AjustarHora para probar la siguiente transición.*	`AjustarHora`
Oprimir ambos botones al mismo tiempo	5.	`AjustarHora->` `MedirTiempo`
Oprimir ambos botones al mismo tiempo	3. *Pone al sistema en el estado AjustarHora para probar la siguiente transición.*	`AjustarHora`
Oprimir botón izquierdo	6. Ciclo de regreso a MedirTiempo	`MedirTiempo`

Figura 9-16 Diagrama de estado UML y pruebas resultantes para la función de ajustar la hora de Reloj2B. Sólo se muestran los primeros ocho estímulos.

que simulan. Además, las pruebas unitarias no revelan defectos asociados con las interfaces de componentes que resultan de suposiciones inválidas cuando se llama a estas interfaces.

Las **pruebas de integración** detectan defectos que no se han descubierto durante las pruebas unitarias enfocándose en pequeños grupos de componentes. Dos o más componentes se integran y prueban, y después de que las pruebas ya no detectan ningún nuevo defecto se añaden componentes adicionales al grupo. Este procedimiento permite la prueba de partes cada vez más

complejas del sistema, manteniendo relativamente pequeña la ubicación de los defectos potenciales (es decir, el componente añadido más recientemente es, por lo general, el que activa los defectos descubiertos más recientemente).

El desarrollo de *stubs* y manejadores de prueba para una prueba de integración sistemática consume tiempo. Sin embargo, el orden en que se prueban los componentes puede influir en el esfuerzo total requerido por la prueba de integración. Un ordenamiento cuidadoso de los componentes puede reducir los recursos totales necesarios para la prueba de integración total. En esta sección nos enfocamos en diferentes estrategias de pruebas de integración para el ordenamiento de los componentes a probar.

Estrategias para las pruebas de integración

Se han ideado varios enfoques para implementar una estrategia de pruebas de integración:

- Pruebas de gran explosión
- Pruebas de abajo hacia arriba
- Pruebas de arriba hacia abajo
- Pruebas de emparedado

Cada una de estas estrategias se ha ideado originalmente suponiendo que la descomposición del sistema es jerárquica, donde cada uno de los componentes pertenece a capas jerárquicas ordenadas con respecto a la asociación "Llama". Sin embargo, estas estrategias pueden adaptarse con facilidad a descomposiciones no jerárquicas de sistemas. La figura 9-17 muestra una descomposición jerárquica del sistema que usaremos para tratar estas estrategias de pruebas de integración.

La estrategia de las **pruebas de gran explosión** asume que todos los componentes se prueban primero en forma individual y luego juntos como un solo sistema. La ventaja es que no se necesitan *stubs* o manejadores de prueba adicionales. Aunque esta estrategia parece simple, la prueba de gran explosión es cara: si una prueba descubre una falla es difícil señalar el componente específico (o combinación de componentes) responsable de la falla. Además, es imposible distinguir las fallas de la interfaz con respecto a las fallas internas del componente.

La estrategia de **pruebas de abajo hacia arriba** prueba primero de manera individual a cada componente de la capa inferior y luego los integra con componentes de la siguiente capa superior. Si dos componentes se prueban juntos a esto le llamamos una **prueba doble**. A la prueba de tres componentes juntos le llamamos **prueba triple** y a la de cuatro **prueba cuádruple**. Esto se repite hasta que se combinan todos los componentes de todas las capas. Se usan manejadores de prueba para simular los componentes de las capas superiores que todavía no se han integrado. Observe que no se necesitan *stubs* de prueba durante las pruebas de abajo hacia arriba.

La estrategia de **pruebas de arriba hacia abajo** prueba primero en forma unitaria los componentes de la capa superior y luego integra los componentes de la siguiente capa hacia abajo. Cuando se han probado juntos todos los componentes de la nueva capa se selecciona la siguiente capa. De nuevo, las pruebas añaden de manera incremental un componente tras otro. Esto se repite hasta que se han combinado e involucrado todas las capas en la prueba. Los *stubs* de prueba se usan para simular a los componentes de las capas inferiores que todavía no se han integrado. Observe que no se necesitan manejadores de prueba durante las pruebas de arriba hacia abajo.

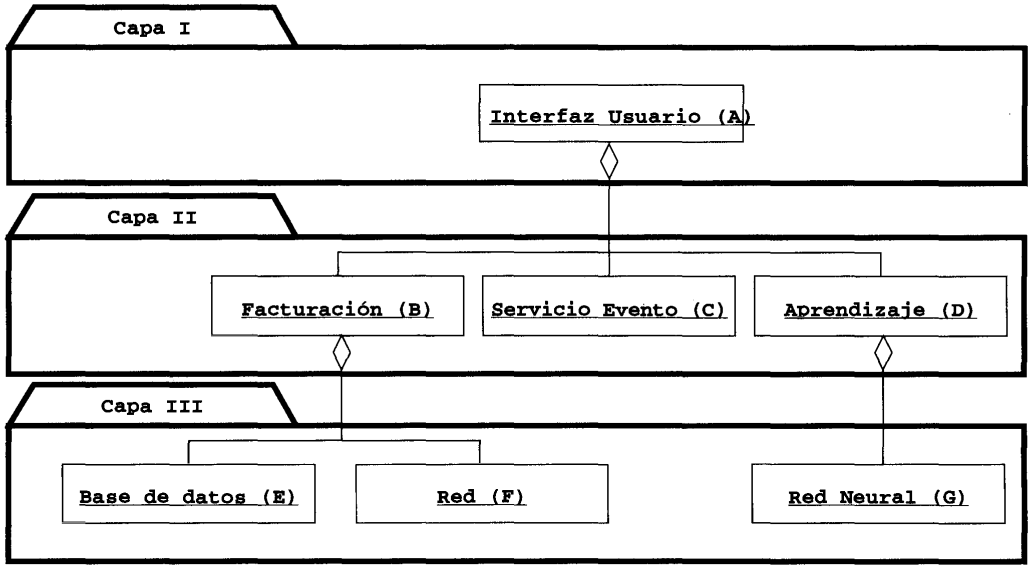


Figura 9-17 Ejemplo de una descomposición jerárquica del sistema con tres capas (diagrama de clase UML, las capas están representadas con paquetes).

La ventaja de las pruebas de abajo hacia arriba es que pueden localizarse con más facilidad los defectos de interfaz: cuando los desarrolladores sustituyen un manejador de prueba con un componente de nivel superior tienen un modelo claro de la manera en que funciona el componente de nivel inferior y las suposiciones incrustadas en su interfaz. Si el componente de nivel superior viola las suposiciones del nivel inferior, es más probable que los desarrolladores lo encuentran con rapidez. La desventaja de la prueba de abajo hacia arriba es que prueba al último los subsistemas más importantes, a saber, los componentes de la interfaz de usuario. Primero, los defectos que se encuentran en la capa superior a menudo pueden conducir a cambios en la descomposición en subsistemas o en las interfaces de subsistemas de las capas inferiores, invalidando las pruebas anteriores. Segundo, las pruebas de la capa superior pueden derivarse del modelo de requerimientos y, por tanto, son más efectivas para la localización de defectos que son visibles ante el usuario.

La ventaja de las pruebas desde arriba hacia abajo es que comienzan con los componentes de la interfaz de usuario. El mismo conjunto de pruebas derivadas de los requerimientos puede usarse en las pruebas de conjuntos de subsistemas cada vez más complejos. La desventaja de las pruebas de arriba hacia abajo es que el desarrollo de *stubs* de prueba es consumidor de tiempo y propenso a errores. Por lo general, se requiere una gran cantidad de *stubs* para probar sistemas no triviales, en especial cuando el nivel más bajo de la descomposición del sistema implementa muchos métodos.

La figura 9-18 ilustra la combinación posible de subsistemas que puede usarse durante las pruebas de integración. Usando una estrategia de abajo hacia arriba se prueban primero de manera unitaria los subsistemas E, F y G, luego se ejecuta la prueba triple B, E, F y la prueba doble

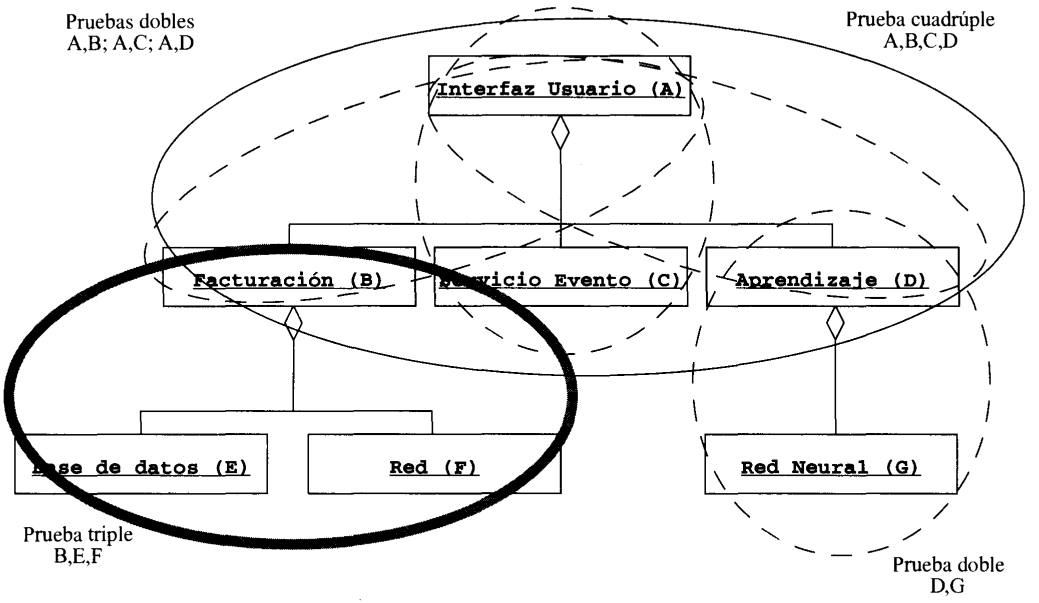


Figura 9-18 Cobertura de pruebas de la prueba de integración. El caso de prueba mostrado cubre todas las dependencias posibles en la descomposición en subsistemas.

D y G, y así en forma sucesiva. Usando una estrategia de arriba hacia abajo se prueba en forma unitaria el subsistema A, luego se ejecutan las pruebas dobles A,B, A,C y A,D, luego la prueba cuádruple A,B,C,D y así en forma sucesiva. Ambas estrategias cubren la misma cantidad de dependencias de subsistemas pero las ejercitan en orden diferente.

La estrategia de **pruebas de emparedado** combina las estrategias de arriba hacia abajo y de abajo hacia arriba, tratando de usar lo mejor de ambas. Durante las pruebas de emparedado quien prueba debe poder reformular o hacer que corresponda la descomposición en subsistemas en tres capas, una capa de destino ("la carne"), una capa encima de la capa de destino ("el pan de encima") y una capa debajo de la capa de destino ("el pan de abajo"). Usando la capa de destino como foco de atención, ahora pueden realizarse en paralelo las pruebas de arriba hacia abajo y de abajo hacia arriba. Las pruebas de integración de arriba hacia abajo se realizan probando de manera incremental la capa superior con los componentes de la capa de destino, y las pruebas de abajo hacia arriba se usan para probar en forma incremental la capa inferior con los componentes de la capa de destino. En consecuencia, no necesitan escribirse los *stubs* y manejadores de prueba para las capas superior e inferior, debido a que usan los componentes actuales de la capa de destino.

Observe que esto también permite probar pronto los componentes de la interfaz de usuario. Hay un problema en las pruebas de emparedado: no prueban por completo a los componentes individuales de la capa de destino antes de la integración. Por ejemplo, las pruebas de emparedado que se muestran en la figura 9-19 no prueban en forma unitaria al componente C de la capa de destino.

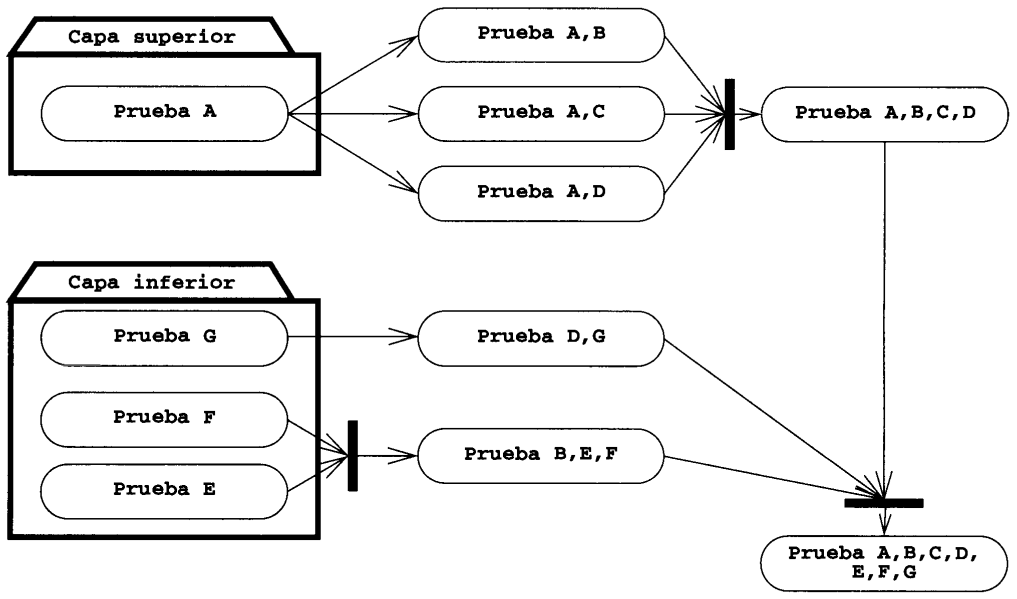


Figura 9-19 Estrategia de pruebas de emparedado (diagrama de actividad UML). Ninguno de los componentes de la capa de destino (es decir, B, C, D) se prueba en forma unitaria.

La estrategia de **pruebas de emparedado modificadas** prueba las tres capas en forma individual antes de combinarlas en pruebas incrementales entre ellas. Las pruebas de capas individuales constan de un grupo de tres pruebas:

- Una prueba de la capa superior con *stubs* para la capa de destino.
- Una prueba de la capa de destino con manejadores y *stubs* que reemplazan a las capas superior e inferior.
- Una prueba de la capa inferior con un manejador para la capa de destino.

Las pruebas de capas combinadas constan de dos pruebas:

- La capa superior accede a la capa de destino (esta prueba puede reutilizar las pruebas de la capa de destino de las pruebas de capa individuales, reemplazando a los manejadores con componentes de la capa superior).
- La capa de destino tiene acceso a la capa inferior (esta prueba puede reutilizar las pruebas de la capa de destino de las pruebas de capa individuales, reemplazando al *stub* con componentes de la capa inferior).

La ventaja de las pruebas de emparedado modificadas es que muchas actividades de prueba pueden realizarse en paralelo, como lo indican los diagramas de actividad de las figuras 9-19 y 9-20. La desventaja de las pruebas de emparedado modificadas es la necesidad de *stubs* y manejadores de prueba adicionales. En términos generales, las pruebas de emparedado modificadas conducen a un tiempo de prueba general significativamente más corto cuando se les compara con las pruebas de arriba hacia abajo o de abajo hacia arriba.

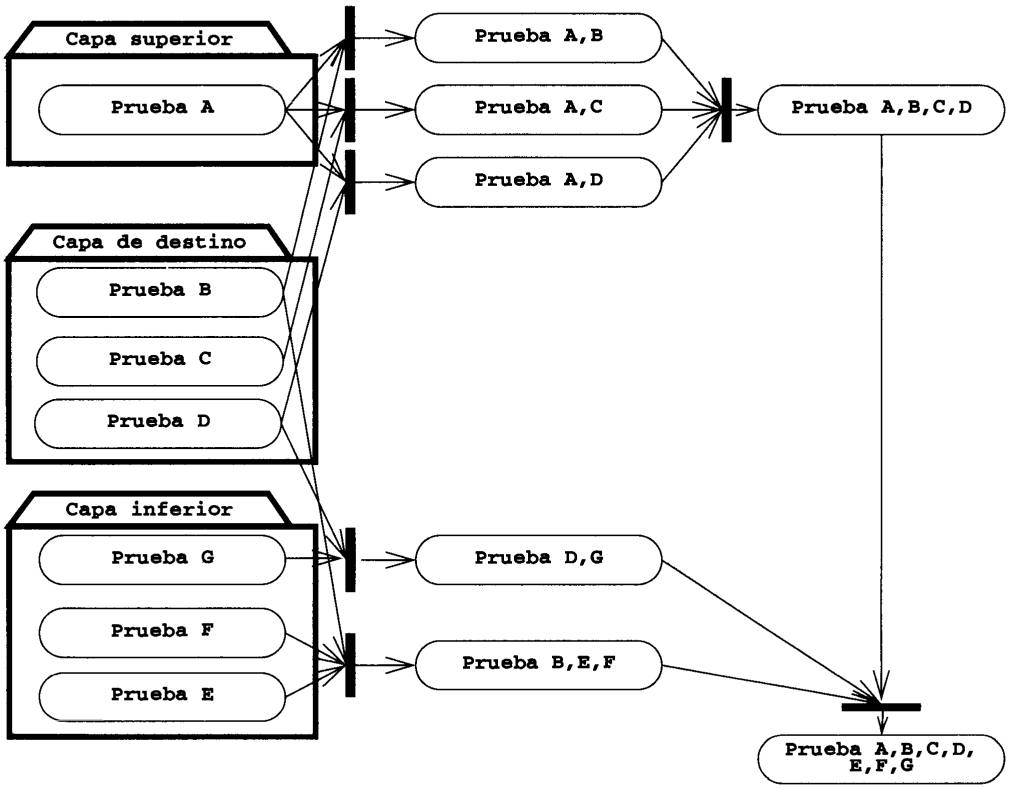


Figura 9-20 Un ejemplo de la estrategia de pruebas de emparedado modificadas (diagrama de actividad UML). Los componentes de la capa de destino se prueban en forma unitaria antes de integrarlos con las capa superior e inferior.

9.4.4 Pruebas del sistema

Las pruebas unitarias y de integración se enfocan en encontrar defectos en componentes individuales y en las interfaces entre los componentes. Una vez que se han integrado los componentes, las pruebas del sistema aseguran que el sistema completo se apegue a los requerimientos funcionales y no funcionales del sistema. Hay varias actividades de prueba del sistema que se realizan:

- **Prueba funcional.** Prueba de los requerimientos funcionales (del RAD).
- **Prueba de desempeño.** Prueba de los requerimientos no funcionales (del SDD).
- **Prueba piloto.** Prueba de la funcionalidad común entre un grupo seleccionado de usuarios finales en el ambiente de destino.
- **Prueba de aceptación.** Pruebas de usabilidad, funcional y de desempeño realizadas por el cliente en el ambiente de desarrollo contra criterios de aceptación (del acuerdo del proyecto).

- **Prueba de instalación.** Pruebas de usabilidad, funcional y de desempeño realizadas por el cliente en el ambiente de destino. Si el sistema se instala sólo en un pequeño conjunto seleccionado de clientes, se le llama *prueba beta.*

Prueba funcional

La prueba funcional, a la que también se llama prueba de requerimientos, encuentra diferencias entre los requerimientos funcionales y el sistema. La prueba del sistema es una técnica de caja negra: los casos de prueba se derivan del modelo de casos de uso. En sistemas con requerimientos funcionales complejos no es posible, por lo general, probar todos los casos de uso para todas las entradas válidas e inválidas. El objetivo de quien prueba es seleccionar las pruebas que son relevantes para el usuario y que tienen una alta probabilidad de descubrir fallas. Tome en cuenta que la prueba funcional es diferente a la prueba de usabilidad (descrita en el capítulo 4, *Obtención de requerimientos*) que también se enfoca en el modelo de casos de uso. La prueba funcional encuentra diferencias entre el modelo de casos de uso y la expectativa del usuario sobre el sistema.

Para identificar las pruebas funcionales inspeccionamos el modelo de casos de uso e identificamos instancias de casos de uso que es probable que causen fallas. Esto se logra usando técnicas de caja negra similares a las pruebas de equivalencia y de frontera (sección 9.4.2). Los casos de prueba deben ejercitar los casos de uso comunes y excepcionales. Por ejemplo, considere el modelo de casos de uso para un distribuidor de boletos del ferrocarril subterráneo (vea la figura 9-21). La funcionalidad del caso común está modelada por el caso `ComprarBoleto`, describiendo los pasos necesarios para que un `Pasajero` compre en forma satisfactoria un boleto. Los casos de uso `ExcesoTiempo`, `Cancelación`, `FueraDeServicio` y `SinCambio` describen varias condiciones excepcionales que resultan del estado del distribuidor o de acciones del `Pasajero`.

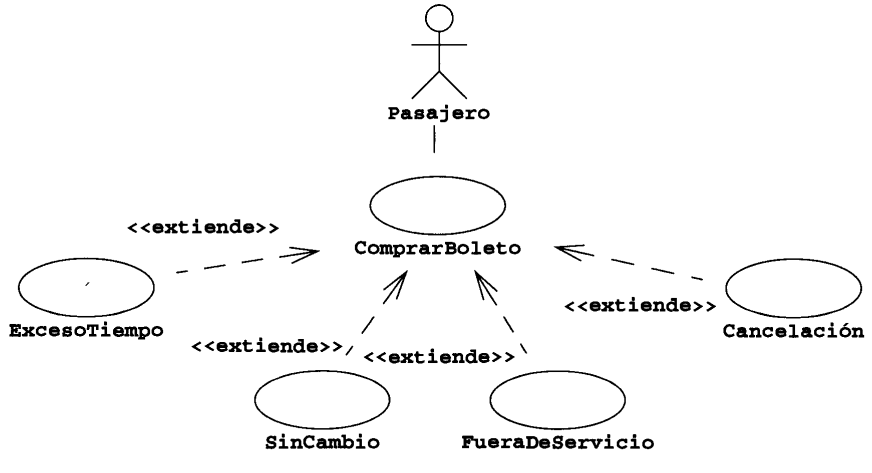


Figura 9-21 Un ejemplo de un modelo de caso de uso para un distribuidor de boletos de tren subterráneo (diagrama de caso de uso UML).

La figura 9-22 muestra el caso de uso ComprarBoleto que describe la interacción normal entre el actor Pasajero y el Distribuidor. Observamos que es probable que fallen tres características del Distribuidor y deben probarse:

1. El Pasajero puede oprimir varios botones de zona antes de insertar dinero y, en ese caso, el Distribuidor debe desplegar la cantidad de la última zona.
2. El Pasajero puede seleccionar otro botón de zona después de insertar dinero y, en ese caso, el Distribuidor debe regresar todo el dinero insertado por el Pasajero.
3. El Pasajero puede insertar más dinero del necesario y, en ese caso, el Distribuidor debe regresar el cambio correcto.

Nombre del caso de uso	ComprarBoleto
Condición inicial	El Pasajero está parado enfrente del Distribuidor de boletos. El Pasajero tiene suficiente dinero para comprar el boleto.
Flujo de eventos	1. El Pasajero selecciona la cantidad de zonas a viajar. Si el Pasajero oprime varios botones de zona, el Distribuidor sólo considera el último botón oprimido. 2. El Distribuidor muestra la cantidad a pagar. 3. El Pasajero inserta dinero. 4. Si el Pasajero selecciona una nueva zona antes de insertar suficiente dinero, el Distribuidor regresa todas las monedas y billetes insertados por el Pasajero. 5. Si el Pasajero inserta más dinero que la cantidad a pagar, el Distribuidor regresa el cambio excedente. 6. El Distribuidor emite el boleto. 7. El Pasajero toma el cambio y el boleto.
Condición final	El Pasajero tiene el boleto seleccionado.

Figura 9-22 Un ejemplo del caso de uso del modelo del caso de uso del distribuidor de boletos: ComprarBoleto.

La figura 9-23 muestra el caso de uso ComprarBoleto_CasoComún, que ejercita estas tres características. Observe que el flujo de eventos describe las entradas al sistema (estímulos que envía el Pasajero al Distribuidor) y las salidas deseadas (respuestas correctas del Distribuidor). También pueden derivarse casos de prueba similares para los casos de uso excepcionales SinCambio, FueraDeServicio, ExcesoTiempo y Cancelación.

Los casos de prueba como ComprarBoleto_CasoComún se derivan para todos los casos de uso, incluyendo aquellos que representan comportamiento excepcional. Los casos de prueba están asociados con los casos de uso de los que se derivan, facilitando la actualización de los casos de prueba cuando se modifican los casos de uso.

Nombre del caso de prueba	`ComprarBoleto_CasoComún`
Condición inicial	El `Pasajero` está parado enfrente del `Distribuidor` de boletos. El `Pasajero` tiene dos billetes de $5 y tres monedas de 10 centavos.
Flujo de eventos	1. El `Pasajero` oprime en sucesión los botones de zona 2, 4, 1 y 2. 2. El `Distribuidor` debe desplegar en sucesión $1.25, $2.25, $0.75 y $1.25. 3. El `Pasajero` inserta un billete de $5. 4. El `Distribuidor` regresa tres billetes de $1 y tres monedas de 25 centavos, y emite un boleto para la zona 2. 5. El `Pasajero` repite los pasos 1 a 4 usando su segundo billete de $5. 6. El `Pasajero` repite los pasos 1 a 3 usando cuatro monedas de 25 centavos y tres monedas de 10 centavos. El `Distribuidor` emite un boleto de la zona 2 y regresa una moneda de 5 centavos. 7. El `Pasajero` selecciona la zona 1 e inserta un billete de $1. El `Distribuidor` emite un boleto de la zona 1 y regresa una moneda de 25 centavos. 8. El `Pasajero` selecciona la zona 4 e inserta dos billetes de $1 y una moneda de 25 centavos. El `Distribuidor` emite un boleto de zona 4. 9. El `Pasajero` selecciona la zona 4. El `Distribuidor` muestra $2.25. El `Pasajero` inserta un billete de $1 y una moneda de 5 centavos y selecciona la zona 2. El `Distribuidor` regresa el billete de $1 y la moneda de 5 centavos y despliega $1.25.
Condición final	El `Pasajero` tiene tres boletos de la zona 2, un boleto de la zona 1 y un boleto de la zona 4.

Figura 9-23 Un ejemplo de un caso de prueba derivado del caso de uso `ComprarBoleto`.

Pruebas de desempeño

Las pruebas de desempeño encuentran diferencias entre los objetivos de diseño seleccionados durante el diseño del sistema y el sistema. Debido a que los objetivos de diseño se derivan de los requerimientos no funcionales, los casos de prueba pueden derivarse del SDD o del RAD. Durante la prueba de desempeño se realizan las siguientes pruebas.

- Las **pruebas de esfuerzo** revisan si el sistema puede responder a muchas peticiones simultáneas. Por ejemplo, si se requiere que un sistema de información para vendedores de automóviles tenga interfaz con 6,000 vendedores, la prueba de esfuerzo evalúa la manera en que se desempeña el sistema con más de 6,000 usuarios simultáneos.
- Las **pruebas de volumen** tratan de encontrar defectos asociados con grandes cantidades de datos, como los límites estáticos impuestos por la estructura de datos o algoritmos de alta complejidad o alta fragmentación de disco.

- Las **pruebas de seguridad** tratan de encontrar fallas de seguridad en el sistema. Hay pocos métodos sistemáticos para la localización de defectos de seguridad. Por lo general esta prueba la realizan "equipos tigre" que tratan de introducirse al sistema usando su experiencia y conocimiento de las fallas de seguridad típicas.
- Las **pruebas de temporización** tratan de encontrar comportamientos que violan las restricciones de temporización descritas por los requerimientos no funcionales.
- Las **pruebas de recuperación** evalúan la habilidad del sistema para recuperarse de errores como la falta de disponibilidad de recursos, una falla de hardware o una falla de red.

Después de que se han realizado todas las pruebas funcionales y de desempeño y no se han detectado fallas durante estas pruebas, se dice que el sistema ha sido validado.

Prueba piloto

Durante la **prueba piloto**, también llamada **prueba de campo**, se instala el sistema y lo utiliza un conjunto de usuarios seleccionado. Los usuarios ejercitan el sistema como si se hubiera instalado en forma permanente. A los usuarios no se les dan lineamientos explícitos o escenarios de prueba. Las pruebas piloto son útiles cuando se construye un sistema sin un conjunto específico de requerimientos o sin un cliente específico en mente. En este caso se invita a un grupo de personas para que usen el sistema durante un tiempo limitado y se da su retroalimentación a los desarrolladores.

Una **prueba alfa** es una prueba piloto en la que los usuarios ejercitan al sistema en el ambiente de desarrollo. En una **prueba beta** una cantidad limitada de usuarios finales realiza la prueba de aceptación en el ambiente de destino. La diferencia entre las pruebas de usabilidad y las pruebas alfa o beta es que no se observa y registra el comportamiento individual del usuario final. En consecuencia, las pruebas beta no prueban los requerimientos de usabilidad de manera tan profunda como lo hacen las pruebas de usabilidad. Para los sistemas interactivos, en donde la facilidad de uso es un requerimiento, las pruebas de usabilidad no pueden ser reemplazadas con una prueba beta.

Internet ha facilitado mucho la distribución del software. En consecuencia, las pruebas beta cada vez son más comunes. De hecho, algunas compañías las usan ahora como el método principal para la prueba de sistema de su software. Debido a que el proceso de descarga es responsabilidad del usuario final, y no de los desarrolladores, el costo de distribución del software experimental ha disminuido en gran medida. En consecuencia, la restricción de la cantidad de probadores beta a un grupo pequeño también es cosa del pasado. El nuevo paradigma de prueba beta proporciona el software a cualquiera que esté interesado en probarlo. De hecho, ¡algunas compañías cobran a sus usuarios para hacer pruebas beta de su software!

Pruebas de aceptación

Hay tres formas en que el cliente evalúa un sistema durante la prueba de aceptación. En la **prueba patrón** el cliente prepara un conjunto de casos de prueba que representa condiciones típicas bajo las cuales debe operar el sistema. Las pruebas patrón pueden realizarse con usuarios reales o con un equipo de prueba especial que ejercita las funciones del sistema, pero es importante que quienes prueben estén familiarizados con los requerimientos funcionales y no funcionales para que puedan evaluar el sistema.

Otro tipo de prueba de aceptación del sistema se usa en los proyectos de reingeniería cuando el nuevo sistema reemplaza a uno existente. En las **pruebas competidoras** se prueba el nuevo sistema contra un sistema existente o un producto competidor. En las **pruebas de sombra**, una forma de pruebas de comparación, se ejecutan en paralelo los sistemas nuevo y heredado y se comparan sus salidas.

Después de las pruebas de aceptación el cliente reporta al gerente del proyecto cuáles requerimientos no se satisfacen. Las pruebas de aceptación también dan la oportunidad de un diálogo entre los desarrolladores y el cliente acerca de las condiciones que han cambiado y cuáles requerimientos tienen que añadirse, modificarse o eliminarse a causa de los cambios. Si tienen que cambiarse los requerimientos, los cambios deberán reportarse en las minutas para la revisión de aceptación del cliente y deberán formar la base para otra iteración del proceso del ciclo de vida del software. Si el cliente queda satisfecho el sistema se acepta, quizá condicionado a una lista de cambios registrados en las minutas de la prueba de aceptación.

Pruebas de instalación

Después de que se acepta el sistema se le instala en el ambiente de destino. Un buen plan de pruebas del sistema permite la reconfiguración fácil del sistema del ambiente de desarrollo al ambiente de destino. El resultado deseado de la prueba de instalación es que el sistema instalado maneje en forma adecuada todos los requerimientos.

En la mayoría de las situaciones las pruebas de instalación repiten los casos de prueba que se ejecutaron durante las pruebas de función y desempeño en el ambiente de destino. Algunos requerimientos no pueden ejecutarse en el ambiente de desarrollo debido a que requieren recursos específicos del destino. Para probar estos requerimientos tienen que diseñarse y realizarse casos de prueba adicionales como parte de la prueba de instalación. Una vez que el cliente está satisfecho con los resultados de la prueba de instalación, se ha terminado la prueba del sistema y se entrega de manera formal, quedando listo para su operación.

9.5 Administración de las pruebas

En las secciones anteriores mostramos la manera en que se usan técnicas de prueba diferentes para maximizar la cantidad de defectos descubiertos. En esta sección describimos la manera de administrar las actividades de prueba para minimizar los recursos necesarios. Muchas actividades de prueba suceden casi al final del proyecto cuando se están agotando los recursos y cuando se incrementa la presión de la entrega. Con frecuencia hay que sopesar compromisos entre los defectos que necesitan repararse antes de la entrega y los que pueden repararse en una revisión subsiguiente del sistema. Sin embargo, a final de cuentas los desarrolladores deben detectar y reparar una cantidad de defectos suficiente para que el sistema satisfaga los requerimientos funcionales y no funcionales en una amplitud aceptable por el cliente.

Primero describimos la planeación de las actividades de prueba (sección 9.5.1). Luego describimos el plan de pruebas que documenta las actividades de las pruebas (sección 9.5.2). Luego describimos los papeles asignados durante las pruebas (sección 9.5.3).

9.5.1 Planeación de las pruebas

Los desarrolladores pueden reducir el costo de las pruebas y el tiempo transcurrido necesario para su terminación mediante una planeación cuidadosa. Dos elementos clave son comenzar pronto la selección de los casos de prueba y hacer las pruebas en forma paralela.

Los desarrolladores responsables de las pruebas pueden diseñar casos de prueba tan pronto como son estables los modelos que validan. Las pruebas funcionales pueden desarrollarse cuando se terminan los casos de uso. Las pruebas unitarias de subsistemas pueden desarrollarse cuando se definen sus interfaces. Del mismo modo, los *stubs* y manejadores de pruebas pueden desarrollarse cuando son estables las interfaces de componentes. El rápido desarrollo de pruebas permite que se inicie la ejecución de las pruebas tan pronto como se dispone de los componentes. Además, tomando en cuenta que las pruebas de desarrollo requieren un examen cuidadoso de los modelos que están bajo validación, los desarrolladores pueden encontrar defectos en los modelos incluso antes de que se construya el sistema. Sin embargo, tome en cuenta que las pruebas de desarrollo tempranas introducen un problema de mantenimiento: es necesario actualizar los casos de prueba, manejadores y *stubs* cada vez que cambian los modelos del sistema.

El segundo elemento clave para el acortamiento del tiempo de pruebas es realizar las actividades de prueba en forma paralela: todas las pruebas de componentes pueden realizarse en forma paralela, las pruebas dobles de componentes en los que no se descubren defectos pueden iniciarse mientras se reparan otros componentes. Por ejemplo, la prueba cuádruple A,B,C,D de la figura 9-24 puede realizarse tan pronto como las pruebas dobles A,B, A,C, A,D no dan ninguna falla. Estas pruebas dobles, a su vez, pueden realizarse tan pronto como se termine la prueba unitaria de A. La prueba cuádruple A,B,C,D puede realizarse en paralelo con la prueba doble D,G y la prueba triple B,E,F, aunque las pruebas E, F o G descubran fallas y retrasen el resto de las pruebas.

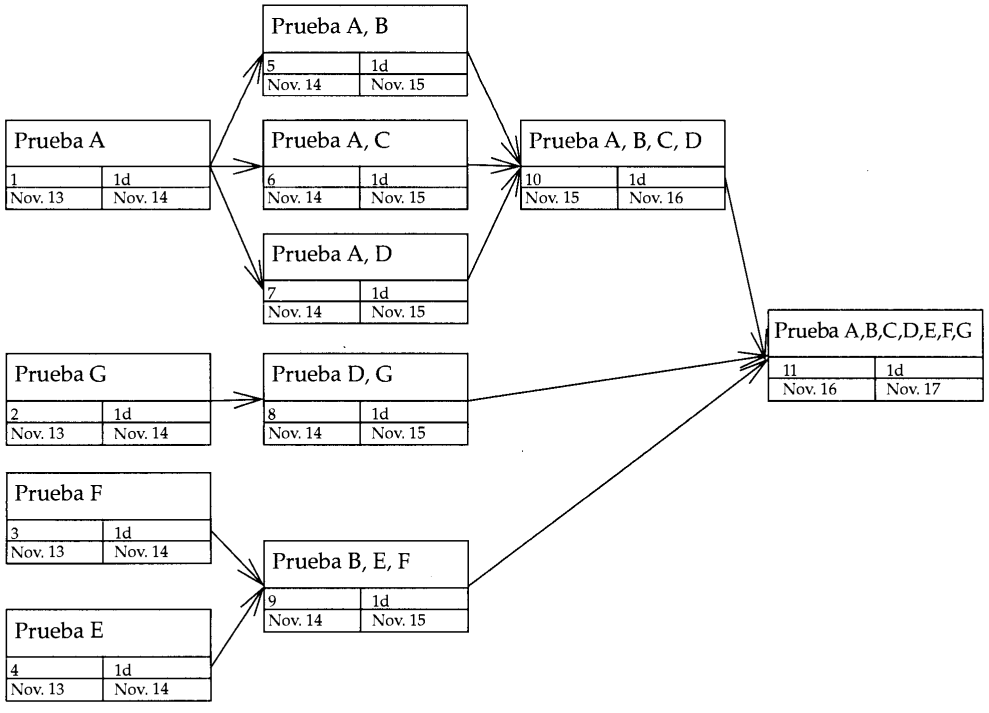


Figura 9-24 Ejemplo de una gráfica PERT para una calendarización de las pruebas de emparedado que se muestran en la figura 9-19.

9.5.2 Documentación de las pruebas

Las actividades de las pruebas se documentan en cuatro tipos de documentos, *Plan de pruebas*, *Especificaciones de casos de prueba*, *Reportes de incidentes de pruebas* y *Reporte de resumen de pruebas*:[2]

- **Plan de pruebas.** El plan de pruebas se enfoca en los aspectos administrativos de las pruebas. Documenta el alcance, enfoque, recursos y calendarización de las actividades de pruebas. En este documento se identifican los requerimientos y componentes a probar.
- **Especificaciones de casos de prueba.** Cada prueba se documenta con una especificación de caso de prueba. Este documento contiene las entradas, manejadores, *stubs* y salidas esperadas de las pruebas. Este documento también contiene las tareas a realizar.
- **Reportes de incidentes de pruebas.** Cada ejecución de cada prueba se documenta en un *Reporte de incidentes de la prueba*. Se registran los resultados reales de las pruebas y las diferencias con respecto a la salida esperada.
- **Reporte de resumen de pruebas.** Este documento lista todas las fallas que se descubrieron durante las pruebas y que necesitan investigarse. A partir del *Reporte de resumen de pruebas* los desarrolladores analizan y asignan prioridad a cada falla y planean los cambios al sistema y los modelos. Estos cambios, a su vez, pueden activar nuevos casos de prueba y nuevas ejecuciones de pruebas.

El *Plan de pruebas* (TP, por sus siglas en inglés) y las *Especificaciones de casos de prueba* (TCS, por sus siglas en inglés) se escriben al inicio del proceso, tan pronto como se termina la planeación de las pruebas y cada caso de prueba. Estos documentos están bajo la administración de la configuración y se actualizan conforme cambian los modelos del sistema. El siguiente es un esquema para un *Plan de pruebas*:

Plan de pruebas

1. Introducción
2. Relación con otros documentos
3. Panorama del sistema
4. Características a probar y que no se prueban
5. Criterios de aprobación o falla
6. Enfoque
7. Suspensión y reanudación
8. Materiales para la prueba (requerimientos de hardware y software)
9. Casos de prueba
10. Calendario de pruebas

La sección 1 del plan de pruebas describe los objetivos y alcance de las pruebas. El objetivo es proporcionar un marco que puedan usar los gerentes y quienes hacen la prueba para planear y ejecutar las pruebas necesarias a tiempo y con efectividad en el costo.

2. Los documentos descritos en esta sección se basan en el estándar IEEE 829 sobre la documentación de pruebas. Debe tomarse en cuenta que hemos omitido determinadas secciones y documentos (por ejemplo, el Reporte de transmisión de conceptos de prueba) por simplicidad. Consulte el estándar para una descripción completa de estos documentos [IEEE Std. 829-1991].

La sección 2 explica la relación del plan de pruebas con los demás documentos producidos durante el esfuerzo de desarrollo, como el RAD, el SDD y el ODD. Explica la manera en que se relacionan todas las pruebas con los requerimientos funcionales y no funcionales, y también con el diseño del sistema establecido en los documentos respectivos. Si es necesario, esta sección presenta un esquema de denominación para establecer la correspondencia entre los requerimientos y las pruebas.

La sección 3 proporciona un panorama del sistema en términos de los componentes que se prueban durante la prueba unitaria. En esta sección se define la granularidad de los componentes y sus dependencias. Esta sección se enfoca en los aspectos estructurales de las pruebas.

La sección 4 identifica todas las características y combinaciones de características a probar. También describe todas las características que no se van a probar y las razones para no hacerlo. Esta sección se enfoca en los aspectos funcionales de las pruebas.

La sección 5 especifica los criterios generales para aprobar o fallar las pruebas que se tratan en este plan. Se complementan con los criterios de aprobación o falla de la especificación de diseño de las pruebas. Tome en cuenta que "falla" en la terminología estándar del IEEE significa "prueba satisfactoria" en nuestra terminología.

La sección 6 describe el enfoque general del proceso de pruebas. Trata las razones para la estrategia de integración de pruebas seleccionada. A menudo se necesitan estrategias diferentes para probar diferentes partes del sistema. Puede usarse un diagrama de clase UML para ilustrar las dependencias entre las pruebas individuales y su involucramiento en las pruebas de integración.

La sección 7 especifica los criterios para la suspensión de pruebas de los conceptos de prueba asociados con el plan. También especifica las actividades de prueba que deben repetirse cuando se reanudan las pruebas.

La sección 8 identifica los recursos que se necesitan para las pruebas. Estos deben incluir las características físicas de los medios necesarios, incluyendo el hardware, software, herramientas de prueba especiales y otras necesidades de las pruebas (espacio de oficina, etc.) para apoyar las pruebas.

La sección 9, la parte medular del plan de pruebas, lista los casos de prueba que se usan durante las pruebas. Cada caso de prueba se describe con detalle en un documento aparte de Especificación del caso de prueba. Cada ejecución de estas pruebas se documentará en un *Reporte de incidentes de la prueba*. Más adelante en esta sección describimos con mayor detalle estos documentos.

La sección 10 del plan de pruebas cubre las responsabilidades, personal y necesidades de entrenamiento, riesgos y contingencias y el calendario de pruebas.

El siguiente es un esquema de una *Especificación del caso de prueba*:

Especificación del caso de prueba

1. Identificador de la especificación del caso de prueba
2. Conceptos a probar
3. Especificaciones de entrada
4. Especificaciones de salida
5. Necesidades ambientales
6. Requerimientos procedurales especiales
7. Dependencias entre casos

El identificador de la Especificación del caso de prueba es el nombre del caso de prueba, y se usa para distinguirlo con respecto a otros casos de prueba. Las convenciones, como la denominación de los casos de prueba a partir de las características del componente a probar, permiten que los desarrolladores se refieran con más facilidad a los casos de prueba. La sección 2 del TCS lista los componentes a probar y las características que se ejercitan. La sección 3 lista las entradas requeridas para los casos de prueba. La sección 4 lista la salida esperada. Esta salida se calcula en forma manual o con un sistema competitivo (como un sistema heredado que se está reemplazando). La sección 5 lista la plataforma de hardware y software necesaria para ejecutar las pruebas, incluyendo los manejadores y *stubs* de prueba. La sección 6 lista cualquier restricción necesaria para ejecutar la prueba, como temporización, carga o intervención del operador. La sección 7 lista las dependencias con respecto a otros casos de prueba.

El *Reporte de incidentes de la prueba* lista los resultados actuales de la prueba y las fallas que se experimentaron. La descripción de los resultados debe incluir cuáles características se demostraron y si se satisficieron. Si se experimentó una falla, el reporte de análisis de la prueba debe contener información suficiente para permitir que se repita la falla. Las fallas de todos los *Reportes de incidentes de la prueba* se recopilan y listan en el *Reporte de resumen de las pruebas*, y luego son analizadas y priorizadas por los desarrolladores.

Tome en cuenta que el estándar IEEE [IEEE Std. 829-1991] para la documentación de pruebas de software usa un esquema un poco diferente que es más adecuado para organizaciones y sistemas más grandes. La sección 10, por ejemplo, es tratada en varias secciones en el estándar (responsabilidades, personal y necesidades de entrenamiento, calendarización, riesgos y contingencias).

9.5.3 Asignación de responsabilidades

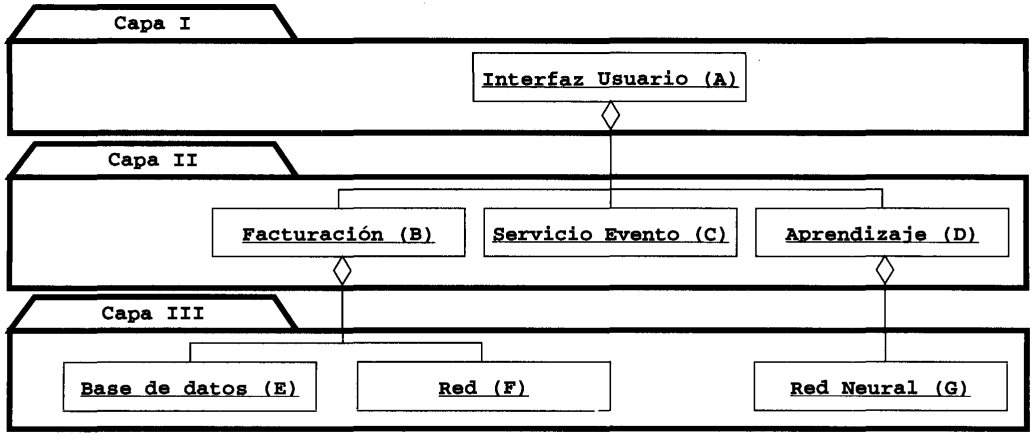
Las pruebas requieren que los desarrolladores encuentren defectos en los componentes del sistema. Esto se lleva a cabo mejor cuando realiza las pruebas un desarrollador que no esté involucrado en el desarrollo del componente que se va a probar, que esté menos reticente a romper el componente que se va a probar y que tenga más probabilidades de encontrar ambigüedades en la especificación del componente.

Para los requerimientos de calidad rigurosos, un equipo aparte dedicado al control de calidad es el único responsable de las pruebas. Al equipo de pruebas se le proporcionan los modelos del sistema, el código fuente y el sistema para desarrollar y ejecutar los casos de prueba. Los *Reportes de incidentes de las pruebas* y los *Reportes de resumen de pruebas* se envían de vuelta a los equipos de subsistemas para su análisis y posible revisión del sistema. Luego el equipo de revisión vuelve a probar el sistema revisado, no sólo para comprobar si se corrigieron las fallas originales sino también para asegurarse que no se hayan insertado nuevos defectos en el sistema.

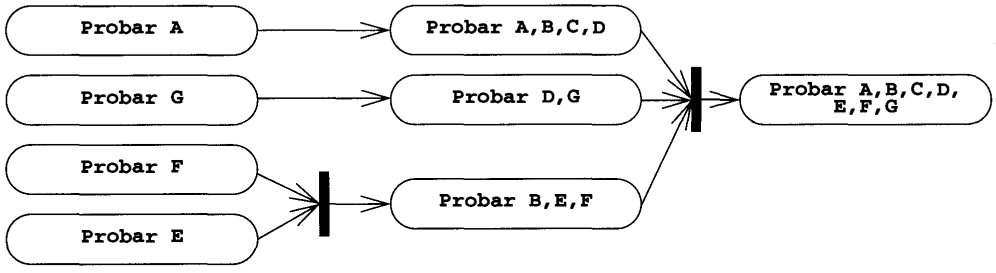
Para sistemas que no tiene requerimientos de calidad rigurosos, los equipos de subsistemas pueden duplicarse como equipos de pruebas para los componentes desarrollados por otros equipos de subsistemas. El equipo de arquitectura puede definir los estándares para los procedimientos de prueba, manejadores y *stubs*, y también puede comportarse como el equipo de pruebas de integración. Pueden usarse los mismos documentos para la comunicación entre los equipos de subsistemas.

9.6 Ejercicios

1. Corrija los defectos de los métodos `esAñoBisiesto()` y `obtenerNumDíasEnMes()` de la figura 9-14 y genere casos de prueba usando los métodos de prueba de ruta. ¿Son diferentes los casos de prueba que encuentra con respecto a los de las figuras 9-14 y 9-15? ¿Por qué? ¿Los casos de prueba que encontró descubrirían los defectos que corrigió?
2. Genere código Java equivalente para el diagrama de gráfica de estado del caso de uso `AjustarHora` del Reloj2B (figura 9-16). Use la prueba de equivalencia, la prueba de frontera y la prueba de ruta para generar casos de prueba para el código que acaba de generar. ¿Cómo se comparan estos casos con los generados usando pruebas basadas en estado?
3. Construya el diagrama de gráfica de estado correspondiente al caso de uso `ComprarBoleto` de la figura 9-22. Genere casos de prueba basados en el diagrama de gráfica de estado usando la técnica de prueba basada en estado. Discuta la cantidad de casos de prueba y las diferencias con el caso de prueba de la figura 9-23.
4. Discuta el uso de otros patrones de diseño para implementar un *stub* en vez del patrón Puente que se muestra en la figura 9-11. Por ejemplo, ¿cuáles son las ventajas de usar un patrón Apoderado en vez de un Puente? ¿Cuáles son las desventajas?
5. Aplique la ingeniería de software y la terminología de pruebas de este capítulo a los siguientes términos usados en el artículo de Feynman mencionado en la introducción:
   - ¿Qué es una "grieta"?
   - ¿Qué es una "iniciación de grieta"?
   - ¿Qué es "alta confiabilidad del motor"?
   - ¿Qué es un "propósito de diseño"?
   - ¿Qué es una "misión equivalente"?
   - ¿Qué significa "10% de la especificación original"?
   - ¿Cómo usa Feynman el término "verificación" cuando dice que "conforme se observan deficiencias y errores de diseño se corrigen y verifican con más pruebas"?
6. Dada la siguiente descomposición en subsistemas:



comente el plan de pruebas usado por el gerente del proyecto:



¿Qué decisiones se tomaron? ¿Por qué? ¿Cuáles son las ventajas y desventajas de este plan de pruebas particular?

Referencias

[Bezier, 1990] B. Bezier, *Software Testing Techniques,* 2a. ed., Van Nostrand, Nueva York, 1990.

[Binder, 1994] R. V. Binder, "Testing object-oriented systems: A status report", *American Programmer,* abril de 1994.

[Brooks, 1975] F. P. Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, MA, 1975.

[Fagan, 1976] M. Fagan, "Design and code inspections to reduce errors in program development", *IBM Systems Journal*, vol. 15, núm. 3, 1976.

[Feynman, 1988] R. P. Feynman, "Personal observations on the reliability of the Shuttle", Rogers Commission, *The Presidential Commission on the Space Shuttle Challenger Accident Report.* Washington, DC, junio de 1986. También en http://www.virtualschool.edu/mon./ SocialConstruction/ FeynmanChallengerRpt.html.

[IEEE Std. 829-1991] *IEEE Standard for Software Test Documentation*, IEEE Standards Board, marzo de 1991, en [IEEE 1997].

[IEEE Std. 982-1989] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Standards Board, julio de 1989, en [IEEE 1997].

[IEEE, 1997] *IEEE Standards Collection Software Engineering*, Piscataway, NJ, 1997.

[Jones, 1977] T. C. Jones, "Programmer Quality and Programmer Productivity", IBM TR-02.764, 1977.

[McCabe, 1976] T. McCabe, "A Software Complexity Measure", *IEEE Transactions on Software Engineering*, vol. 2, núm. 12, diciembre de 1976.

[Myers, 1979] G. J. Myers, *The Art of Software Testing*, Wiley, Nueva York, 1979.

[Parnas y Weiss, 1985] D. L. Parnas y D. M. Weiss, "Active design reviews: principles and practice", *Proceedings of the Eight International Conference on Software Engineering*, agosto de 1985, págs. 132–136.

[Pfleeger, 1991] S. L. Pfleeger, *Software Engineering: The Production of Quality Software*, 2a. ed., Macmillan, 1991.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach.* Clarendon, Oxford, 1992.

[Siewiorek y Swarz, 1992] D. P. Siewiorek y R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2a. ed., Digital, Burlington, MA, 1992.

[Turner y Robson, 1993] C. D. Turner y D. J. Robson, "The state-based testing of object-oriented programs", *Conference on Software Maintenance*, septiembre de 1993, págs. 302–310.

[Walker *et al.*, 1980] B. J. Walker, R. A. Kemmerer y G. J. Popek, "Specification and verification of the UCLA Unix security kernel", *Communications of the ACM*, vol. 23, núm. 2, 1980, págs. 118–131.

10

10

Administración de la configuración del software

Quienes quieran repetir el pasado deben controlar la enseñanza de la historia.

—Frank Herbert, *en Chapterhouse: Dune*

El cambio se extiende por todo el proceso de desarrollo: los requerimientos cambian cuando los desarrolladores mejoran su comprensión del dominio de aplicación, el diseño del sistema cambia con nueva tecnología y objetivos de diseño, el diseño de objetos cambia con la identificación de objetos de solución y la implementación cambia conforme se descubren y se reparan los defectos. Estos cambios pueden tener un impacto en cada uno de los productos de trabajo, desde los modelos del sistema hasta el código fuente y la documentación. La *administración de la configuración del software* es el proceso de controlar y supervisar el cambio de los productos de trabajo. Una vez que se define una línea básica, la administración de la configuración del software minimiza los riesgos asociados con los cambios definiendo un proceso formal de aprobación y seguimiento de los cambios.

En este capítulo describimos las siguientes actividades:

- La *identificación de los artículos de configuración* es el modelado del sistema como un conjunto de componentes en evolución.
- La *administración de promociones* es la creación de versiones para los demás desarrolladores.
- La *administración de lanzamientos* es la creación de versiones para el cliente y los usuarios.
- La *administración de ramas* es la administración del desarrollo concurrente.
- La *administración de variantes* es la administración de las versiones que se pretende que coexistan.
- La *administración de cambios* es el manejo, aprobación y seguimiento de las peticiones de cambio.

Concluimos este capítulo tratando los asuntos de la administración de proyectos relacionados con la administración de la configuración del software.

10.1 Introducción: un ejemplo de aeronave

Las aeronaves de pasajeros son una de las hazañas de ingeniería más complejas intentadas por una corporación privada. Por un lado, las aeronaves de pasajeros necesitan ser seguras y confiables. Por el otro, necesitan ser económicas. A diferencia de los trenes o los automóviles, los sistemas de aeronaves no pueden simplemente apagarse cuando sucede una falla. A diferencia de la NASA, las aerolíneas necesitan obtener ganancias y pagar dividendos a sus accionistas. Estos requerimientos dan como resultado diseños muy complejos que incluyen muchas redundancias y sistemas de diagnóstico. Un Boeing 747, por ejemplo, está compuesto por más de seis millones de partes. Para desarrollar y mantener sistemas tan complejos, y al mismo tiempo seguir siendo competitivos desde el punto de vista económico, los fabricantes de aeronaves extienden la vida de un modelo mejorando incrementalmente el mismo diseño. El Boeing 747, por ejemplo, se lanzó por primera vez en 1967 y sigue estando en producción al momento en que esto se escribe. Ha tenido cuatro revisiones mayores; la última, el 747-400, lanzada en 1988. Otro enfoque para el manejo de la gran complejidad de una aeronave es la reutilización de la mayor cantidad posible de partes y subsistemas en todos los modelos de aeronaves. Considere el siguiente ejemplo.

Airbus A320

En 1988, Airbus Industries, un consorcio de industrias aeronáuticas europeas, lanzó el A320, el primer avión de pasajeros de vuelo por programa alambrado. Los pilotos controlan el avión como un jet de combate F16 usando un pequeño bastón que se encuentra a su lado. Los impulsos digitales se transmiten luego a una computadora que los interpreta y los pasa a los controles de las alas y la cola. A diferencia de los controles hidráulicos, el control computarizado permite protección de envoltura, que impide que el piloto se exceda de determinados parámetros, como las velocidades mínima y máxima, el ángulo de ataque máximo y valores G máximos. El A320 tiene 150 asientos y está destinado para rutas de corto y mediano alcance.

A321 y A319

El éxito comercial del A320 permitió que Airbus trabajara en dos aeronaves derivadas, el A319, una versión más pequeña con 124 asientos, y el A321, una versión más grande con 185 asientos. El cambio de la longitud de un diseño básico para obtener una nueva aeronave ha sido una práctica estándar en la industria. Esto permite que el fabricante ahorre costo y tiempo apoyándose en un diseño existente. También ahorra costos de operación para las aerolíneas, ya que sólo necesitan mantener un lote de partes de repuesto. Sin embargo, en este caso Airbus llevó el concepto más lejos asegurando que las tres aeronaves tuvieran los mismos controles de cabina y las mismas características de manejo. El A321, por ejemplo, tiene alerones ranurados y controles de alas ligeramente modificados para hacer que el avión se sienta como el A320, aunque el A321 sea más largo. En consecuencia, cualquier piloto certificado para alguna de las aeronaves puede volar las otras dos. Esto da como resultado ahorros en costo para los operadores de aerolíneas, quienes necesitan entrenar a los pilotos sólo una vez, compartir simuladores de vuelo, partes de repuesto y cuadrillas de mantenimiento para las tres versiones.

A330 y A340

Llevando la filosofía aún más lejos, Airbus puso mucho cuidado en las características de manejo del A330 y el A340. Estas dos aeronaves, construidas para los mercados de largo alcance y ultra largo alcance, pueden transportar el doble de pasajeros que el A320 hasta tres veces más lejos. Tienen la misma disposición de cabina y sistema de vuelo por programa alambrado. Los pilotos entrenados para la familia A320 pueden volar el A330 o el A340 con un reentrenamiento mínimo, reduciendo aún más los costos de operación de la aerolínea. En comparación, las características de manejo de un 737 (comparable con el A319) son muy diferentes de las del 747 (comparable con la versión más grande del A340).

Los refinamientos incrementales y la reutilización de subsistemas no se logran sin problemas. Cada cambio que se hace a una sola aeronave necesita ser evaluado de manera minuciosa en el contexto de las otras dos. Por ejemplo, si decidimos instalar una planta de energía nueva más eficiente en el A320, necesitaríamos evaluar si la misma planta de energía también puede usarse en el A319 y el A321 para conservar la ventaja de compartir partes en los tres modelos. Luego necesitamos evaluar si el manejo de cada aeronave cambia en forma significativa, ya que, en ese caso, tal vez necesitemos modificar el software del control computarizado para que tome esto en cuenta. Si no fuera así, sería necesario volver a entrenar y certificar a todos los pilotos que vuelan el A320, sin mencionar la pérdida de una plataforma y un programa de entrenamiento comunes para las tres aeronaves. Luego se necesitaría que la autoridad gubernamental (por ejemplo, la Agencia de Aviación Federal en Estados Unidos, las Autoridades de Aviación Conjuntas en la Unión Europea) volviera a certificar la aeronave modificada para que pudieran decidir sobre la seguridad de la misma y si se necesitan nuevos procedimientos de mantenimiento o entrenamiento de pilotos. Por último, necesitaría documentarse con cuidado el estado de cada aeronave individual para que se reemplazaran las partes correctas. En resumen, es necesario identificar y rechazar cualquier cambio que pudiera amenazar la seguridad de la aeronave, su eficiencia o la movilidad de los pilotos entre aeronaves. Estos problemas requieren que los fabricantes de aeronaves y los operadores sigan procedimientos complejos de control de versiones y cambios.

El desarrollo de sistemas de software, aunque por lo general no es tan complejo como una aeronave de pasajeros, también sufre problemas similares. Los sistemas de software tienen un ciclo de vida largo para permitir la recuperación de la inversión inicial: por ejemplo, muchos sistemas Unix todavía contienen código que se remonta a los setenta. Los sistemas de software pueden existir en diferentes variantes; por ejemplo, hay versiones de sistemas operativos Unix que ejecutan en mainframes y también en PC caseras y computadoras Macintosh. Los cambios de mantenimiento a sistemas de software existentes y en ejecución necesitan controlarse y evaluarse en forma meticulosa para asegurar un determinado nivel de confiabilidad: por ejemplo, la introducción de la divisa Euro, un simple cambio de escala, ha tenido un impacto muy visible y considerable en muchos sistemas de software financieros y de negocios por todo el mundo. Por último, los sistemas de software evolucionan mucho más rápido que las aeronaves, lo que requiere que los desarrolladores sigan el cambio, sus suposiciones y su impacto.

La administración de la configuración permite que los desarrolladores y fabricantes de aeronaves manejen el cambio. La primera función de la administración de la configuración es la identificación de los artículos de configuración. ¿Cuáles subsistemas tienen probabilidad de cambio? ¿Cuáles interfaces de subsistemas no deben cambiar? Cada subsistema que tiene probabilidades de cambiar se modela como un artículo de configuración y se etiqueta su estado con un número de versión. El software de vuelo por programa alambrado del A320 es un artículo de configuración. El manejador de dispositivo para un puerto serial es un artículo de configuración para un sistema operativo Unix.

La segunda función de la administración de la configuración es administrar el cambio mediante un proceso formal. Una petición de cambio primero se registra, luego se analiza, y se acepta si es consistente con los objetivos del proyecto. Una petición de cambio para una aeronave es un reporte grueso que enumera todos los subsistemas y contratistas implicados en el cambio. Una petición de cambio para un sistema de software simple puede ser tan sólo un correo electrónico que solicita una nueva característica. Luego se aprueba o rechaza el cambio, dependiendo del impacto previsto del cambio sobre el sistema general.

Por último, la tercera función de la administración de la configuración es registrar suficiente información de estado sobre cada versión de cada artículo de configuración y sus dependencias. Viendo el libro de mantenimiento de un A320 y el número de versión de sus subsistemas, un ingeniero de mantenimiento puede decir cuáles subsistemas necesitan reemplazarse o mejorarse. Viendo la versión más reciente de un manejador de dispositivo de puerto serial, sus mejoras y los cambios desde la última versión, podemos determinar si debemos mejorar hacia un nuevo manejador o no.

La administración de la configuración del software se ha tratado en forma tradicional como un tema de mantenimiento. Sin embargo, se ha hecho difusa la distinción entre desarrollo y mantenimiento y, con frecuencia, se introduce tempranamente la administración de la configuración en el proceso. En este capítulo nos enfocamos sobre todo en las primeras fases de la administración de la configuración y tratamos su uso durante el mantenimiento en forma breve. Pero primero definimos de manera más formal el concepto de administración de la configuración.

10.2 Un panorama de la administración de la configuración

La **administración de la configuración del software** (a la que mencionaremos de aquí en adelante simplemente como administración de la configuración) es la disciplina de administrar y controlar los cambios en la evolución de los sistemas de software [IEEE Std. 1042-1987]. Los sistemas de administración de la configuración automatizan la identificación de versiones, su almacenamiento y recuperación, y soportan la contabilización del estado. La administración de la configuración incluye las siguientes actividades:

- **Identificación de los artículos de configuración.** Se identifican y etiquetan los componentes del sistema y sus productos de trabajo y versiones en forma única. Los desarrolladores identifican los artículos de configuración después del *Acuerdo del proyecto* (sección 11.4.5), una vez que se han acordado los principales productos a entregar y componentes del sistema. Los desarrolladores crean versiones y artículos de configuración adicionales conforme evoluciona el sistema.
- **Control del cambio.** Se controlan los cambios al sistema y las versiones para los usuarios para asegurar la consistencia con los objetivos del proyecto. El control del cambio puede ser realizado por los desarrolladores, los gerentes o un comité de control, dependiendo del nivel de calidad requerido y la tasa de cambios.
- **Contabilización del estado.** Se registra el estado de los componentes individuales, productos de trabajo y peticiones de cambio. Esto permite a los desarrolladores distinguir con más facilidad entre versiones y dar seguimiento a los problemas relacionados con los cambios. Esto también permite que la gerencia dé seguimiento al estado del proyecto.
- **Auditoría.** Las versiones seleccionadas para entrega se validan para asegurar la suficiencia, consistencia y calidad del producto. El equipo de control de calidad realiza la auditoría.

Además, también se consideran muchas veces parte de la administración de la configuración las siguientes actividades [Dart, 1991].

- **Administración de la construcción.** La mayoría de los sistemas de administración de la configuración permiten la construcción automática del sistema conforme los desarrolladores crean nuevas versiones de los componentes. El sistema de administración de la configuración tiene suficiente conocimiento del sistema para minimizar la cantidad de recompilación. También puede ser capaz de combinar diferentes versiones de los componentes

para construir diferentes variantes del sistema (por ejemplo, para diferentes sistemas operativos y plataformas de hardware).

- **Administración de procesos.** Además del control del cambio, los proyectos pueden tener políticas acerca de la creación y documentación de versiones. Una de estas políticas puede ser que sólo forme parte de una versión el código sintácticamente correcto. Otra política puede ser que las construcciones se intenten (y tengan éxito) cada semana. Por último, el proceso de administración de la configuración incluye políticas para la notificación a desarrolladores relevantes cuando se crean nuevas versiones o cuando falla una construcción. Algunos sistemas de administración de la configuración permiten que los desarrolladores automaticen tales flujos de trabajo.

Por tradición se ha visto la administración de la configuración como una disciplina administrativa que ayuda a los gerentes de proyecto en las actividades de control del cambio, contabilización del estado y auditoría ([Bersoff *et al.*, 1980] e [IEEE Std. 1042-1987]). Sin embargo, en fechas más recientes, también se ha visto la administración de la configuración como una disciplina de apoyo al desarrollo, ayudando a los desarrolladores a manejar la complejidad asociada con gran cantidad de cambios, componentes y variantes [Babich, 1986]. En este capítulo nos enfocamos con mayor detalle en esta última perspectiva y sólo tratamos en forma breve las actividades de control del cambio v contabilización del estado.

En nuestra opinión, la administración de la configuración se extiende por todo el ciclo de vida del software. Comienza con la identificación de los artículos de configuración después de que se han definido los productos a entregar y los componentes principales del sistema. Continúa a lo largo del desarrollo conforme los desarrolladores crean versiones de los productos de trabajo durante el análisis, diseño del sistema, diseño de objetos e implementación. Junto con la administración de la fundamentación (vea el capítulo 8, *Administración de la fundamentación*), la administración de la configuración es la principal herramienta de que disponen los desarrolladores para manejar el cambio.

A continuación nos centraremos con más detalle en los conceptos de la administración de la configuración.

10.3 Conceptos de la administración de la configuración

En esta sección presentamos los conceptos principales de la administración de la configuración (figura 10-1). Con tanta frecuencia como sea posible, usaremos la misma terminología utilizada en los lineamientos de la IEEE sobre la administración de la configuración [IEEE Std. 1042-1987]:

- Un **artículo de configuración** (llamado ArC para abreviar) es un producto de trabajo o un fragmento de software que se trata como una sola entidad para efectos de la administración de la configuración. Un conjunto de artículos de configuración se define como un **agregado de administración de la configuración** (llamado agregado AC para abreviar). El software del vuelo por programa alambrado del A320 es un artículo de configuración. El A320 es un agregado AC. El manejador de dispositivo de puerto serial es un artículo de configuración. El sistema operativo Linux es un agregado AC.
- Una **petición de cambio** es un reporte formal, hecho por un usuario o un desarrollador, que solicita una modificación a un artículo de configuración. Por ejemplo, la *Propuesta de cambio de ingeniería* [MIL Std. 480], el formulario estándar de petición de cambio del

gobierno de Estados Unidos, es de siete páginas. Una petición de cambio informal puede ser un mensaje de correo electrónico en línea.

- Una **versión** identifica el estado de un artículo de configuración o de una configuración en un momento bien definido. Para un agregado AC dado, a un conjunto de versiones consistente de sus artículos de configuración se le define como una **configuración**. Puede verse una configuración como una versión de un agregado AC.
- Una **promoción** es una versión que se ha puesto a disposición de los demás desarrolladores del proyecto. Un **lanzamiento** es una versión que se ha puesto a disposición de los clientes o usuarios.
- Un **depósito** es una biblioteca de lanzamientos. Un **espacio de trabajo** es una biblioteca de promociones.

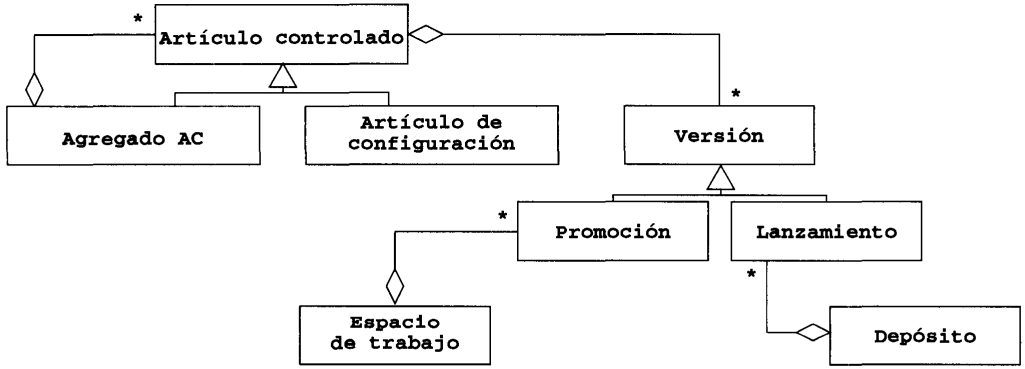


Figura 10-1 Conceptos de la administración de la configuración (diagrama de clase UML).

10.3.1 Artículos de configuración y agregados AC

Un **artículo de configuración** es un producto de trabajo, o un componente de un producto de trabajo, que está bajo la administración de la configuración, y para estos efectos se le trata como una sola entidad. Por ejemplo, el software de vuelo por programa alambrado del A320 es un artículo de configuración (vea la figura 10-2). Durante una mejora se reemplaza el software completo. El software de vuelo por programa alambrado no puede dividirse en componentes más pequeños que puedan instalarse en forma independiente. Del mismo modo, el manejador de dispositivo para un puerto serial en cualquier sistema operativo es un artículo de configuración. Este componente es tan simple que ya no puede dividirse en lo que concierne a la instalación.

Un agregado AC es una composición de artículos de configuración. El 747 es un agregado AC de seis millones de partes. El sistema operativo Linux[1] es un agregado AC que incluye un calendarizador de procesos, un administrador de memoria, muchos manejadores de dispositivo, demonios de red, sistemas de archivo y muchos otros subsistemas.

1. Linux es un sistema operativo POSIX [POSIX, 1990] disponible en forma gratuita creado por Linus Torwalds. Para mayor información, vea http://www.linux.org/.

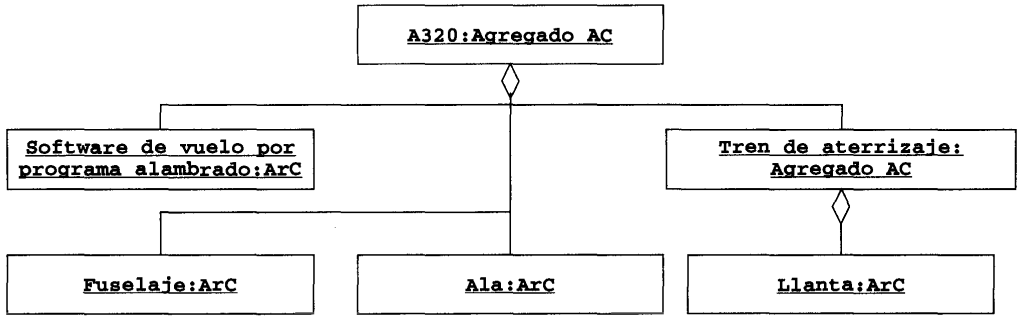


Figura 10-2 Un ejemplo de agregados AC y artículos de configuración (diagrama de objetos UML).

10.3.2 Versiones y configuraciones

Una **versión** identifica el estado de un artículo de configuración en un momento del tiempo bien definido. Las versiones sucesivas de un producto de trabajo difieren por uno o más cambios, como la corrección de un defecto, la adición de nueva funcionalidad o la eliminación de funcionalidad innecesaria u obsoleta. Una **configuración** identifica el estado de un agregado AC.

Una **línea base** es una versión de un artículo de configuración que ha sido revisado y acordado de manera formal por la gerencia o el cliente y que sólo puede cambiarse mediante una petición de cambio. Por ejemplo, cada aeronave debe pasar por un proceso de certificación riguroso de una agencia gubernamental (por ejemplo, la FAA en Estados Unidos o la JAA en la Unión Europea) antes de que pueda operarla una aerolínea. Cualquier cambio a la aeronave después de la certificación requiere un proceso de cambio formal y volver a pasar el proceso de certificación. Esto asegura que los cambios sean consistentes con los objetivos del proyecto (por ejemplo, seguridad y confiabilidad) y los reglamentos, y que se comuniquen a los desarrolladores y usuarios relevantes (por ejemplo, los pasajeros y el operador de la aerolínea).

A las versiones que se pretende que coexistan se les llama **variantes**. Por ejemplo (figura 10-3), el A319, el A320 y el A321 son variantes de la misma aeronave básica. La diferencia principal es su longitud; esto es, la cantidad de pasajeros y carga que pueden transportar. Sin embargo, el A320-200 es una versión del A320 que reemplaza a la versión inicial. En otras palabras, una aerolínea puede comprar el A319 y el A320-200, pero no puede comprar el A320-100, más antiguo. En el caso de un sistema de software, el sistema puede tener una variante Macintosh, una variante Windows y una variante Linux, proporcionando cada una funcionalidad idéntica. Un sistema también puede tener una variante estándar y una profesional o de lujo que soporten diferentes rangos de funcionalidad. Las variantes comparten gran cantidad de código que implementa la funcionalidad medular y las diferencias están confinadas a una pequeña cantidad de subsistemas de nivel más bajo.

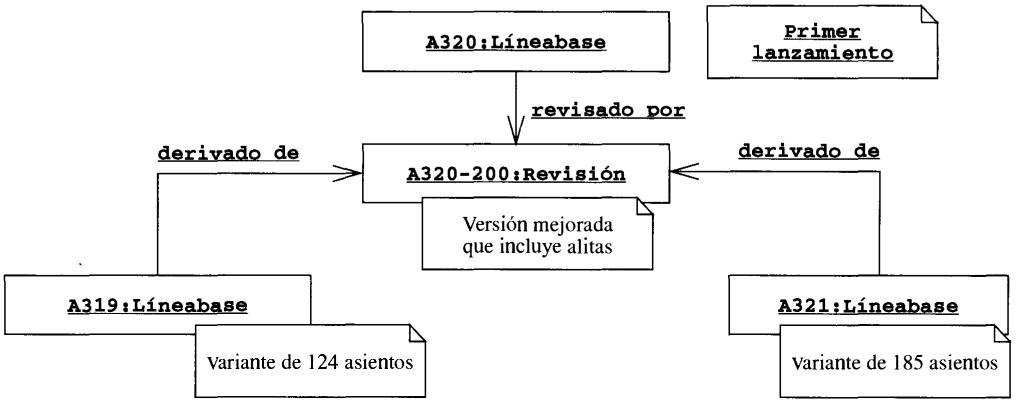


Figura 10-3 Ejemplos de líneas base, revisiones y variantes (diagrama de objetos UML). El A319, el A320 y el A321 se basan en el mismo diseño. Varían sobre todo en la longitud del fuselaje.

10.3.3 Peticiones de cambio

Una **petición de cambio** es un paso formal que inicia el proceso de cambio. Un usuario, cliente o desarrollador descubre un defecto en un producto de trabajo y quiere una nueva característica. El autor de la petición de cambio especifica el artículo de configuración al que se aplica la petición, su versión, el problema que necesita resolverse y la solución propuesta. En el caso de un proceso de cambio formal, luego se valoran los costos y beneficios del cambio antes de que se apruebe o rechace el cambio. En ambos casos, las razones de la decisión se registran con la petición de cambio.

Por ejemplo, poco después de que se certificó al A320, en febrero de 1988, se volvió a certificar una versión revisada, el A320-200. La nueva versión incluía unas cuantas modificaciones que producían un rango más largo y un peso de despegue más grande. Uno de estos cambios fue la adición de alitas al extremo de las alas que reducen significativamente la resistencia al avance y, por tanto, reducen el consumo de combustible. El consumo de combustible es un costo principal para el operador de la aeronave y, por consiguiente, un fuerte argumento de venta para el fabricante de la aeronave. Durante el diseño del A320 se envió una petición de cambio que describía el cambio de alitas, la evaluación de su desempeño y su costo estimado. El cambio se aprobó e implementó, y el A320-200 pasó la certificación en noviembre de 1988. Una petición de cambio para el kernel del sistema operativo Linux es un mensaje de correo electrónico para Linus Torwalds.

10.3.4 Promociones y lanzamientos

Una **promoción** es una versión que se pone a disposición de otros desarrolladores. Una promoción indica a un artículo de configuración que ha alcanzado un estado relativamente estable y que pueden usarlo o revisarlo otros desarrolladores. Por ejemplo, se promueven los subsistemas para los demás equipos que los usan. Luego se promueven para que el equipo de control de calidad valore su calidad. Después, conforme se descubren y reparan defectos, se promueven las revisiones del subsistema para reevaluación.

Un **lanzamiento** es una versión que se pone a disposición de los usuarios. Un lanzamiento indica que un artículo de configuración ha satisfecho los criterios de calidad establecidos por el equipo de control de calidad y pueden usarlo o revisarlo los usuarios. Por ejemplo, el sistema se lanza a los probadores beta para que encuentren defectos adicionales y valoren la calidad percibida del sistema. Conforme se descubren y reparan defectos, se promueven las revisiones del subsistema para reevaluación por parte del equipo de control de calidad y, cuando se satisfacen los criterios de calidad, se vuelve a lanzar a los usuarios.

10.3.5 Depósitos y espacios de trabajo

Los sistemas de administración de la configuración proporcionan un **depósito** en el que se guardan y se da seguimiento a todos los artículos de configuración, y un **espacio de trabajo** donde el desarrollador hace cambios. Los desarrolladores crean nuevas versiones enviando los cambios desde su espacio de trabajo hacia el depósito.

Una **biblioteca de software**, como la define el estándar [IEEE Std. 1042-1987] proporciona facilidades para guardar, etiquetar e identificar versiones de los artículos de configuración (es decir, documentación, modelos y código). Una biblioteca de software también proporciona funcionalidad para el seguimiento del estado de los cambios en los artículos de configuración. Distinguimos tres tipos de bibliotecas.

1. El **espacio de trabajo del desarrollador**, también conocido como biblioteca dinámica, se usa para el desarrollo diario de los desarrolladores. Los cambios no están restringidos y sólo los controla el desarrollador individual.
2. El **directorio maestro**, también conocido como biblioteca controlada, lleva cuenta de las promociones. Los cambios necesitan aprobarse y las versiones necesitan satisfacer determinados criterios del proyecto (por ejemplo, "sólo puede entrar el código que compile sin errores") antes de que se pongan a disposición del resto del proyecto.
3. El **depósito de software**, también conocido como biblioteca estática, lleva cuenta de los lanzamientos. Las promociones necesitan satisfacer determinados criterios de control de calidad (por ejemplo, "deben estar reparados todos los defectos detectados por pruebas de regresión") antes de que una promoción se convierta en lanzamiento.

10.3.6 Esquemas de identificación de versiones

Las versiones son identificadas en forma única por los desarrolladores y sistemas usando un **identificador de versión**, al que también se llama número de versión. Algunos ejemplos exóticos incluyen lo siguiente:

- La especificación Ada pasó por cinco versiones principales sucesivas, llamadas Strawman, Woodenman, Tinman, Ironman y Steelman [Steelman, 1978].
- El esquema de identificación de versión para TEX, un programa de tipografía para textos técnicos [Knuth, 1986], se basa en decimales del número π: cada vez que se encuentra un error (que es raro) y se repara, el número de versión de TEX se incrementa para añadir otro dígito. La versión actual es 3.14159.

Sin embargo, por lo general los números de versión de un artículo de configuración pueden ser algo grandes. En este caso, los desarrolladores y los sistemas para la administración de la configuración usan esquemas de identificación de versión que soportan con más facilidad la automatización, como números secuenciales con dos o tres decimales. Por ejemplo, considere un editor UML llamado MUE (por My UML Editor), el cual se construyó y lanzó en forma incremental. Podemos usar un esquema de tres dígitos para distinguir entre cambios funcionales, pequeñas mejoras y corrección de errores (figura 10-4). El dígito de la extrema izquierda indica la versión mayor (por ejemplo, revisión general de la funcionalidad o de la interfaz de usuario), el segundo dígito la versión menor (por ejemplo, adición de funcionalidad limitada) y el tercer dígito las revisiones (por ejemplo, correcciones). Por convención, las versiones anteriores a la 1.0.0 indican versiones lanzadas con el propósito de realizar pruebas alfa o beta.

Sin embargo, este esquema secuencial simple sólo funciona para una serie secuencial de versiones. Una **rama** identifica una ruta de desarrollo concurrente que requiere administración de configuración independiente. Los lanzamientos son vistos por el usuario como un proceso de desarrollo incremental y secuencial. Sin embargo, diferentes equipos pueden realizar el desarrollo de diferentes características en forma concurrente y combinarlas después en una sola versión. La secuencia de versiones creadas por cada equipo es una rama que es independiente de las versiones creadas por los demás equipos. Cuando necesitan reconciliarse las versiones de diferentes ramas se **intercalan** las versiones; esto es, se crea una nueva versión que contiene elementos seleccionados de las versiones predecesoras.

Para los lanzamientos es suficiente, por lo general, el esquema de identificación secuencial, debido a que el concepto de rama no es visible para los usuarios. Sin embargo, para los desarrolladores y los sistemas de administración de configuración esto no es suficiente, debido a que a

Esquema de identificación de versión de tres dígitos

<versión> ::= <nombre del artículo de configuración >.<mayor>.<menor>.<revisión>

<mayor> ::= <entero no negativo>

<menor> ::= <entero no negativo>

<revisión> ::= <entero no negativo>

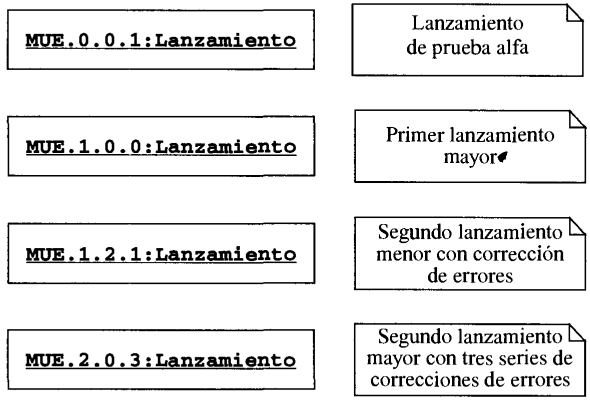


Figura 10-4 Esquema de identificación de versiones de tres dígitos (BNF y diagrama de objetos UML).

menudo se usan las ramas para soportar el desarrollo concurrente. El esquema de identificación de versiones usado por CVS [Berliner, 1990], por ejemplo, representa en forma explícita las ramas y versiones. Los números de versión incluyen un identificador de rama seguido por un número de revisión. El identificador de rama incluye el número de versión en el que se inició la rama seguido por un número único que identifica la rama. Esto permite que los desarrolladores identifiquen a qué rama pertenece una versión y en qué orden se produjeron las versiones de una rama.

En la figura 10-5 se muestran dos ramas: el tronco principal (el paquete de la izquierda) y la rama `1.2.1` que se deriva de la versión `1.2` (paquete de la derecha). En el ejemplo MUE, la rama pudo haberse derivado para efectos de evaluación de incrementos competitivos de la misma característica (por ejemplo, soporte para diagramas de interacción UML en MUE). Observe que este esquema de identificación no identifica la manera en que se vuelven a combinar las versiones

Esquema de identificación de versión CVS

<versión> ::= <nombre del artículo de configuración >.<identificador de versión>

<identificador de versión> ::= <rama>.<revisión>

<rama> ::= <identificador de versión>.<número de rama > | <número de rama >

<número de rama> ::= <entero no negativo>

<revisión> ::= <entero no negativo>

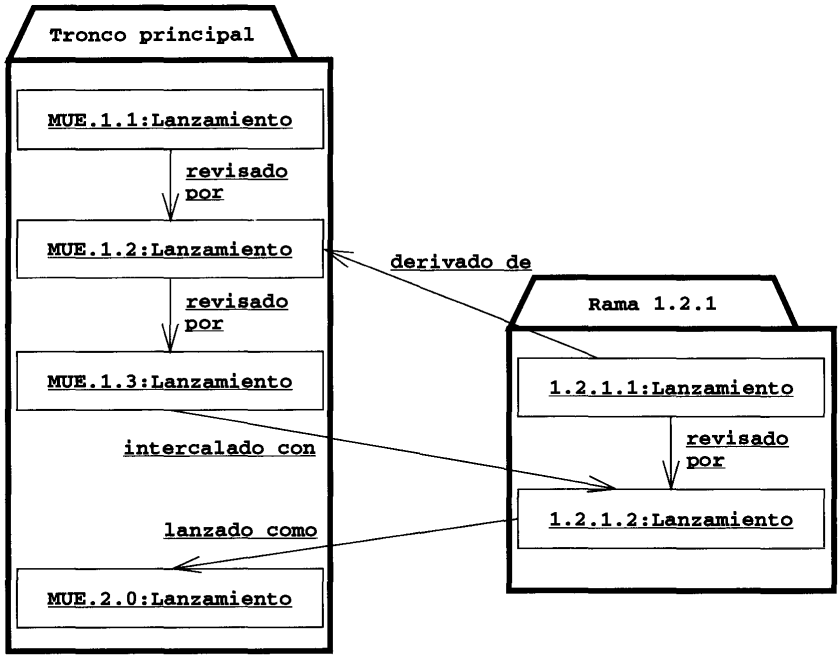


Figura 10-5 Esquema de identificación de versiones CVS (diagrama de objetos UML). Las ramas se identifican con la versión de la que se derivaron seguidas por un número único.

hacia una rama. En la figura 10-5 la versión `1.2.1.2` se combina con la versión `1.3` del tronco principal para producir la versión `2.0`.

10.3.7 Cambios y conjuntos de cambio

La evolución de un artículo de configuración puede modelarse de dos formas.

- **Vista basada en estado.** Como una serie de versiones, esto es, como una serie de estados del artículo de configuración. Cada estado se identifica con un número de versión (por ejemplo, el A320-200, `MUE.1.0`). Ésta es la vista que se encuentra con más frecuencia en la práctica.
- **Vista basada en cambio.** Como una línea base seguida por una serie de **cambios**, también llamados **deltas**. Un cambio representa la diferencia entre dos versiones sucesivas desde el punto de vista de líneas o párrafos que se han añadido o eliminado del artículo de configuración. Con frecuencia, la reparación de un defecto o la adición de funcionalidad a un sistema requiere cambios a varios artículos de configuración. Todos los cambios a los artículos de configuración asociados con una sola revisión de una configuración se agrupan en un **conjunto de cambios**. Si dos conjuntos de cambios no se traslapan (es decir, se aplican a conjuntos de artículos de configuración diferentes y no relacionados), se les puede aplicar la misma línea base en un orden arbitrario, proporcionando así más flexibilidad al desarrollador cuando selecciona configuraciones.

Continuando con el ejemplo MUE, supongamos que revisamos dos veces la primera línea base: `MUE.1.1` fue lanzado para corregir un error relacionado con clases que no tienen operaciones y `MUE.1.2` fue lanzado para corregir un error relacionado con el trazado de líneas de guiones. Cada una de estas revisiones corresponde a cambios en un solo subsistema. Por tanto, el conjunto de cambios correspondiente a cada una de estas revisiones es independiente y puede aplicarse a la línea base en cualquier orden. Por ejemplo, cuando se aplica `correcciónClaseVacía:ConjuntoCambio` al `MUE.1.0:Lanzamiento` derivamos el `MUE.1.1:Lanzamiento`. Luego la aplicación de `correcciónLíneaGuiones:ConjuntoCambio` da como resultado al `MUE.1.2:Lanzamiento`. Si en vez de ello aplicáramos primero `correcciónLíneaGuiones:ConjuntoCambio` obtendríamos al `MUE.1.1a:Lanzamiento`. Sin embargo, al aplicar después la `correcciónClaseVacía:ConjuntoCambio` también resultaría el `MUE.1.2:Lanzamiento`. Sin embargo, el conjunto de cambios correspondiente a la segunda línea base depende de estas dos revisiones. La figura 10-6 ilustra la historia de lanzamientos del MUE como una serie de conjuntos de cambios y sus dependencias.

La vista basada en cambios de la administración de la configuración es más general que la vista basada en estado. Permite que el desarrollador vea versiones relacionadas de diferentes artículos de configuración como una sola acción. Además, cuando no se traslapan, los conjuntos de cambios pueden aplicarse a más de una versión. Este enfoque se usa para la entrega de correcciones de errores y mejoras pequeñas después de que ha sido lanzada una pieza de software. Cada conjunto de cambio se entrega como un parche separado que puede aplicarse en forma directa a la línea base entregada o a cualquier versión derivada. Mientras no se superpongan, los parches pueden aplicarse en cualquier orden a la línea base.

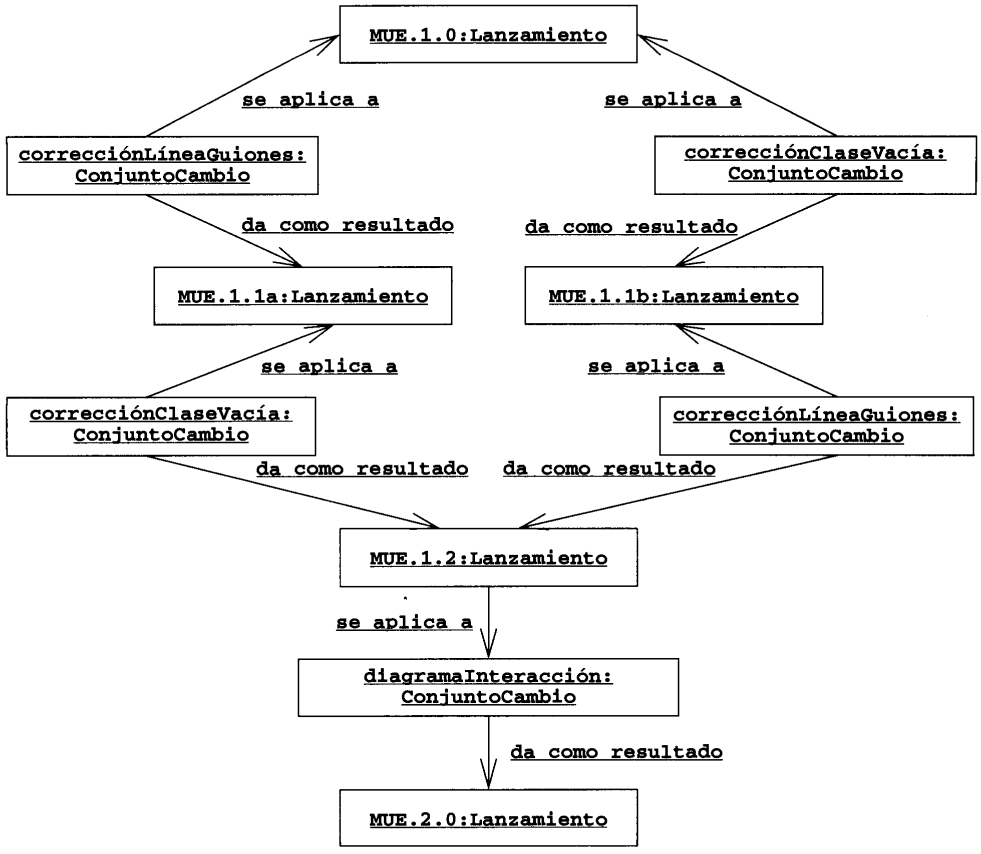


Figura 10-6 Representación de conjuntos de cambios de la historia de lanzamientos del MUE (diagrama de objetos UML). `correcciónLíneaGuiones:ConjuntoCambio` y `correcciónClaseVacía:ConjuntoCambio` pueden aplicarse al `MUE.1.0:Lanzamiento` en un orden arbitrario porque no se superponen.

10.3.8 Herramientas para la administración de la configuración

Debido a la importancia de la administración de la configuración en el desarrollo de software, los desarrolladores disponen de muchas herramientas de administración de la configuración y administración de versiones. En esta sección describimos en forma breve cuatro de ellas, RCS [Tichy, 1985], CVS [Berliner, 1990], Perforce [Perforce] y ClearCase [Leblang, 1994].

El sistema de control de revisiones (RCS, por sus siglas en inglés), una herramienta gratuita, controla un depósito que guarda todas las versiones de los artículos de configuración. Para obtener una versión específica los desarrolladores sacan una versión hacia su espacio de trabajo especificando un número de versión o una fecha. Para cambiar un artículo de configuración, el desarrollador necesita bloquearlo primero para impedir que los demás desarrolladores lo cambien. Cuando termina el cambio el desarrollador devuelve el artículo modificado al depósito, creando al mismo tiempo una nueva versión y liberando el bloqueo. Para optimizar el almacenamiento, RCS sólo

guarda la versión más reciente de cada artículo de configuración y las diferencias entre cada versión. El concepto de configuración puede realizarse añadiendo una etiqueta especificada por el desarrollador a todas las versiones que pertenecen a la configuración. Luego los desarrolladores pueden sacar un conjunto consistente de versiones usando la etiqueta. Observe que este enfoque no permite el control de versión de la configuración misma. RCS no soporta el concepto de rama.

El sistema de versiones concurrentes (CVS, por sus siglas en inglés), también es una herramienta gratuita, extiende al RCS con el concepto de rama. En vez de una secuencia de diferencias, CVS guarda un árbol de diferencias para cada artículo de configuración. CVS también proporciona herramientas para intercalar dos ramas y detectar superposiciones. La política de control de cambios del CVS también es diferente a la del RCS. En vez de bloquear artículos de configuración, CVS considera a cada desarrollador como una rama aparte. Si un solo desarrollador modifica un artículo de configuración entre dos devoluciones, CVS combina en forma automática la rama con el tronco principal. Si CVS detecta un cambio concurrente primero trata de intercalar los dos cambios y luego, en caso de superposición, notifica al último desarrollador que se registró. Con esta política, CVS puede soportar un nivel más alto de desarrollo concurrente que el RCS.

Perforce es un reemplazo comercial para el CVS. Se basa en el mismo concepto del depósito central que CVS y RCS. Sin embargo, Perforce también soporta el concepto de cambio y conjunto de cambios, permitiendo que los desarrolladores den seguimiento con más facilidad a los artículos de configuración que se involucraron en un cambio dado.

ClearCase, otra herramienta comercial, también soporta el concepto de agregados AC y configuraciones. Un agregado AC se realiza como un directorio, el cual es administrado como un artículo de configuración por ClearCase. Éste también permite la especificación de configuraciones con reglas, seleccionando versiones de cada artículo de configuración. Una versión puede especificarse con una regla estática (es decir, refiriéndose a un número de versión específico) o con una regla dinámica (por ejemplo, refiriéndose a la versión más reciente de artículo). ClearCase también proporciona mecanismos de control de acceso para definir la propiedad de cada artículo de configuración y de cada configuración.

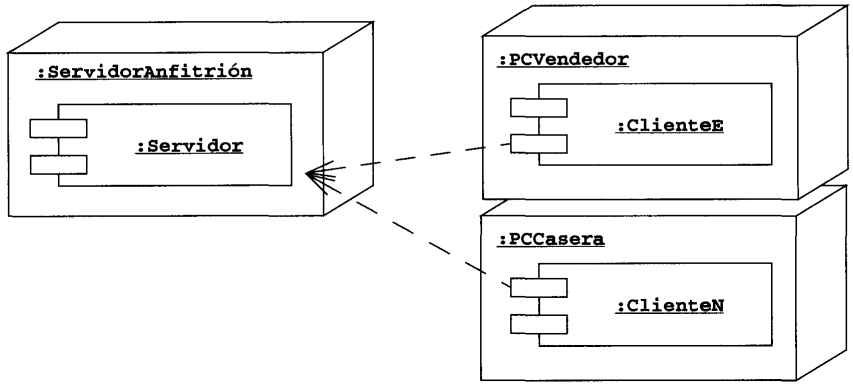
10.4 Actividades de la administración de la configuración

En la sección anterior describimos los conceptos principales de la administración de la configuración. En esta sección nos enfocamos en las actividades necesarias para definir y administrar los artículos de configuración, las promociones y los lanzamientos. También describimos actividades relacionadas con el uso de ramas y variantes para el desarrollo concurrente. Las actividades de administración de la configuración que se describen en esta sección incluyen:

- Identificación de artículos de configuración y agregados AC
- Administración de promociones
- Administración de lanzamientos
- Administración de ramas
- Administración de variantes
- Administración de cambios

Como base para los ejemplos de esta sección usamos un catálogo de partes de automóvil distribuido llamado MisPartesCarro. Éste permite que los vendedores y propietarios de automóviles examinen y soliciten partes desde su computadora, MisPartesCarro tiene una arquitectura

cliente/servidor con dos tipos de clientes: el `ClienteE` para usuarios expertos, como los mecánicos y vendedores de partes, y el `ClienteN` para los usuarios principiantes, que incluye a los propietarios de automóviles que reparan su propio automóvil. El sistema requiere que los usuarios se autentifiquen, lo que permite determinar la lista de precios de partes que se da al usuario particular. Un usuario que ordena muchas partes es elegible para descuentos por volumen, mientras que un cliente ocasional paga el precio de lista completo. El sistema también lleva cuenta de los intereses de cada usuario y usa esta información para optimizar el uso de la red. Al usuario también se le envían noticias personalizadas acerca de nuevos productos o nuevos descuentos en los precios. La figura 10-7 muestra la descomposición en subsistemas de MisPartesCarro.



:PCVendedor	La `:PCVendedor` es la máquina que usa un vendedor para solicitar partes. La `:PCVendedor` a menudo tiene un vínculo con ancho de banda más grande hacia el servidor.
:ClienteE	El `:ClienteE` se encuentra en la `:PCVendedor`. Proporciona funcionalidad a un usuario experto para que localice partes por identificador de parte, marca y año de vehículo e historia de pedidos. El `:ClienteE` está diseñado para clientes de alto volumen, como los talleres de reparación de automóviles y vendedores de partes.
:PCCasera	La `:PCCasera` es la máquina que usa un propietario de automóvil para solicitar partes. La `:PCCasera` está conectada al servidor mediante un módem.
:ClienteN	El `:ClienteN` se encuentra en la `:PCCasera`. Proporciona funcionalidad al usuario principiante para la localización de partes mediante la descripción y, en lanzamientos subsiguientes, haciendo clic en un plano del vehículo. El `:ClienteN` está diseñado para el cliente ocasional, como los aficionados a los automóviles.
:ServidorAnfitrión	El `:ServidorAnfitrión` aloja al servidor del catálogo de partes.
:Servidor	El `:Servidor` permite que los clientes recuperen listas de partes por criterio y entradas de partes, que soliciten partes y da seguimiento a las actividades de los clientes.

Figura 10-7 Descomposición en subsistemas de MisPartesCarro y asignación de hardware (diagrama de distribución UML).

El desarrollo y evolución del sistema MisPartesCarro requieren el lanzamiento coordinado de varios componentes.

- El protocolo usado por los clientes y el servidor para intercambiar información se mejora en forma ocasional para soportar nueva funcionalidad del cliente y para mejorar el tiempo de respuesta y la producción. Debido a que nada garantiza que todos los usuarios tengan la misma versión del cliente, es necesario que el servidor mantenga compatibilidad hacia atrás con clientes antiguos. Cada nueva revisión necesita probarse con clientes antiguos para validar esta propiedad.
- Nuevas versiones de los clientes pueden implementar funcionalidad que sólo es soportada por las nuevas versiones del servidor. Por tanto, el servidor necesita mejorarse primero antes de que se tengan disponibles las versiones de los clientes.
- Cuando se dispone de una nueva versión de cualquier cliente el usuario puede descargar un parche. Sin embargo, el manual impreso correspondiente se envía por correo normal. Si sucede que se lanzan varias versiones en un periodo corto, el usuario necesita poder identificar cuál manual corresponde a cuál versión.

Por ejemplo, supongamos que el lanzamiento inicial de MisPartesCarro permite que los usuarios examinen el catálogo usando sólo información textual. Las partes tienen un identificador de parte, un nombre, una descripción, la lista de vehículos y años para los que se fabrica esta parte y una referencia cruzada hacia otras partes con las cuales pueden ensamblarse. El usuario puede buscar en la base de datos usando cualquiera de estos campos. Esto puede ser problemático para los usuarios principiantes que no saben el nombre de la parte que están buscando. En un segundo lanzamiento resolvemos este problema proporcionando al usuario un plano de navegación por las partes. Si el usuario conoce la apariencia de la parte y en qué parte del vehículo se encuentra, sólo necesita hacer clic en la parte correspondiente del plano. El cambio para introducir planos de navegación y el lanzamiento de los componentes de MisPartesCarro necesita secuenciarse con cuidado para tomar en consideración las descripciones que dimos antes. El cambio de plano de navegación requiere la siguiente secuencia de pasos.

1. Es necesario modificar el `Servidor` para que soporte el almacenamiento y recuperación de planos de subensambles.
2. Es necesario que se lance e instale el `Servidor`.
3. Es necesario crear y guardar un plano por vehículo y año en la base de datos.
4. Es necesario modificar el `ClienteN` para que use los planos de navegación como una interfaz posible.
5. Es necesario lanzar e instalar el `ClienteN`.

En las siguientes secciones usamos MisPartesCarro y el cambio del plano de navegación como base para los ejemplos.

10.4.1 Identificación de los artículos de configuración y agregados AC

La identificación de los artículos de configuración y los agregados AC sucede, principalmente, después del acuerdo del proyecto, cuando se acuerda con el cliente un conjunto de productos a entregar, y después del diseño del sistema, cuando ya se ha identificado la mayoría de los subsistemas. Sin embargo, la identificación de los artículos de configuración y agregados AC continúa durante todo el desarrollo conforme se redefine el conjunto de productos a entregar, y se añaden y eliminan subsistemas de la descomposición en subsistemas.

La identificación de los artículos de configuración y agregados AC es similar a la identificación de objetos durante el análisis. No es algorítmica, debido a que es trivial la identificación de algunos artículos de configuración (por ejemplo, el RAD, el SDD) pero es más sutil la de otros (por ejemplo, la definición de un protocolo cliente servidor). Los artículos de configuración son documentos autocontenidos o fragmentos de código cuya evolución necesita rastrearse y controlarse. Éstos incluyen documentos a entregar, productos de trabajo, subsistemas, componentes comprados que pueden evolucionar durante el ciclo de vida del sistema y descripciones de interfaces. Por lo general, la mayoría de los artículos de configuración ya se han identificado como objetos de interés durante la planeación del proyecto o el diseño del sistema.

En el ejemplo MisPartesCarro identificamos cada uno de los documentos a entregar como un artículo de configuración. En el *Acuerdo de proyecto* de MisPartesCarro se definieron los siguientes como productos a entregar:

- Documentos en el nivel de usuario, incluyendo el RAD y el Manual de usuario (MU).
- Documentos de sistema, incluyendo el SDD y el ODD.
- El código fuente del sistema, incluyendo sus subsistemas (mostrado en la figura 10-7) y diversos programas de instalación.

Además, identificamos las interfaces entre las entidades de subsistemas cuya evolución debe estar controlada en forma cuidadosa, ya que los cambios que se les hagan pueden introducir grandes problemas. Identificamos a dos componentes del SDD, la *Especificación de protocolo cliente servidor* y la *Descripción del esquema de datos*, como artículos de configuración.

La figura 10-8 muestra los artículos de configuración y los agregados AC que identificamos para MisPartesCarro. Debido a que los subsistemas de MisPartesCarro pueden lanzarse en forma independiente, para cada subsistema identificamos un agregado AC que incluye los artículos de configuración relevantes para el subsistema. Por ejemplo, si modificamos el subsistema `ClienteN` para que incluya la funcionalidad de plano de navegación, necesitamos revisar al RAD para especificar los nuevos casos de uso, cambiar al `ClienteN MU:AC` para explicar la manera en que se usa esta funcionalidad, modificar al `ClienteN ODD:AC` para incluir nuevas clases para el plano e implementar y probar los cambios al código fuente.[2] Luego se revisa el `ClienteN:Agregado AC` para que incluya las versiones más recientes de los artículos de configuración relacionados con el `ClienteN` antes de que se lance a los usuarios. Tome en cuenta que este cambio puede hacerse sin modificar el `ClienteE:Agregado AC` y, por tanto, los artículos

2. Por brevedad, en la figura 10-8 no se muestra el código fuente, ni las pruebas ni el manual de pruebas.

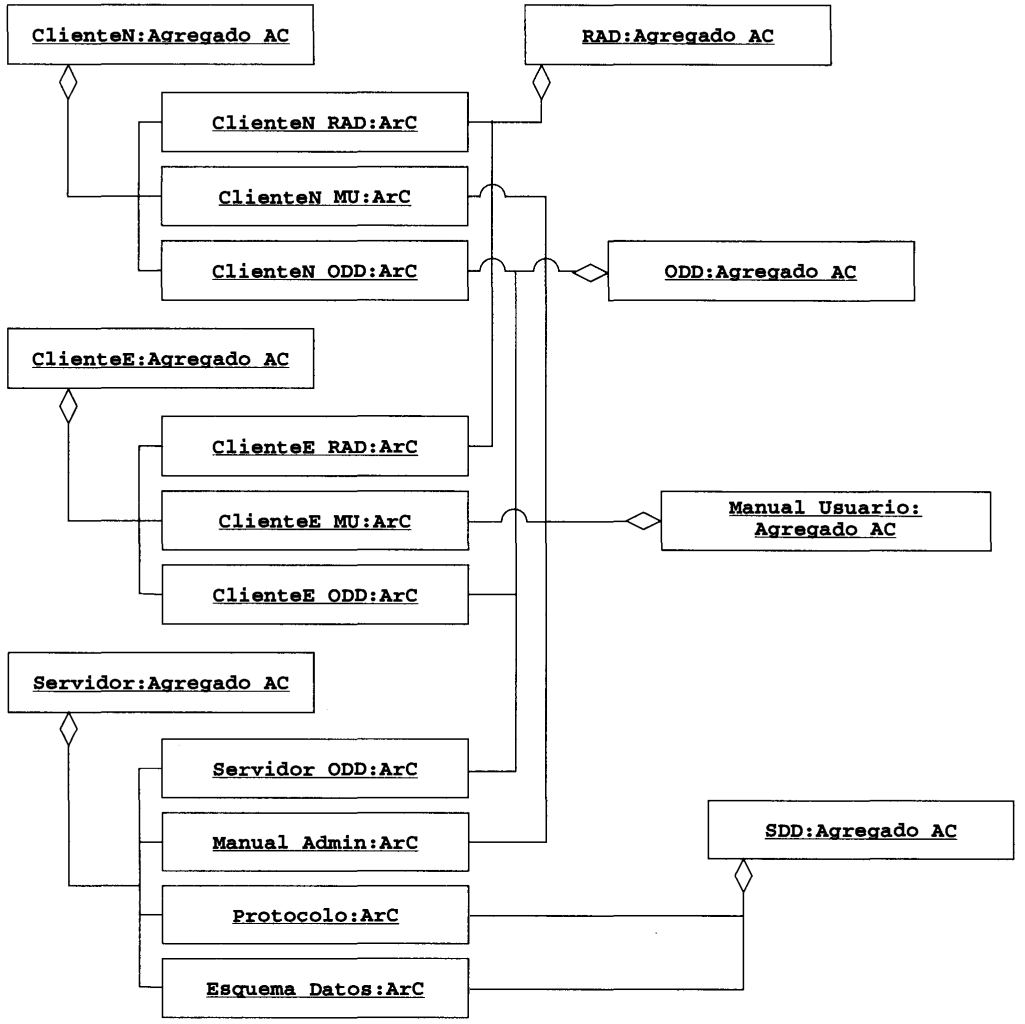


Figura 10-8 Artículos de configuración y agregados AC para MisPartesCarro (diagrama de objetos UML).

de configuración relacionados con los subsistemas de `ClienteE` y `ClienteN` se modelan como artículos de configuración y agregados AC separados.

Sin embargo, los tres subsistemas que constituyen al sistema MisPartesCarro no son independientes por completo. Antes de que podamos añadir planos de navegación al subsistema `ClienteN` necesitamos proporcionar primero funcionalidad para guardar y recuperar estas tablas en el subsistema `Servidor`. Aunque esta funcionalidad se realiza y lanza por separado, todavía necesitamos asegurarnos que los lanzamientos de clientes estén coordinados con los lanzamientos de servidor (en el sentido de que se necesita mejorar a los servidores antes que a los clientes).

Para este propósito también identificamos agregados AC en el nivel de sistema para cada uno de los productos a entregar. El RAD:Agregado AC, el SDD:Agregado AC y el ODD:Agregado AC representan versiones consistentes de los productos a entregar. Por ejemplo, las versiones del ODD:Agregado AC con el ODD ClienteN:ArC que describe las clases del plano de navegación también contiene al ODD Servidor:ArC que describe la manera de guardar y recuperar estos planos.

10.4.2 Administración de la promoción

La creación de nuevas promociones para un artículo de configuración sucede cuando un desarrollador quiere compartir el artículo de configuración con otras personas. Los desarrolladores crean promociones para poner a disposición el artículo de configuración para revisión, para la depuración de otro artículo de configuración o para una revisión de sanidad. Una vez que se crea y guarda la promoción en el sistema de administración de la configuración, los desarrolladores interesados en la promoción se registran en el depósito. Los desarrolladores que no están interesados en la promoción continúan trabajando con las versiones anteriores. Una vez que se crea una promoción ya no se puede modificar. El desarrollador que creó la promoción puede continuar modificando el artículo de configuración sin interferir con los desarrolladores que usan la promoción. Para distribuir nuevos cambios el desarrollador necesita crear una nueva promoción.

Por ejemplo, considere el cambio del plano de navegación en MisPartesCarro (figura 10-9). Primero los desarrolladores modifican el modelo de análisis para especificar los casos de uso del mapa de navegación y su interacción con los casos de uso existentes, dando como resultado la ClienteN RAD.2.0:Promoción. Luego se modifica el modelo de diseño del sistema para acomodar el almacenamiento y la descarga de los planos, dando como resultado las Protocolo.2.0:Promoción y Esquema Datos.2.0:Promoción. Luego, una primera implementación del protocolo del lado del servidor se realiza en la Servidor.2.0:Promoción, que se pone a disposición del equipo de cliente principiante para probar su implementación del plano de navegación (ClienteN. 2.0:Promoción y anteriores). Cuando prueban al ClienteN encuentran varios errores en el servidor. El equipo del servidor identifica y corrige estos errores, dando como resultado las Servidor.2.1:Promoción y Servidor.2.2:Promoción. Mientras tanto, el equipo de documentación revisa la ClienteN MU.2.0:Promoción basado en la ClienteN RAD.2.0:Promoción. Tome en cuenta que durante este tiempo el equipo de cliente experto puede estar reparando errores del ClienteE.1.5, en forma independiente del cambio del plano de navegación. Para probar el ClienteE, el equipo del cliente experto continúa usando el lanzamiento anterior del servidor (es decir, Servidor.1.4). La figura 10-9 ilustra este escenario anterior mostrando una instantánea del espacio de trabajo de cada equipo. Aunque los equipos estén trabajando hacia un sistema consistente, puede ser que todos estén trabajando con promociones diferentes de los mismos componentes hasta que se estabilizan todos los componentes.

Las promociones representan el estado de un artículo de configuración al momento en que se pone a disposición de otros desarrolladores. Un proyecto puede requerir, por lo general, que las promociones de código no contengan errores de compilación, pero muy pocas restricciones adicionales para motivar el intercambio de productos de trabajo entre equipos. Una vez que se mejora la calidad de los artículos de configuración y el equipo de control de calidad la valora, puede elegirse a la promoción para un lanzamiento.

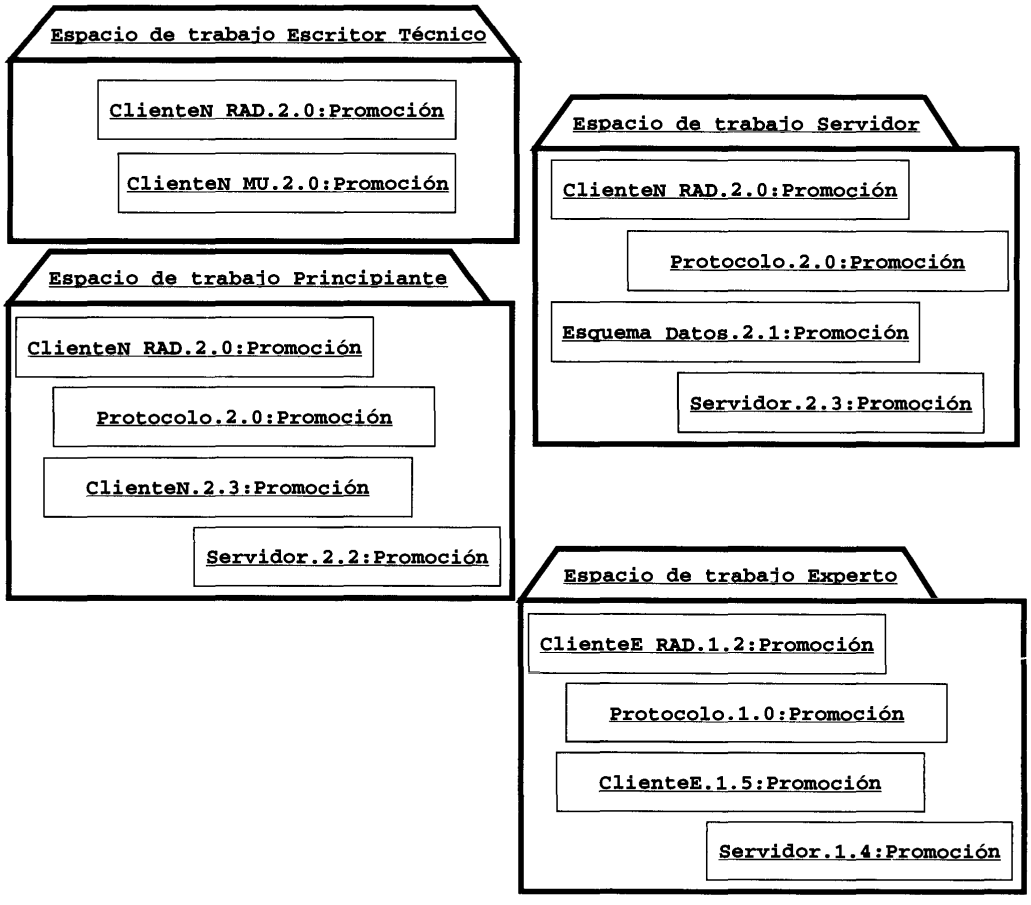


Figura 10-9 Instantánea de los espacios de trabajo usados por los desarrolladores de MisPartesCarro (diagrama de objetos UML). Los `espacios de trabajo Principiante`, `Escritor Técnico` y `Servidor` contienen promociones relacionadas con la funcionalidad del mapa de navegación. Sin embargo, el `espacio de trabajo Experto` contiene versiones antiguas y más estables. Para todos los artículos de configuración, los números de versión de la forma 1.x se refieren a las promociones sin la funcionalidad de mapas de navegación, mientras que los números de versión de la forma 2.x se refieren a las promociones que contienen implementaciones parciales o completas de los mapas de navegación.

10.4.3 Administración de los lanzamientos

La creación de un nuevo lanzamiento de un artículo de configuración o un agregado AC es una decisión gerencial basada, por lo general, en los consejos de mercadotecnia y control de calidad. Un lanzamiento se pone a disposición para proporcionar funcionalidad adicional (o revisada) o para resolver errores críticos.

Aunque la creación de un lanzamiento parece similar a la creación de una promoción, es un proceso más complicado y caro. Los desarrolladores manejan el cambio como parte de su trabajo. Si la nueva versión de un componente introduce más problemas de los que resuelve, tan sólo regresan a una promoción anterior. Un manual de usuario que no está actualizado no interfiere con su trabajo, ya que conocen el producto. Los usuarios, por otro lado, no están en el negocio de probar software. El descubrimiento de inconsistencias en el sistema y su documentación interfiere con su trabajo y puede dar lugar a que cambien de productos o contratistas. Por tanto, los lanzamientos se crean en un proceso mucho más controlado que las promociones, tratando de asegurar la consistencia y la calidad. El equipo de control de calidad valora la calidad de los componentes individuales de un lanzamiento y coordina las revisiones de los componentes defectuosos. Este proceso de auditoría permite que el equipo de control de calidad asegure la consistencia del lanzamiento con una interferencia mínima hacia el trabajo de los desarrolladores.

Por ejemplo, considere el lanzamiento de la funcionalidad del plano de navegación de MisPartesCarro (figura 10-10). Primero creamos una promoción estable de `ClienteN.2.4` y de `Servidor.2.3` que implementa la nueva funcionalidad. El equipo de control de calidad prueba estas dos promociones contra la versión más reciente del RAD (es decir, `ClienteN RAD.2.0`). El equipo de control de calidad encuentra un error que se corrige en una promoción subsiguiente del `ClienteN.2.5`. Luego el equipo de control de calidad prueba la versión actual de `ClienteE` (es decir, `ClienteE.1.5`) con la versión actual del servidor. Estando satisfechos con el estado del software, el equipo de control de calidad decide incluir `ClienteN.2.5`, `ClienteE.1.5` y `Servidor.2.3` en el siguiente lanzamiento mayor de MisPartesCarro. Luego el equipo de control de calidad revisa la consistencia del manual de usuario (`ClienteN MU.2.0`) frente a `ClienteN.2.5`. Se encuentran varios problemas adicionales y se resuelven en la promoción `ClienteN MU.2.1`, y luego se incluye en el lanzamiento que se está construyendo. Luego un conjunto seleccionado de usuarios hace pruebas beta con el lanzamiento. En las pruebas beta se descubren dos problemas adicionales en `ClienteN.2.5` y `Servidor.2.3`, y se corrigen en `ClienteN.2.6` y `Servidor.2.4`. Se vuelve a probar el software y se lanza a los usuarios como `MisPartesCarro.2.0`.

El administrador del sitio MisPartesCarro mejora primero el `servidor` y su esquema de base de datos, y observa, durante una prueba beta, si algún usuario descubre problemas de compatibilidad no previstos con los clientes antiguos. Después de la prueba beta el administrador pone el `ClienteN.2.6` a disposición de los usuarios que quieran usar la funcionalidad del plano de navegación. Una vez que se valida en el campo la estabilidad del nuevo lanzamiento, se motiva a todos los usuarios para que mejoren y se descontinúan las versiones antiguas del servidor y los clientes. Cualquier cambio a la documentación o a los subsistemas de MisPartesCarro se entregará como un parche o como un tercer lanzamiento.

El enfoque del proceso de lanzamientos en el ejemplo de MisPartesCarro es sobre la calidad y consistencia. El equipo de control de calidad actúa como un portero entre los usuarios y los desarrolladores. La suposición que hay tras este proceso es que los usuarios no están interesados en depurar el sistema que utilizan. Esperan que estos sistemas apoyen su trabajo. Esta suposición no es válida cuando los usuarios son desarrolladores de software. En este caso, este grupo de usuarios también desarrolla y mantiene una cantidad sustancial de las herramientas que usan. Pueden ser desde scripts simples que facilitan la realización de tareas repetitivas hasta lenguajes de programación completos, editores de sintaxis o sistemas de administración de configuración.

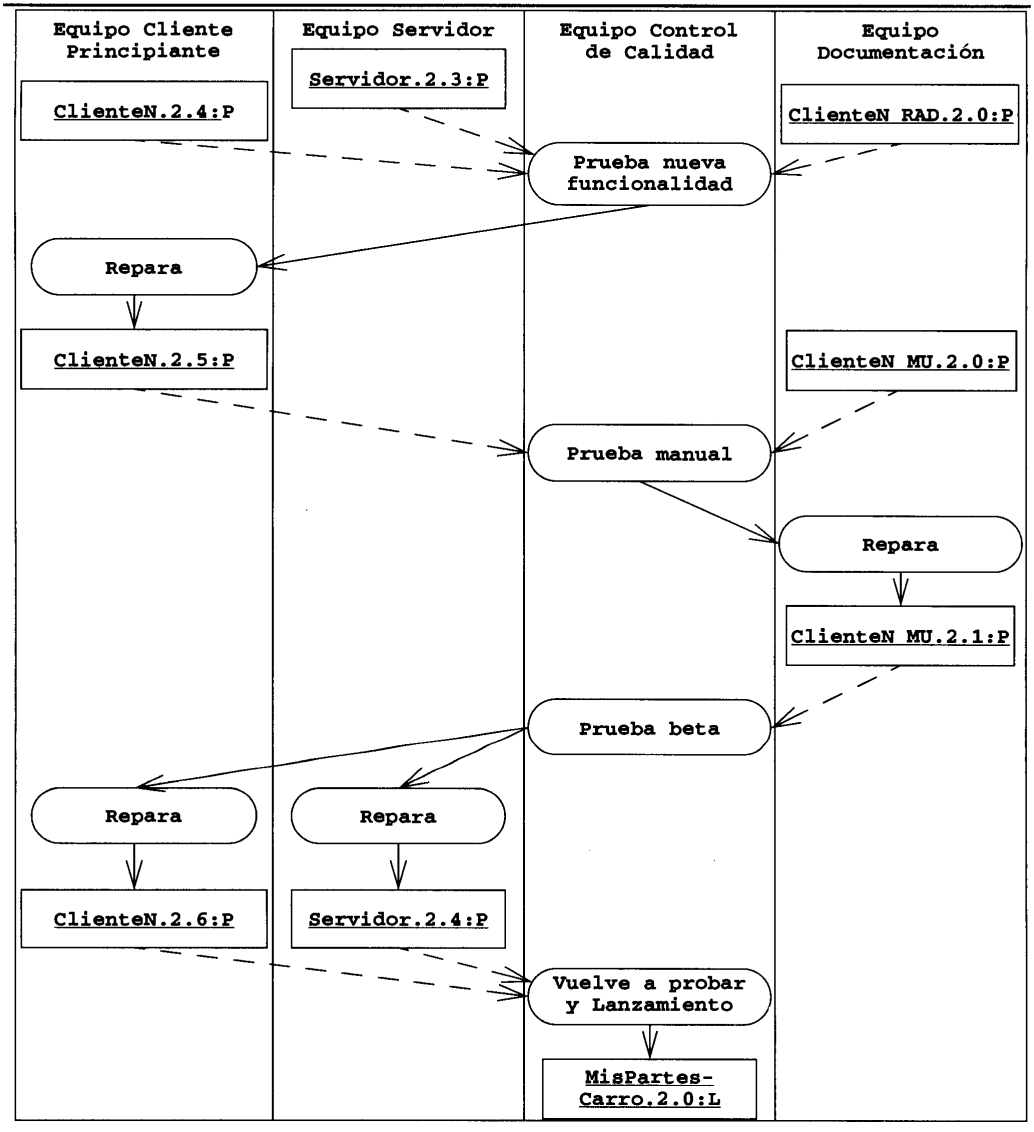


Figura 10-10 Proceso de lanzamiento para la funcionalidad de planos de navegación de `MisPartes-Carro.2.0` (diagrama de actividad UML). `:P` indica una promoción, `:L` indica un lanzamiento.

Con la popularidad de Internet entre los desarrolladores de software, se comparte y distribuye una cantidad cada vez mayor de estas herramientas *ad hoc*, en especial las que resuelven problemas comunes. Esto conduce a la disponibilidad de muchos programas gratuitos que van desde scripts simples hasta sistemas operativos (por ejemplo, Linux). En esta situación, los usuarios (que son desarrolladores de software) están deseosos de probar, depurar y enviar contribuciones

de código al autor del programa a cambio de un acceso temprano a la pieza de software. En este modelo de desarrollo de software, conocido como el modelo bazar [Raymond, 1998], los lanzamientos y las promociones llegan a ser mucho más similares que en el contexto del proceso controlado que describimos antes.

10.4.4 Administración de las ramas

Hasta ahora hemos examinado la administración de promociones y lanzamientos en el contexto de un solo cambio. Diversos artículos de configuración se revisan hasta que se termina el cambio, y luego lo valora el equipo de control de calidad. Durante este proceso hemos manejado la complejidad causada por la dificultad del mantenimiento de la consistencia entre promociones relacionadas y de la minimización de la introducción de nuevos defectos. Sin embargo, nos hemos enfocado tan sólo en un hilo de desarrollo.

Por lo general, los desarrolladores trabajan en varias mejoras en forma concurrente. Por ejemplo, mientras los desarrolladores implementan la funcionalidad del plano de navegación en el `ClienteN` y en el `Servidor`, puede ser que otros desarrolladores estén mejorando el tiempo de respuesta del `Servidor` y otros más estén extendiendo al `ClienteE` para guardar la historia de las consultas enviadas por los usuarios. Cuando los hilos del desarrollo se enfocan en subsistemas diferentes de MisPartesCarro (por ejemplo, planos de navegación en el `ClienteN` y en el `Servidor`, historias de consultas en `ClienteE`), puede aislarse a los equipos que están trabajando en configuraciones diferentes del mismo grupo de subsistemas (por ejemplo, el grupo del plano de navegación trabaja con las versiones más recientes de `ClienteN` y `Servidor`, mientras que el grupo de historias de consultas trabaja con las versiones más recientes de `ClienteE` y una versión estable más antigua de `Servidor`). Este enfoque funciona sólo si los cambios tienen un impacto en conjuntos de componentes que no se traslapan y si las interfaces de subsistemas siguen siendo compatibles hacia atrás. Sin embargo, cuando dos cambios requieren modificaciones del mismo componente se requiere un enfoque diferente. Las ramas concurrentes y su intercalación subsiguiente puede usarse para coordinar los cambios.

Por ejemplo, enfoquémonos en dos cambios concurrentes y traslapados: mientras los desarrolladores implementan la funcionalidad del plano de navegación (descrito en las secciones 10.4.2 y 10.4.3), asignamos a otro equipo la tarea de mejorar el tiempo de respuesta del `Servidor`. Ambos equipos pueden cambiar el `Servidor`, el `ClienteN` y sus interfaces para lograr sus objetivos respectivos.

Estos cambios se asignan a equipos diferentes debido a la diferencia de sus riesgos asociados. La mejora del plano de navegación extiende la funcionalidad de `MisPartesCarro` y está bien definida. Sin embargo, la mejora del tiempo de respuesta requiere trabajo experimental y está abierta: los desarrolladores necesitan identificar primero los cuellos de botella del desempeño y diseñar heurística para agilizar las peticiones comunes. Luego es necesario medir y valorar las mejoras resultantes frente a cualquier pérdida de confiabilidad y disminución de la mantenibilidad. Por último, la separación de ambos cambios nos proporciona mayor flexibilidad durante la entrega: si la mejora del tiempo de respuesta se termina pronto podemos intercalarla con la mejora funcional y lanzarla al mismo tiempo. En caso contrario, si la mejora del tiempo de respuesta se lleva más tiempo podemos integrarla más adelante como un parche.

Para soportar ambos cambios en forma concurrente, manteniendo al mismo tiempo independientes a los equipos, abrimos una rama que comienza en las últimas promociones de los subsistemas en el momento en que se aprueban los cambios (vea la figura 10-11).

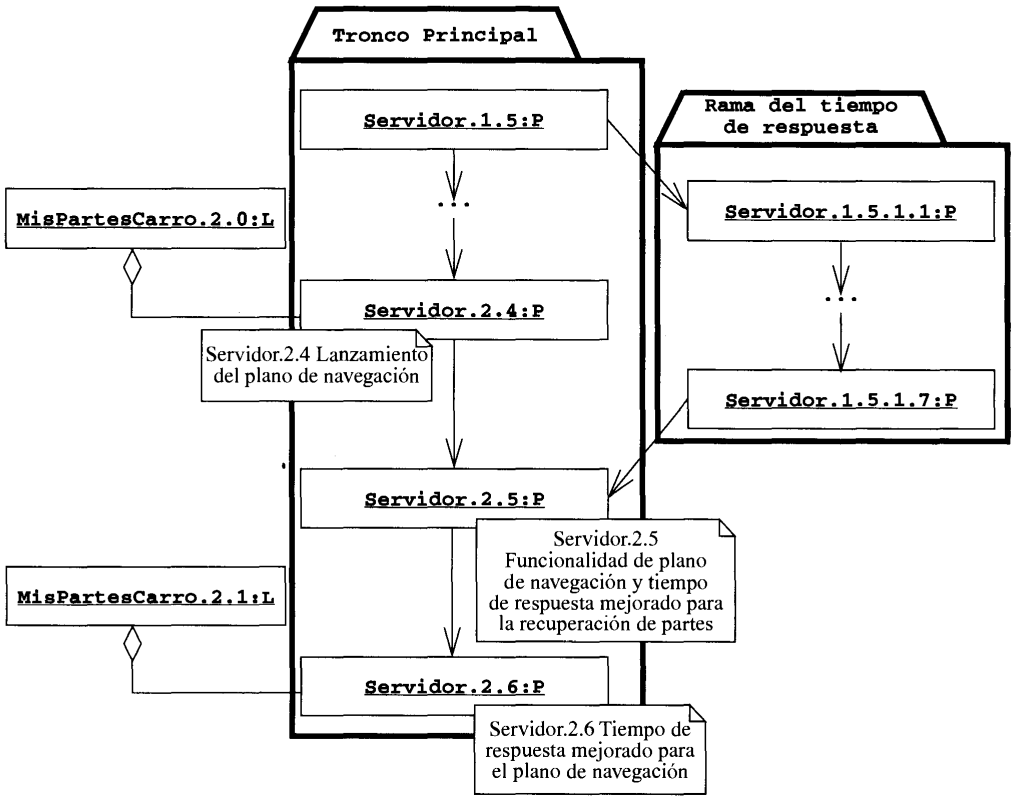


Figura 10-11 Un ejemplo de una rama (diagrama de objetos UML, se omitieron algunas promociones por brevedad, `:P` indica una promoción, `:L` indica un lanzamiento). En el tronco principal, los desarrolladores añaden la funcionalidad del plano de navegación a MisPartesCarro. En una rama concurrente los desarrolladores mejoran el tiempo de respuesta del servidor integrando un caché entre el servidor y la base de datos. La mejora del tiempo de respuesta se termina después del lanzamiento de la funcionalidad de plano de navegación y se pone a disposición como un parche.

Los equipos que trabajan en las mejoras funcionales continúan trabajando en el tronco principal (comenzando en `Servidor.1.5` y `ClienteN.1.6`). El equipo responsable de la mejora del tiempo de respuesta trabaja en la rama (que comienza en `Servidor.1.5.1.1`).[3] El equipo de

3. Usamos el esquema de identificación CVS para la identificación de versiones y ramas [Berliner, 1990].

mejoras del desempeño restringe sus cambios al subsistema `Servidor` y decide evitar la modificación de la interfaz de `Servidor`. Luego ambos equipos trabajan en forma independiente hasta que terminan sus mejoras. La mejora del plano de navegación se termina primero y se hace parte del segundo lanzamiento de MisPartesCarro, como vimos en la figura 10-10 (como `Servidor.2.4`). Poco después termina la mejora del tiempo de respuesta (`Servidor.1.5.1.7` en la figura 10-11). La mejora del tiempo de respuesta se considera satisfactoria, produciendo una disminución de cuatro veces en el tiempo de respuesta para las peticiones comunes mientras extiende en forma marginal el protocolo cliente servidor. En este punto necesitamos integrar estas mejoras con el tronco principal, esperando producir una versión de `MisPartesCarro` que tenga la funcionalidad del plano de navegación y la mejora de cuatro veces en el tiempo de respuesta.

Las intercalaciones pueden realizarse, por lo general, con la ayuda del sistema de administración de la configuración, el cual trata de intercalar las partes más recientes de las versiones a intercalar. Cuando se detectan conflictos, esto es, cuando ambas versiones contienen modificaciones a las mismas clases o métodos, la herramienta de administración de la configuración reporta el problema al desarrollador. Luego el desarrollador resuelve el conflicto en forma manual. En nuestro ejemplo, la herramienta reporta un conflicto en la clase `InterfazBD` que fue modificada por ambos equipos (figura 10-12).

El equipo de plano de navegación añadió un método, `procesaPetPlano()` para recuperar los planos de la base de datos. El equipo de mejora del tiempo de respuesta modificó el `procesaPetParte()` que recupera partes de la base de datos dando un ID de parte. Construimos una versión intercalada de la clase `InterfazBD` seleccionando `procesaPetPlano()` del tronco principal y `procesaPetParte()` de la rama. Luego probamos la clase `InterfazBD` revisada para asegurarnos de haber resuelto todos los conflictos, y esto da como resultado `Servidor.2.5:Promoción`. Después nos damos cuenta que el mismo mecanismo de caché puede usarse para `procesaPetPlano()`, y puede dar como resultado mejoras adicionales en el tiempo de respuesta. Modificamos el método `procesaPetPlano()`, volvemos a probar al servidor y creamos `Servidor.2.6:Promoción`.

Heurística para la administración de ramas

Como vimos en este ejemplo, la intercalación de dos ramas no es una operación trivial. Por lo general, necesita estar apoyada por una herramienta de administración de configuración y requiere una intervención considerable y pruebas de los desarrolladores. En el ejemplo de `MisPartesCarro`, que ilustra un caso muy simple, el equipo de mejora del tiempo de respuesta tuvo cuidado de no cambiar la interfaz del `Servidor` y limitar sus cambios al subsistema `Servidor`. Estas restricciones minimizaron la probabilidad de una superposición con los cambios hechos por el equipo del plano de navegación. Por lo general, si no se aplican restricciones, las ramas pueden divergir a tal punto que ya no puedan intercalarse. Aunque no hay formas confiables para resolver este problema, pueden usarse varias heurísticas para disminuir el riesgo de ramas divergentes.

- **Identificar probables superposiciones.** Una vez que se han especificado las ramas de desarrollo, pero antes de que comience el trabajo de diseño e implementación, los desarrolladores pueden anticipar dónde podrían ocurrir superposiciones. Luego se usa esta información para especificar decisiones sobre el contenido de estas superposiciones. Ejemplos de tales restricciones incluyen que no se modifiquen las interfaces de las clases que están involucradas en la superposición.

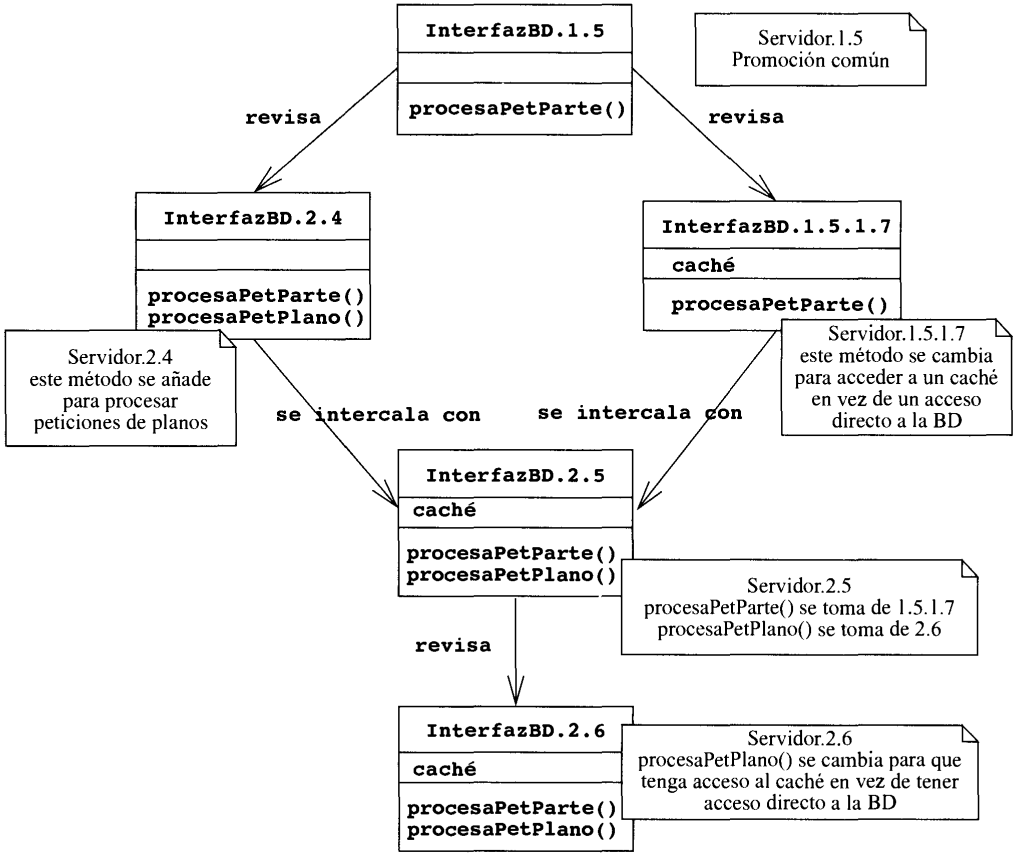


Figura 10-12 Un ejemplo de una intercalación de la clase `InterfazBD` del sistema `MisPartesCarro` (diagrama de clase UML).

- **Intercalar con frecuencia.** La política de administración de la configuración puede requerir que los desarrolladores trabajen en una rama para intercalarla a menudo con la versión más reciente del tronco principal (por ejemplo, diario, por semana o cada vez que se crea una nueva promoción). Las intercalaciones sólo se crean en la rama y no se propagan al tronco principal. La política también puede especificar que tales intercalaciones sólo necesiten asegurarse que el código todavía compila; esto es, las intercalaciones no tienen que resolver necesariamente todas las superposiciones. Esta política motiva a los desarrolladores para que encuentren pronto superposiciones y piensen en la manera de resolverlos antes de la intercalación real.
- **Comunicar la probabilidad de conflictos.** Aunque los equipos que trabajan en diferentes ramas necesitan trabajar en forma independiente, deben anticipar los conflictos durante la intercalación futura y comunicarlo a los equipos relevantes. Esto también tienen el beneficio de mejorar el diseño de ambos cambios tomando en cuenta las restricciones de ambos equipos.

- **Minimizar los cambios del tronco principal.** Al minimizar la cantidad de cambios a una de las ramas que se va a intercalar se reduce la probabilidad de conflictos. Aunque esta restricción no siempre es aceptable, es buena política de administración de la configuración restringir los cambios a la rama principal sólo para corregir errores y hacer todos los demás cambios en la rama de desarrollo.
- **Minimizar la cantidad de ramas.** Las ramas de administración de la configuración son mecanismos complejos de los que no hay que abusar. El trabajo de intercalación causado por una ramificación imprudente puede requerir más esfuerzo que si se hubiera usado una sola rama. Los cambios que pueden dar como resultado superposiciones y conflictos tienen, por lo general, dependencias mutuas y pueden tratarse en forma secuencial. Las ramas sólo deberán usarse cuando se requiere desarrollo concurrente y cuando pueden reconciliarse los conflictos.

En todos los casos, la creación de una rama es un evento significativo en el desarrollo. Deberá requerir aprobación de la gerencia y deberá estar planeada en forma meticulosa.

10.4.5 Administración de las variantes

Las variantes son versiones que se pretende que coexistan. Un sistema tiene diversas variantes cuando está soportado por diferentes sistemas operativos y diferentes plataformas de hardware. Un sistema también tiene muchas variantes cuando se entrega con diferentes niveles de funcionalidad (por ejemplo, versión para principiantes frente a versión para expertos, versión estándar frente a versión de lujo). Pueden tenerse dos enfoques fundamentales cuando se manejan variantes (figura 10-13).

- **Equipos redundantes.** Se asigna un equipo a cada variante. A cada equipo se le dan los mismos requerimientos y es responsable del diseño, implementación y pruebas completos de la variante. Una pequeña cantidad de artículos de configuración se comparten entre variantes, como el manual de usuario y el RAD.
- **Proyecto único.** Se diseña una descomposición en subsistemas que maximiza la cantidad de código compartido entre variantes. Para plataformas múltiples se confina el código específico de la variante en subsistemas de bajo nivel. Para varios niveles de funcionalidad se confinan los incrementos de funcionalidad en subsistemas individuales y principalmente independientes.

La opción de equipo redundante conduce a varios proyectos más pequeños que comparten la especificación de los requerimientos. La opción de proyecto único conduce a un solo proyecto más grande en donde la mayoría de los equipos comparten los subsistemas medulares. A primera vista, la opción de equipos redundantes conduce a redundancias en el proyecto, ya que la funcionalidad medular del sistema se diseñará e implementará varias veces. La opción de proyecto único parece más eficiente, tomando en cuenta que puede reutilizarse una cantidad de código potencialmente mayor entre las variantes, siempre y cuando se tenga un buen diseño del sistema. Es sorprendente que en desarrollos comerciales se seleccione con frecuencia la opción de equipo redundante [Kemerer, 1997] para evitar la complejidad organizacional.

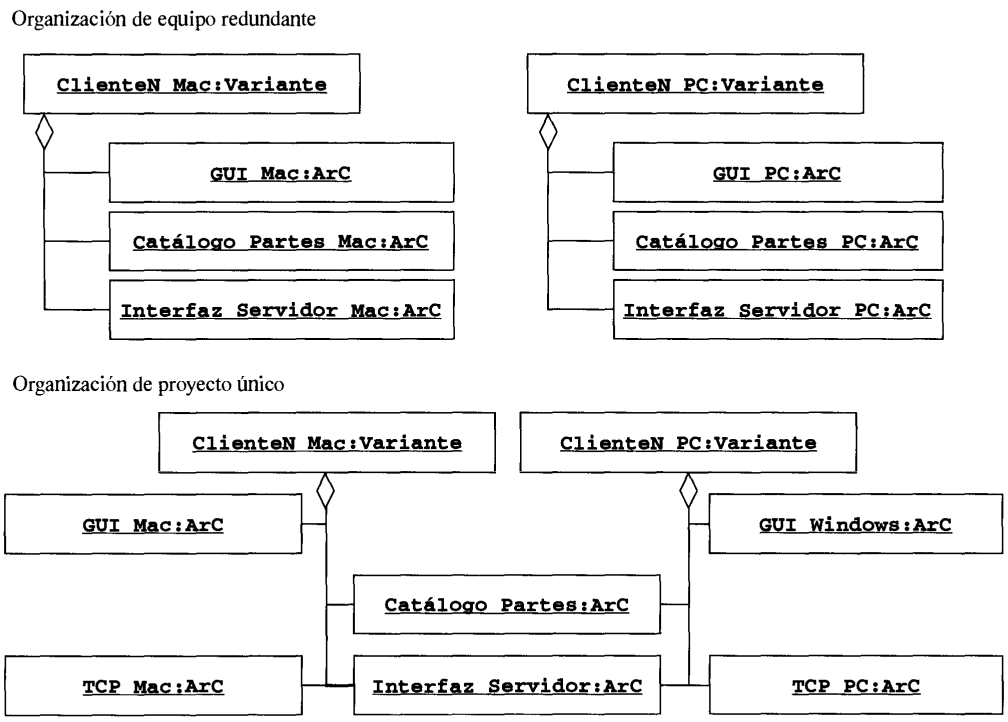


Figura 10-13 Ejemplos de variantes redundantes y variantes que comparten artículos de configuración (diagrama de objetos UML). En la organización de equipo redundante, los `ClienteN` de MisPartesCarro para Macintosh y PC se realizan en forma independiente. En la organización de proyecto único los `ClienteN` de MisPartesCarro para Macintosh y PC difieren en su interfaz de usuario.

Los problemas introducidos al compartir código incluyen:

- **Proveedor único/varios consumidores.** Los equipos que trabajan en variantes diferentes usan los subsistemas medulares y, por tanto, existe la posibilidad de requerimientos divergentes. Los equipos de los subsistemas medulares necesitan satisfacer los requerimientos de manera uniforme.
- **Ciclo largo de peticiones de cambio.** Cuando un equipo de una variante específica emite una solicitud de cambio de un subsistema medular, la aprobación e implementación del cambio puede tardar más que para otros cambios, tomando en cuenta que el equipo del subsistema medular necesita asegurarse que el cambio no interfiera con los demás equipos específicos de variantes.
- **Inconsistencias entre plataformas cruzadas.** Los subsistemas medulares introducen restricciones en subsistemas específicos de las variantes que pueden interferir con las restricciones de la plataforma. Por ejemplo, un subsistema medular puede estar diseñado con un flujo de control con hilos, mientras que el juego de herramientas de interfaz de usuario de una variante específica asume un flujo de control manejado por eventos.

Los equipos específicos de variante pueden percibir cada uno de estos problemas como una motivación para implementar sus propios subsistemas medulares. Sin embargo, estas cuestiones pueden tratarse anticipando los problemas específicos de variante durante el diseño del sistema y mediante una administración efectiva de la configuración. Un buen diseño de sistema es adaptable a los problemas de plataforma y específicos de la variante. Esto da como resultado una descomposición en subsistemas que es idéntica para todas las variantes, y en donde cada variante del sistema difiere por la sustitución de uno o más subsistemas. Por ejemplo, en la figura 10-13 las variantes Macintosh y PC de `ClienteN` comparten los subsistemas `Catálogo Partes` e `Interfaz Servidor`. Los subsistemas de interfaz de usuario e interfaz de red son diferentes, pero tienen la misma interfaz. Esto da como resultado una descomposición en subsistemas en donde cada subsistema es independiente de la variante (es decir, soporta a todas las variantes) o específico de variante (es decir, soporta a una pequeña cantidad de variantes). Los problemas que presentamos antes pueden tratarse entonces de la siguiente manera.

- El problema de *proveedor único/varios consumidores* se maneja con una administración del cambio cuidadosa: si un cambio solicitado es específico de una variante no debe tratarse en un subsistema medular. Si el cambio solicitado beneficia a todas las variantes deberá tratarse sólo en los subsistemas medulares.
- El *ciclo largo de peticiones de cambio* se acorta involucrando al equipo que envió la petición de cambio durante la variación. El cambio sugerido se implementa en una nueva promoción del subsistema medular y se lanza al equipo que solicitó el cambio. El equipo evalúa la solución y prueba su implementación, mientras que los demás equipos continúan usando la promoción anterior. Una vez que se valida el cambio, los demás equipos de variantes pueden comenzar a usar el subsistema revisado. Describimos un escenario de éstos con los cambios del lado del `Servidor` requeridos por el plano de navegación (vea la figura 10-9): el equipo del cliente principiante valida la nueva versión del servidor mientras que el equipo del `ClienteE` continúa trabajando con un lanzamiento anterior (y más estable). Tome en cuenta que cuando los equipos de variantes requieren cambios concurrentes al subsistema medular, los desarrolladores de la parte medular pueden usar ramas para aislar el impacto de cada cambio hasta que llegan a un estado estable.
- Las *inconsistencias de plataforma cruzada* se evitan durante el diseño del sistema, en la medida de lo posible, enfocándose en una descomposición en subsistemas independiente de variantes. Las inconsistencias de plataforma cruzada de nivel más bajo se tratan en subsistemas específicos de variante, con el costo de algunos objetos de pegamento o redundancia entre subsistemas medulares y específicos de variante. Si todo esto falla, deberán considerarse rutas de desarrollo independientes cuando las variantes soportadas son considerablemente diferentes.

La administración de diversas variantes con subsistemas compartidos es compleja, como lo dijimos antes. Sin embargo, después de una inversión anticipada durante el diseño del sistema, el enfoque de código compartido produce muchas ventajas, como mayor calidad y estabilidad del código compartido y una mayor consistencia de la calidad entre las variantes. Por último, cuando la cantidad de variantes es grande, la consideración temprana de los problemas específicos de las variantes, y el diseño de mecanismos de la administración de la configuración para satisfacerlos, conduce a ahorros considerables de tiempo y costo.

10.4.6 Administración del cambio

La creación de nuevas promociones y lanzamientos es impulsada por las peticiones de cambio. Un equipo repara un defecto y crea una promoción para que la evalúe el equipo de control de calidad. Los clientes definen nuevos requerimientos o los desarrolladores aprovechan una nueva tecnología de implementación causando un nuevo lanzamiento del sistema. Las peticiones de cambio pueden ir desde la corrección de ortografía en una etiqueta de menú hasta la reimplementación de un sistema principal para resolver asuntos de desempeño. Las peticiones de cambio también varían en su oportunidad: una petición de funcionalidad adicional durante la revisión de requerimientos puede conducir a la modificación del borrador del RAD, mientras que la misma petición hecha durante las pruebas puede conducir además a una cirugía mayor de subsistemas. Es necesario manejar en forma diferente las peticiones de cambio, dependiendo de su alcance y oportunidad. El manejo de las peticiones de cambio, llamado administración del cambio, es parte de la administración de la configuración.

Los procesos de administración del cambio varían en su formalidad y complejidad con los objetivos del proyecto. En el caso de un sistema complejo con requerimientos de confiabilidad elevados, por ejemplo, el formulario para la petición de cambios tiene varias páginas (por ejemplo, [MIL Std. 480]), requiere la aprobación de varios gerentes y se lleva varias semanas para procesarlo. En el caso de una herramienta simple escrita por un desarrollador, la comunicación informal es suficiente. En ambos casos, el proceso de cambio incluye los siguientes pasos (figura 10-14).

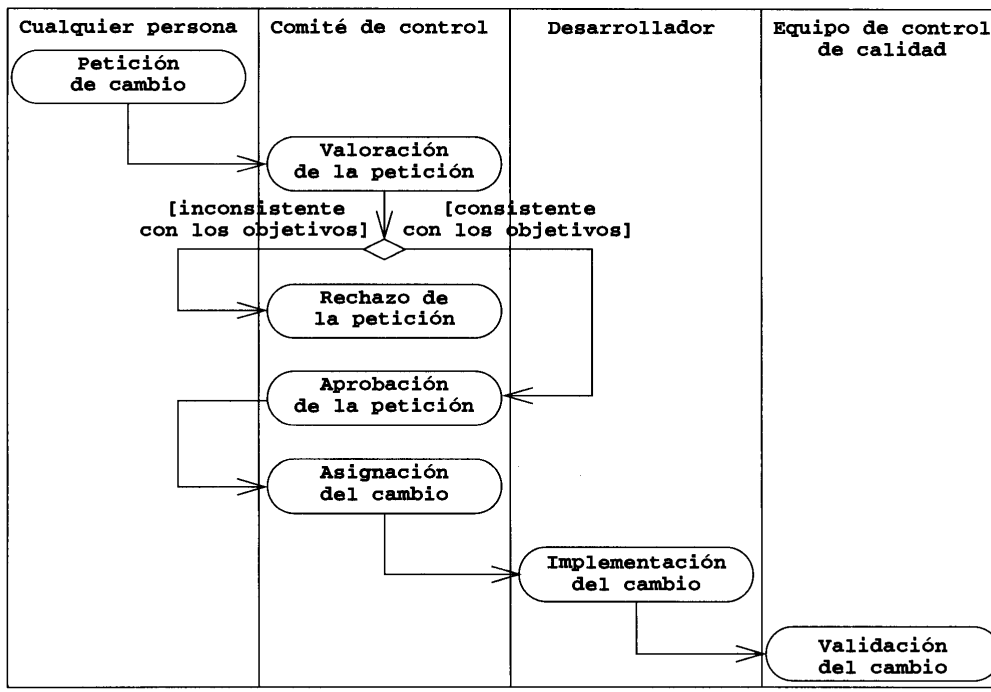


Figura 10-14 Un ejemplo del proceso de administración del cambio (diagrama de actividad UML).

1. Se solicita el cambio identificando un defecto o una nueva característica. Esto puede hacerlo cualquier persona, incluyendo un usuario o un desarrollador.
2. La petición se valora contra los objetivos del proyecto. En proyectos grandes esto lo realiza el comité de control. En proyectos más pequeños lo realiza el gerente del proyecto. Esto puede incluir un análisis de costos y beneficios y una evaluación del impacto del cambio en el resto del sistema.
3. Después de la valoración, la petición se acepta o se rechaza.
4. Si se acepta, el cambio se planea, se establece su prioridad, se asigna a un desarrollador y se implementa.
5. Se audita el cambio implementado. Esto lo realiza el equipo de control de calidad o quien sea responsable de la administración de lanzamientos.

10.5 Gestión de la administración de la configuración

En esta sección tratamos los problemas administrativos relacionados con la administración de la configuración, los cuales incluyen:

- La documentación de la administración de la configuración.
- La asignación de responsabilidades en la administración de la configuración.
- La planeación de las actividades de la administración de la configuración.

10.5.1 Documentación de la administración de la configuración

El estándar [IEEE Std. 828-1990] es una norma para la redacción de *Planes de administración de configuración de software* (SCMP, por sus siglas en inglés). Dicho plan documenta toda la información relevante de las actividades de la administración de configuración de un proyecto específico. El plan se genera durante la fase de planeación, se pone bajo la administración de la configuración y se revisa cuando es necesario. El alcance y la longitud del plan pueden variar de acuerdo con las necesidades del proyecto. Un proyecto de misión crítica con un proceso de control de cambios formal puede requerir un documento de 30 páginas. Un proyecto que desarrolla un prototipo conceptual puede requerir sólo un documento de cinco páginas.

Un SCMP contiene seis tipos de información: introducción, administración, actividades, calendarización, recursos y plan de mantenimiento (vea la figura 10-15).

La *introducción* describe el alcance y audiencia del documento, términos principales y referencias.

La sección de *administración* describe la organización del proyecto (que puede ser una referencia a la sección correspondiente en el *Plan de administración de proyectos de software*, vea el capítulo 11, *Administración del proyecto*) y la manera en que se asignan las responsabilidades de administración dentro de esta organización.

La sección de *actividades* describe con detalle la identificación de los artículos de configuración, el proceso de control de cambios, el proceso para la creación de lanzamientos y auditorías y el proceso para la contabilización de estado. Las responsabilidades de cada una de estas actividades se asignan a un papel definido en la sección de administración.

Plan de administración de la configuración del software

1. Introducción
   1.1 Propósito
   1.2 Alcance
   1.3 Términos principales
   1.4 Referencias
2. Administración
   2.1 Organización
   2.2 Responsabilidades
3. Actividades
4. Calendarización
5. Recursos
6. Plan de mantenimiento

Figura 10-15 Un ejemplo de una plantilla para el *plan de administración de la configuración del software.*

La sección de *calendarización* describe cuándo se realizan las actividades de administración de la configuración y la manera en que se coordinan. En particular, define en qué punto pueden solicitarse y aprobarse cambios a lo largo del proceso de cambio formal.

La sección de *recursos* identifica las herramientas, técnicas, equipo, personal y entrenamiento necesarios para la realización de las actividades de configuración del software.

Por último, la sección del *plan de mantenimiento* define la manera en que el SCMP mismo se mantiene y revisa bajo la administración de la configuración. En particular, esta sección detalla a la persona responsable de su mantenimiento, la frecuencia de las actualizaciones y el proceso de cambio para la actualización del plan.

El estándar [IEEE Std. 828-1990] está definido de tal forma que el esquema anterior puede aplicarse a cualquier proyecto, desde el diseño de sistemas confiables hasta un producto freeware.

10.5.2 Asignación de las responsabilidades de la administración de la configuración

La administración de la configuración es una función del proyecto que involucra muchas tareas y participantes diferentes en el proyecto. Así como sucede en el diseño del sistema, las tareas que requieren consistencia deberá realizarlas un pequeño número de personas. Durante el diseño del sistema un equipo pequeño de arquitectos toma decisiones acerca de la descomposición en subsistemas. Del mismo modo, un equipo pequeño de gerentes de configuración identifica los artículos de configuración y los agregados AC que necesitan controlarse. Al igual que en las pruebas, las tareas de control de calidad, como la administración de lanzamientos, deben realizarlas participantes diferentes a los de las tareas de desarrollo, como el diseño de objetos y la implementación.

La administración de la configuración incluye los siguientes papeles:

- **Gerente de configuración.** Éste es el papel responsable de la identificación de los artículos de configuración y agregados AC. Este papel también puede ser responsable de la definición de procedimientos para la creación de promociones y lanzamientos. Este papel se combina a menudo con el de arquitecto.

- **Miembro del comité de control del cambio.** Este papel es responsable de aprobar o rechazar las peticiones de cambio con base en los objetivos del proyecto. Dependiendo de la complejidad del proceso de cambio, este papel también puede estar involucrado en la valoración del cambio y en la planeación de los cambios aceptados. Este papel se combina con frecuencia con el de gerente de equipo o proyecto.
- **Desarrollador.** Este papel crea las promociones activadas por las peticiones de cambio o por las actividades normales del desarrollo. La actividad principal de la administración de la configuración de los desarrolladores es dar entrada a los cambios y resolver los conflictos de intercalación.
- **Auditor.** Este papel es responsable de la selección y evaluación de promociones para su lanzamiento. El auditor también es responsable de asegurar la consistencia y suficiencia del lanzamiento. Este papel lo realiza a menudo el equipo de control de calidad.

Los papeles para la administración de la configuración se definen durante la planeación del proyecto y se asignan desde el principio para asegurar la consistencia. La reasignación demasiado frecuente de los papeles de la administración de la configuración puede limitar en forma severa los beneficios de la administración de la configuración y fallar en el control del cambio.

10.5.3 Planeación de las actividades de la administración de la configuración

Los gerentes de proyecto deben planear las actividades de la administración de la configuración durante la fase de inicio del proyecto. La mayoría de los procedimientos para la administración de la configuración pueden definirse antes de que comience el proyecto, ya que no dependen del sistema mismo sino de los objetivos del proyecto y requerimientos no funcionales, como la seguridad y la confiabilidad.

Los elementos principales en la planeación de la administración de la configuración son:

- Definición de los procesos de la administración de la configuración.
- Definición y asignación de papeles.
- Definición del criterio de cambio; esto es, cuáles atributos de los productos de trabajo pueden cambiarse y qué tan tarde en el proceso.
- Definición de los criterios de lanzamiento; es decir, cuáles criterios deben evaluarse cuando se auditan las promociones para un lanzamiento.

Además, los procedimientos y herramientas para la administración de la configuración deberán estar en efecto antes de que comiencen a suceder los cambios para que puedan registrarse, aprobarse y darles seguimiento.

10.6 Ejercicios

1. RCS adopta un enfoque delta inverso para el almacenamiento de varias versiones de un archivo. Por ejemplo, supongamos que un archivo tiene tres revisiones, 1.1, 1.2 y 1.3. RCS almacena el archivo como la versión 1.3, luego las diferencias entre 1.2 y 1.3 y las diferencias

entre 1.1 y 1.2. Cuando se crea una nueva versión, digamos la 1.4, se calculan y guardan las diferencias entre 1.3 y 1.4, y la versión 1.3 se borra y se reemplaza por la 1.4. Explique por qué RCS no guarda tan sólo la versión inicial (en este caso la 1.1) y las diferencias entre cada versión sucesiva.

2. CVS usa una regla simple basada en texto para identificar las superposiciones durante una intercalación: hay una superposición si la misma línea se cambió en ambas versiones que se están intercalando. Si no existe tal línea, CVS decide que no hay conflicto y las versiones se intercalan de manera automática. Por ejemplo, supongamos que un archivo contiene una clase con tres métodos, `a()`, `b()` y `c()`. Dos desarrolladores trabajan en forma independiente en el archivo. Si ambos modifican las mismas líneas de código, digamos la primera línea del método `a()`, CVS decide que hay un conflicto.

   Explique por qué este enfoque no detecta determinados tipos de conflictos. Proporcione un ejemplo en su respuesta.
3. Los sistemas de administración de la configuración, como RCS, CVS y Perforce, usan nombres de archivo y sus rutas para identificar a los artículos de configuración. Explique por qué esta característica impide la administración de la configuración de agregados AC, incluso en presencia de etiquetas.
4. Explique la manera en que puede ser benéfica la administración de la configuración para los desarrolladores, aun en ausencia de un control de cambios o un proceso de auditoría. Liste dos escenarios que ilustren su explicación.
5. En el capítulo 8, *Administración de la fundamentación*, describimos la manera en que la información de la fundamentación puede representarse usando un modelo de problemas. Trace un diagrama de clase UML para un sistema de seguimiento de problemas que use un modelo de problemas para la descripción y discusión de cambios y su relación con las versiones. Enfóquese únicamente en los objetos de dominio del sistema.
6. En el capítulo 9, *Pruebas*, describimos la manera en que el equipo de control de calidad encuentra defectos en las promociones creadas por los equipos de subsistemas. Trace un diagrama de actividad UML que incluya las actividades del proceso de cambio y las actividades de las pruebas en un proyecto de varios equipos.

Referencias

[Babich, 1986] W. A. Babich, *Software Configuration Management*, Addison-Wesley, Reading, MA, 1986.

[Berliner, 1990] B. Berliner, "CVS II: parallelizing software development", *Proceedings of the 1990 USENIX Conference*, Washington, DC, enero de 1990, págs. 22–26.

[Bersoff *et al.*, 1980] E. H. Bersoff, V. D. Henderson y S. G. Siegel, *Software Configuration Management: An Investment in Product Integrity*, Prentice Hall, Englewood Cliffs, 1980.

[Conradi *et al.*, 1998] R. Conradi y B. Westfechtel, "Version models for software configuration management", *ACM Computing Surveys*, vol. 30, núm. 2, junio de 1998.

[Dart, 1991] S. Dart, "Concepts in configuration management systems", *Third International Software Configuration Management Workshop*, junio de 1991.

[IEEE Std. 828-1990] *IEEE Standard for Software Configuration Management Plans*, IEEE Standards Board, septiembre de 1990, en [IEEE 1997].

[IEEE Std. 1042-1987] *IEEE Guide to Software Configuration Management*, IEEE Standards Board, septiembre de 1987 (confirmada en diciembre de 1993), en [IEEE 1997].

[IEEE, 1997] *IEEE Standards Collection Software Engineering*, Piscataway, NJ, 1997.

[Kemerer, 1997] C. F. Kemerer, "Case 7: Microsoft Corporation: Office Business Unit", *Software Project Management: Readings and Cases*, Irwin/McGraw-Hill, Boston, MA, 1997.

[Knuth, 1986] D. E. Knuth, *The TeXbook*, Addison-Wesley, Reading, MA, 1986.

[Leblang, 1994] D. Leblang, "The CM challenge: Configuration management that works", en W. F. Tichy (ed.), *Configuration Management*, vol. 2 de *Trends in Software*, Wiley, Nueva York, 1994.

[MIL Std. 480] *MIL Std. 480*, U. S. Department of Defense, Washington, DC.

[Perforce] Perforce, Inc., 2420 Santa Clara Ave., Alameda, CA.

[POSIX, 1990] *Portable Operating System Interface for Computing Environments*, en IEEE Std. 1003.1, 1990.

[Raymond, 1998] E. Raymond, "The cathedral and the bazaar", disponible en http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html, 1998.

[Steelman, 1978] *Requirements for high order computer programming languages: Steelman*, U. S. Department of Defense, Washington, DC, 1978.

[Tichy, 1985] W. Tichy, "RCS - a system for version control", *Software Practice and Experience*, vol. 15, núm. 7, 1985.

11

11

Administración del proyecto

Quítate tu sombrero de ingeniero y ponte el de administrador.
— Declaración hecha durante la discusión del lanzamiento del 51-L

Los gerentes no generan un producto útil propio. En vez de ello, proporcionan y coordinan recursos para que los demás puedan generar productos útiles. La administración de un proyecto de software requiere una combinación de habilidades administrativas y sociales para predecir los problemas potencialmente dañinos y para implementar la respuesta adecuada. Debido a la falta de productos visibles y a la dependencia de habilidades que no son técnicas, la administración es, a menudo, motivo de bromas de los desarrolladores y otras personas. Sin embargo, la administración es una función crítica para llevar el proyecto a un final satisfactorio, tomando en cuenta la complejidad de los sistemas actuales y la alta tasa de cambio durante el desarrollo.

Los gerentes no toman decisiones técnicas y, muchas veces, no tienen los conocimientos para tomarlas. En vez de ello, son responsables de coordinar y administrar el proyecto y de asegurar que el sistema de alta calidad se entregue a tiempo y dentro del presupuesto. Las herramientas principales de la administración son la planeación, la supervisión, la administración del riesgo y el manejo de las contingencias.

En este capítulo describimos las actividades de administración que son visibles ante los desarrolladores y el líder del equipo. Suponemos una jerarquía de administración de dos niveles, que es típica de muchas industrias de software actuales. Nuestro objetivo no es proporcionar al lector la pericia de la administración de proyectos, sino darle suficientes conocimientos para que pueda actuar como líder de equipo e interactúe con la administración del proyecto. En consecuencia, tratamos temas como los planes de tareas, la asignación de papeles, la administración del riesgo y los planes de administración del proyecto de software. Dejamos atrás temas como la estimación de costos, los asuntos laborales y la negociación de contratos.

11.1 Introducción: la decisión del lanzamiento del STS-51L

Considere el siguiente ejemplo:

Ejemplo:[a] la decisión del lanzamiento del STS-51L

El 28 de enero de 1986 explotó, a los 73 segundos de vuelo, la misión STS-51L del transbordador espacial Challenger, matando a los siete miembros de la tripulación que iban a bordo. La comisión presidencial Rogers, formada para investigar el accidente, determinó que la fuga de gases de combustión a través de una junta en el cohete sólido secundario derecho causó la ignición del combustible de hidrógeno del tanque de combustible externo. La comisión Rogers también determinó que había serias fallas en el proceso de toma de decisiones de la NASA.

Los componentes principales del transbordador espacial son el orbitador, los cohetes sólidos secundarios y el tanque externo. El orbitador es el vehículo que transporta la carga y a los astronautas. Los cohetes sólidos secundarios proporcionan la mayor parte del empuje necesario para poner en órbita al orbitador. Cuando consumen todo su combustible se desprenden y caen al océano. El tanque externo, que proporciona combustible al motor del orbitador, también se desprende antes de alcanzar la órbita. Los cohetes sólidos secundarios fueron construidos por un contratista privado, Morton Thiokol. Los cohetes sólidos secundarios se construyeron en secciones para que pudieran transportarse con facilidad. Una vez entregados en el sitio de lanzamiento se ensamblan las secciones. Cada sección está conectada con una unión. Una falla de diseño de esta unión fue responsable de la fuga de gases de combustión que causó el accidente.

La comisión Rogers determinó que los ingenieros de Thiokol, el contratista que construyó los cohetes sólidos secundarios, estaban conscientes de una posible falla en las uniones del cohete sólido secundario. Los cohetes sólidos secundarios recuperados de vuelos anteriores del transbordador indicaron que las juntas de hule, llamadas anillos O, en las uniones de campo habían sido erosionadas por los gases de combustión. Esta erosión parecía bastante severa en los vuelos que se realizaban a bajas temperaturas. La erosión preocupó a los ingenieros porque los anillos O no habían sido diseñados para estar en contacto con gases de combustión. Los gerentes trataron el problema de las uniones como un riesgo aceptable, tomando en cuenta que la erosión había ocurrido en misiones anteriores sin causar incidentes. Los ingenieros de Thiokol habían tratado de comunicar la severidad del problema en forma repetida e inútil a su administración. En los días anteriores al lanzamiento STS-51L, los ingenieros estaban alarmados por las bajas temperaturas del sitio de lanzamiento y trataron de convencer a los administradores de que hicieran recomendaciones en contra del lanzamiento. Thiokol al principio hizo recomendaciones en contra del lanzamiento y luego invirtió su posición bajo presiones de la NASA.

Tanto la NASA como Thiokol tienen estructura de reportes jerárquica, típica de las organizaciones de administración públicas. Además, el flujo de información por la organización es mediante la estructura de reportes. Los ingenieros reportan a su gerente de grupo, quien luego reporta al gerente del proyecto, y así sucesivamente. Cada nivel puede modificar, distorsionar o suprimir cualquier información antes de pasarla hacia arriba, basado en sus propios objetivos. La misión STS-51L había sido pospuesta en repetidas ocasiones y ya estaba atrasada varios meses. Las administraciones de la NASA y Thiokol querían cumplir con la fecha límite. Esta situación dio como resultado la falta de comunicación, la diseminación de información incorrecta y una evaluación inadecuada del riesgo, tanto en la NASA como en Thiokol.

La comisión Rogers determinó que la causa principal del accidente del Challenger fue una falla de diseño en la unión del cohete sólido secundario. Sin embargo, también determinó que el proceso de toma de decisiones y la estructura de comunicación de Thiokol y la NASA eran defectuosos y fallaron en la valoración de los riesgos de lanzar un vehículo defectuoso.

a. El accidente del Challenger se ha resumido aquí como ejemplo de una falla de administración de un proyecto. Las fallas mecánicas y administrativas que causaron el accidente del Challenger son mucho más complejas que lo que sugiere este resumen. Para una referencia completa de la investigación, consulte el reporte de la Comisión Rogers [Rogers *et al.*, 1986] y libros relacionados con el tema [Vaughan, 1996].

La administración de proyectos asegura la entrega de un sistema de calidad a tiempo y dentro del presupuesto. Los componentes principales de esta definición son calidad, tiempo y di-nero. Para asegurar la entrega dentro del presupuesto se requiere que un gerente estime y asigne los recursos requeridos por el sistema desde el punto de vista de participantes, entrenamiento y herramientas. Para asegurar la entrega a tiempo se requiere que un gerente planee los esfuerzos de trabajo y supervise el estado del proyecto desde el punto de vista de tareas y productos de trabajo. Por último, para asegurar el control de la calidad se requiere que un gerente de proyecto proporcione mecanismos para el reporte de problemas y supervise el estado del producto en cuanto a defectos y riesgos. Los tres aspectos del proyecto, calidad, presupuesto y tiempo, son esenciales y necesitan abordarse juntos para que el proyecto tenga éxito.

En este capítulo describimos el papel de la administración en los proyectos de software. Sin embargo, el material que aquí se presenta está muy lejos de ser suficiente para aprender a actuar como gerente de proyecto, en el programa del transbordador o en cualquier otro lado. Este capítulo está orientado a desarrolladores que serán líderes de equipo después de que hayan obtenido alguna experiencia en proyectos. En este capítulo suponemos una jerarquía de administración de dos niveles, con el gerente del proyecto hasta arriba, el líder del equipo en medio y los desarrolladores en la parte inferior. Esto es bastante grande como para experimentar los efectos de la complejidad de la administración y bastante pequeño como para que sea típico de la industria de software actual.

En la sección 11.2 proporcionamos una vista general de la administración y su relación con otras actividades de desarrollo. En la sección 11.3 describimos los modelos usados para la administración, como las organizaciones, papeles, productos de trabajo, tareas y calendarizaciones. En la sección 11.4 describimos las actividades administrativas que manejan estos modelos e ilustramos la manera en que pueden usarse estos bloques de construcción usando un ejemplo de construcción de instalaciones. En la sección 11.5 describimos los problemas administrativos relacionados con la administración de proyectos.

11.2 Un panorama de la administración de proyectos

La administración de proyectos tiene que ver con la planeación y asignación de recursos para asegurar la entrega de un sistema de software a tiempo y dentro del presupuesto. La administración de proyectos está sujeta a las mismas barreras que las actividades técnicas: complejidad y cambio. Los productos complejos requieren gran cantidad de participantes con diversas habilidades y conocimientos. Los mercados competitivos y los requerimientos evolutivos introducen cambios en el desarrollo, desencadenan frecuentes reasignaciones de recursos y dificultan el seguimiento del estado del proyecto. Los gerentes manejan la complejidad y el cambio en la misma forma en que lo hacen los desarrolladores: mediante el modelado, la comunicación, la fundamentación y la administración de la configuración. Hemos hablado acerca de los aspectos de la comunicación, la fundamentación y la configuración de la administración en capítulos anteriores (capítulos 3, 8 y 10, respectivamente). En este capítulo nos enfocamos en los modelos de administración y las actividades necesarias para crearlos y mantenerlos.

Los modelos de administración nos permiten representar los recursos disponibles para el proyecto, las restricciones a las que están sujetos los recursos y las relaciones entre recursos. En este capítulo describimos los siguientes elementos:

- Los **equipos** representan conjuntos de participantes que trabajan en un problema común.
- Los **papeles** representan conjuntos de responsabilidades. Los papeles se usan para distribuir el trabajo a los participantes de un equipo.
- Los **productos de trabajo** representan los productos a entregar e intermedios del proyecto. Los productos de trabajo son los resultados visibles del trabajo.
- Las **tareas** representan el trabajo en función de los pasos secuenciales necesarios para generar uno o más productos de trabajo.
- Los **calendarios** representan la correspondencia entre un modelo de tareas y una línea de tiempo. Un calendario representa el trabajo desde el punto de vista de tiempo de calendario.

El gerente del proyecto y los líderes de equipo construyen y mantienen estos elementos del modelo a todo lo largo del desarrollo. Las actividades administrativas que corresponden a un líder de equipo se muestran en la figura 11-1.

Usando estos modelos, los líderes de equipo pueden comunicarse con los gerentes y desarrolladores acerca del estado y las responsabilidades. Si sucede una desviación considerable entre el trabajo planeado y el real, el gerente del proyecto reasigna recursos o cambia el ambiente del proyecto.

Al nivel de abstracción más alto, el ciclo de vida del desarrollo puede descomponerse en tres fases principales: **inicio del proyecto**, durante el cual se definen el alcance y recursos del proyecto, **estado estable**, durante el cual sucede la mayor parte del trabajo de desarrollo, y **terminación del proyecto**, durante la cual se entrega y acepta el sistema.

Inicio del proyecto. Durante esta fase el gerente del proyecto define el alcance del sistema junto con el cliente y construye la versión inicial de los modelos de administración. Al principio, sólo están involucrados el gerente del proyecto, algunos líderes de equipo seleccionados y el cliente. El gerente del proyecto especifica el ambiente del proyecto, organiza los equipos, contrata participantes y arranca el proyecto. El inicio del proyecto incluye las siguientes actividades:

- **Definición del enunciado del problema.** Durante esta actividad el cliente y el gerente del proyecto definen el alcance del sistema desde el punto de vista de la funcionalidad, restricciones y productos a entregar. El cliente y el gerente del proyecto se ponen de acuerdo sobre los criterios de aceptación y fechas de destino.
- **Diseño inicial de alto nivel.** El gerente del proyecto y el arquitecto del sistema descomponen el sistema en subsistemas para efectos de la asignación de equipos. El gerente del proyecto define los equipos y productos de trabajo en función de los subsistemas. El arquitecto del sistema y los desarrolladores revisarán la descomposición en subsistemas durante el diseño del sistema.
- **Formación de equipos.** El gerente del proyecto asigna un equipo a cada subsistema, define los equipos de funcionalidad cruzada y selecciona a los líderes de equipo. El gerente del proyecto y los líderes de equipo asignan juntos los papeles y responsabilidades a los participantes en función de las habilidades de éstos. Durante esta fase los líderes de equipo establecen las necesidades de entrenamiento para el equipo.
- **Establecimiento de la infraestructura de comunicación.** El gerente del proyecto y los líderes de equipo establecen la infraestructura de comunicación del proyecto, incluyendo tableros de noticias, sitios Web, herramientas para la administración de la configuración,

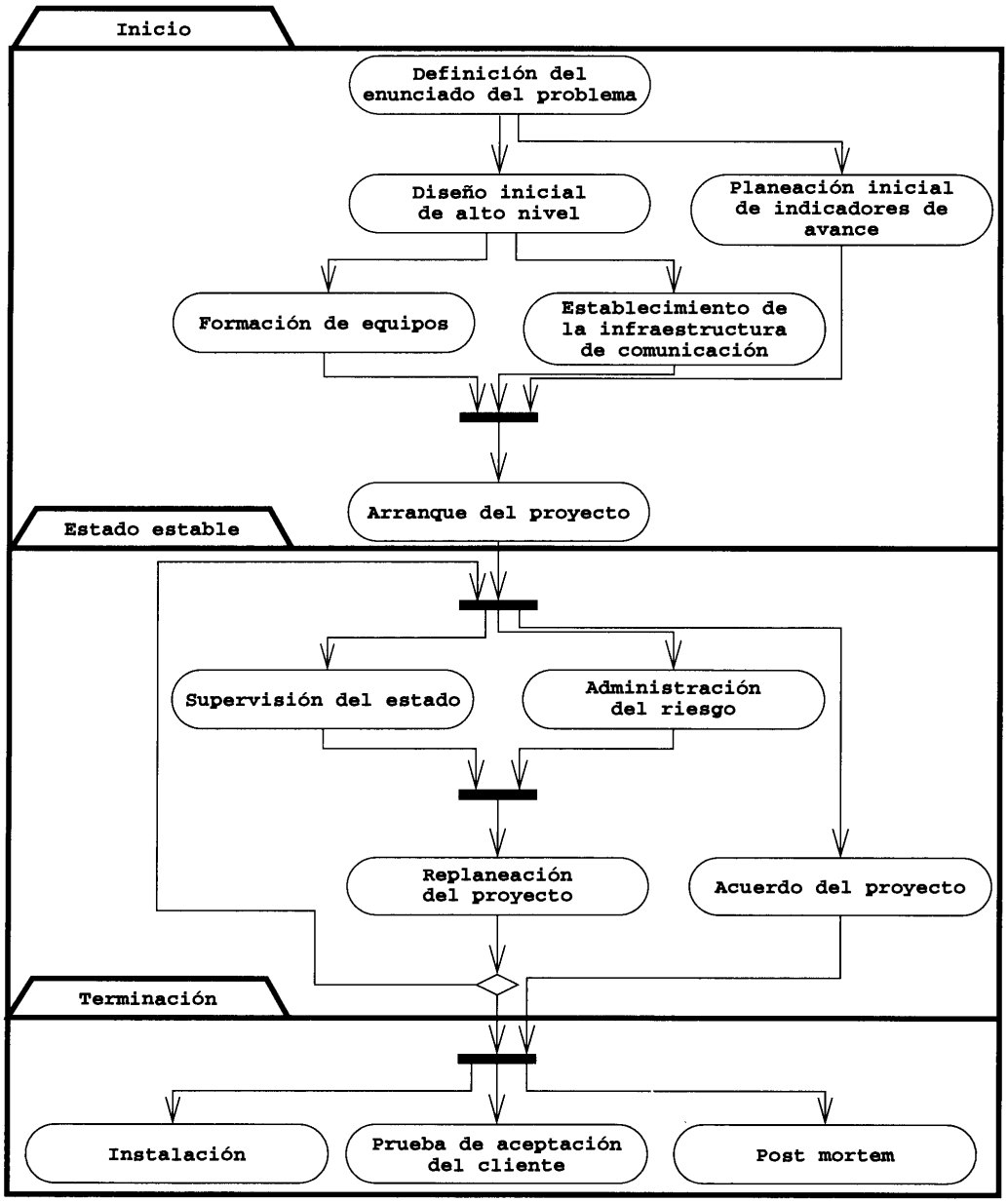


Figura 11-1 Actividades de la administración durante el inicio, el estado estable y la terminación (diagrama de actividad UML).

plantillas de documentos y procedimientos de reunión. Los participantes en el proyecto usarán esta infraestructura para la distribución de información y el reporte de problemas en forma ordenada.

- **Planeación inicial de indicadores de avance.** El gerente del proyecto calendariza los productos de trabajo intermedios y los productos a entregar en forma externa y los asigna a los equipos.
- **Arranque del proyecto.** El gerente del proyecto, los líderes de equipo y el cliente arrancan el proyecto en forma oficial. El propósito de las reuniones de arranque es explicar a los participantes el alcance del proyecto, la infraestructura de comunicación y las responsabilidades de cada equipo. Después del arranque el proyecto está en el estado estable.

Estado estable. Durante esta fase llega a ser crítico el papel del líder de equipo. Durante el inicio del proyecto la mayoría de las decisiones las toma el gerente del proyecto. Durante el estado estable los líderes de equipo son responsables del seguimiento del estado de su equipo y de la identificación de problemas mediante reuniones del equipo. Los líderes de equipo reportan el estado de su equipo al gerente del proyecto, el cual luego evalúa el estado de todo el proyecto. Los líderes de equipo responden a las desviaciones del plan reasignando tareas a los desarrolladores u obteniendo recursos adicionales del gerente del proyecto. El gerente del proyecto todavía es responsable de la interacción con el cliente, obteniendo el acuerdo formal y renegociando recursos y fechas límite. Las actividades del estado estable incluyen las siguientes:

- **Definición del acuerdo del proyecto.** Una vez que está estable el modelo de análisis, el cliente y el gerente del proyecto acuerdan de manera formal el alcance del sistema y la fecha de entrega. El documento *Acuerdo del proyecto* se deriva del enunciado del problema y se revisa durante el análisis.
- **Supervisión del estado.** A lo largo del proyecto los líderes de equipo y la administración supervisan el estado y lo comparan con la calendarización planeada. Los líderes de equipo son responsables de la recopilación de información de estado mediante reuniones, revisiones, reportes de problemas y terminación de productos de trabajo, y de resumirla para el gerente del proyecto.
- **Administración del riesgo.** Durante esta actividad los participantes en el proyecto identifican los problemas potenciales que pueden causar retrasos en la calendarización y sobregiros en el presupuesto. El gerente del proyecto y los líderes de equipo identifican, analizan y priorizan los riesgos y preparan planes de contingencia para los riesgos importantes.
- **Replaneación del proyecto.** Cuando el proyecto se desvía de la calendarización, o cuando se activa un plan de contingencia, el gerente del proyecto o el líder de equipo revisa el calendario y reasigna recursos para satisfacer el tiempo de entrega. El gerente del proyecto puede contratar nuevo personal, crear nuevos equipos o intercalar equipos existentes.

Terminación del proyecto. Durante esta fase se entrega el producto y se recopila la historia del proyecto. La mayor parte de la participación de los desarrolladores en el proyecto termina antes de esta fase. Unos cuantos desarrolladores principales, los escritores técnicos y los líderes de equipo se involucran con la envoltura del sistema para su instalación y aceptación, y recopilan la historia del proyecto para usarla en el futuro.

- **Prueba de aceptación del cliente.** El sistema se evalúa con el cliente de acuerdo con los criterios de aceptación establecidos en el *Acuerdo del proyecto*. Se muestran y prueban los requerimientos funcionales y no funcionales usando escenarios definidos en el *Acuerdo del proyecto*. En esta etapa el cliente acepta de manera formal el producto.
- **Instalación.** El sistema se entrega en el ambiente de destino y se entregan los documentos. La instalación puede incluir el entrenamiento de los usuarios y una fase de transición durante la cual se mueven los datos desde el sistema anterior hacia el nuevo. Por lo general, la aceptación precede a la instalación.
- **Post mortem.** El gerente del proyecto y los líderes de equipo recopilan la historia del proyecto para el aprendizaje de la organización. Al capturar la historia de las fallas mayores y menores y analizar sus causas, una organización puede evitar la repetición de errores y mejorar su práctica.

En todas las actividades anteriores, la comunicación oportuna y precisa entre los administradores del proyecto, los líderes de equipo y los desarrolladores es crucial para determinar el estado de un proyecto y para tomar las decisiones correctas. Los participantes en el proyecto disponen de varias herramientas que incluyen la infraestructura de comunicación, los documentos administrativos, como el *Plan de administración del proyecto de software* (SPMP, por sus siglas en inglés), y, en términos más generales, la motivación del libre intercambio de información. Un líder de equipo puede mejorar de manera considerable la comunicación recompensando los reportes precisos, en vez de castigar al portador de malas noticias, comunicando la mayor cantidad de información posible en forma expedita y respondiendo de manera constructiva a los reportes de problemas.

Los gerentes usan modelos para manejar la complejidad y los revisan para manejar el cambio. En la siguiente sección describimos los elementos del modelo que se usan para la administración y los representamos usando diagramas de clase UML. En la sección 11.4 describimos las actividades durante las que se crean y revisan estos modelos usando diagramas de objetos y diagramas de actividad.

11.3 Conceptos de administración

En esta sección describimos los conceptos principales que se usan durante la administración del proyecto. Cada elemento representa trabajo desde una perspectiva diferente (figura 11-2).

- Un **equipo** es un pequeño grupo de personas que trabajan sobre el mismo subproblema (sección 11.3.1). El concepto de equipo maneja la complejidad asociada con la comunicación entre muchas personas. En vez de organizar y coordinar a los participantes del proyecto como un grupo grande, el proyecto se divide en equipos donde cada uno ataca un subproblema separado.
- Un **papel** representa un conjunto de responsabilidades asignadas a un participante o a un equipo (sección 11.3.2). La definición clara de los papeles permite que se descubran y asignen con facilidad las tareas no planeadas. El papel de un probador, por ejemplo, es diseñar y ejecutar pruebas. Si se identifican nuevas actividades de pruebas no planeadas se asignan al participante que cumple el papel de probador.

- Un **producto de trabajo** representa un producto entregable concreto producido por un papel (sección 11.3.3). Los productos de trabajo incluyen documentos técnicos y administrativos, características de productos, casos de prueba, resultados de las pruebas, estudios de tecnología y reportes de usabilidad.
- Una **tarea** representa un producto de trabajo desde el punto de vista de una actividad (sección 11.3.4). Una tarea da como resultado uno o más productos de trabajo, está asignada a un papel y consume recursos. Las tareas están relacionadas entre sí mediante dependencias. Por ejemplo, *Planeación de pruebas de la base de datos* es la tarea de diseñar pruebas para el subsistema de base de datos. Requiere la interfaz hacia el subsistema, y da como resultado un plan de pruebas y un paquete de pruebas a ejecutar durante la *Tarea de prueba de la base de datos*.
- Una **calendarización** representa tareas asignadas en una línea de tiempo (sección 11.3.5). La calendarización es, por lo general, el más visible y evolutivo de todos los modelos de administración. La calendarización se revisa cuando se refinan las estimaciones, cuando cambian las restricciones o cuando se reasignan recursos.

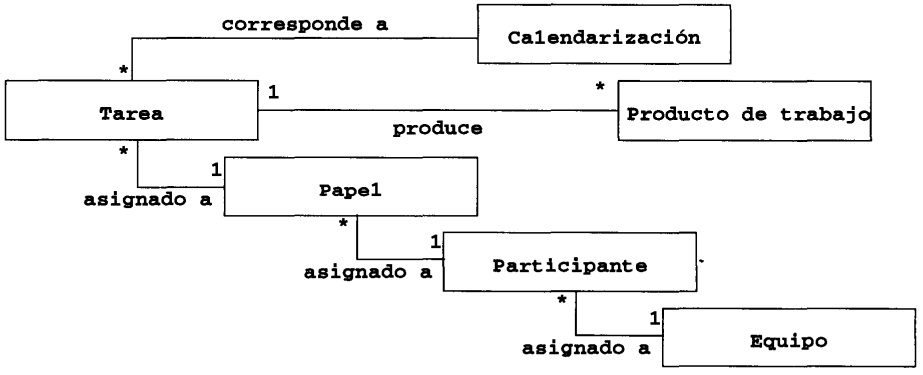


Figura 11-2 Relaciones entre participantes, equipos, papeles, tareas, productos de trabajo y calendarizaciones (diagrama de clase UML). Los `Participante` se asignan a `Equipo` que es responsable de un subsistema diferente. A cada `Participante` se le asignan varios `Papel`, donde cada uno corresponde a un conjunto de `Tarea`. La `Tarea` da como resultado uno o más `Producto de trabajo`. Una `Calendarización` consiste en un conjunto de tareas que corresponden a una línea de tiempo.

11.3.1 Equipos

Un **equipo** es un pequeño grupo de personas que trabajan sobre el mismo subproblema. En vez de organizar y coordinar a los participantes en el proyecto como un grupo grande de individuos, separando cada uno el mismo problema, el sistema se divide en subsistemas y cada uno se asigna a un equipo. Cada equipo es responsable del análisis, diseño e implementación de sus subsistemas. Cada equipo negocia con los demás equipos las interfaces de los subsistemas que necesita. A tales equipos se les llama **equipos de subsistema**. La figura 11-3 representa una arquitectura de sistema de tres subsistemas que corresponden a una organización de tres equipos de subsistemas.

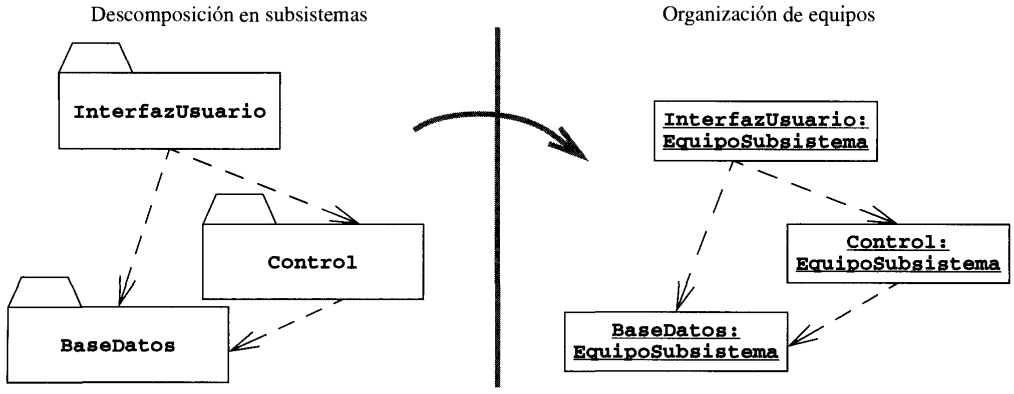


Figura 11-3 Un ejemplo de descomposición en subsistemas y su correspondencia con una organización en equipos (diagrama de objetos UML). Las flechas representan dependencias entre subsistemas y entre equipos.

A cada equipo lo administra un **líder de equipo**, quien asigna responsabilidades a sus miembros, facilita las reuniones del equipo y da seguimiento al estado. El líder del equipo es, por lo general, un desarrollador experimentado que está familiarizado con el contenido técnico del subsistema. El líder del equipo recopila y resume la información sobre el estado para el gerente del proyecto y reporta los asuntos pendientes.

La relación jerárquica entre los desarrolladores, los líderes de equipo y el gerente del proyecto representa la **estructura de reporte**. El desarrollador, que tiene el conocimiento más minucioso de los detalles técnicos, toma decisiones en forma local y las reporta al líder del equipo. El líder del equipo, que tiene un panorama del subsistema, puede hacer a un lado las decisiones de los desarrolladores cuando lo considera necesario y lo reporta al gerente del proyecto. El gerente del proyecto, que tiene una visión global de los objetivos del proyecto y su estado, puede, virtualmente, hacer a un lado cualquier decisión. Las decisiones y lineamientos del gerente del proyecto son implementadas por los líderes de equipo, asignando conceptos de acción a desarrolladores específicos. La figura 11-4 ilustra el flujo de información hacia arriba y abajo de la estructura de reporte de una organización de tres equipos de subsistema.

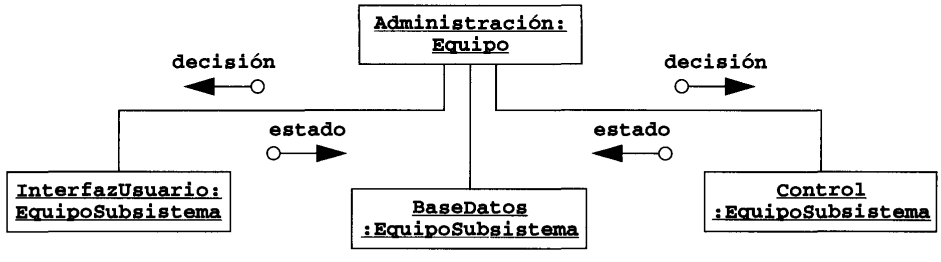


Figura 11-4 Ejemplo de una estructura de reporte (diagrama de colaboración UML). La información de estado se reporta al gerente del proyecto y las decisiones correctivas se comunican a los equipos mediante los líderes de equipo. A los líderes de equipo y al gerente del proyecto se les llama equipo de administración.

La estructura de reporte corresponde al flujo de la información de estado y la información para la decisión. Sin embargo, las relaciones de reporte representan sólo una pequeña parte del flujo de información del proyecto. Los líderes de equipo, por ejemplo, en general no discuten los detalles de la interfaz de subsistemas con el gerente del proyecto. Sin embargo, necesitan negociar estos detalles con los equipos que dependen de sus subsistemas. A veces, los líderes de equipo de equipos dependientes interactúan en forma directa. Sin embargo, lo más frecuente es que un desarrollador, llamado **enlace**, sea responsable de la tarea de comunicarse con otro equipo y transfiera información entre ellos. El equipo de documentación, por ejemplo, tiene un enlace con el equipo de interfaz de usuario para facilitar la información acerca de cambios recientes hechos a la apariencia del sistema. A los equipos que no trabajan de manera directa en un subsistema, sino que realizan una función en el nivel de proyecto, se les llama **equipos de funcionalidad cruzada**. Algunos ejemplos de estos equipos incluyen al equipo de documentación, al equipo de arquitectura y al equipo de pruebas.

En toda organización hay tres tipos de asociaciones entre equipos:

- La asociación de reporte, que se usa para reportar la información de estado.
- La asociación de decisión, que se usa para la propagación de decisiones.
- La asociación de comunicación, que se usa para el intercambio de otra información necesaria para las decisiones (por ejemplo, requerimientos, modelos de diseño, asuntos).

En organizaciones jerárquicas, como las militares o religiosas, la estructura de reporte también sirve para la toma de decisiones y la comunicación. Sin embargo, en proyectos de software complejos la mayoría de las decisiones técnicas las hacen los desarrolladores en forma local. Esto requiere que se intercambie y actualice mucha información entre los equipos. Esta información no puede intercambiarse mediante la estructura de reporte, dada su complejidad y volumen. En vez de ello, este intercambio de información lo manejan los equipos de funcionalidad cruzada y los enlaces. La figura 11-5 muestra un ejemplo de una organización de equipos con enlaces hacia equipos de funcionalidad cruzada.

La figura 11-6 es un ejemplo de una organización de equipos para un sistema con tres subsistemas. Los vínculos entre cada equipo y el equipo de administración (mostrados con líneas gruesas para mayor claridad) representan la estructura de reporte. Las demás asociaciones representan comunicación. Los vínculos entre los equipos de subsistemas representan la comunicación resultante de las dependencias entre subsistemas.

El papel de un líder de equipo es asegurarse que no sólo el gerente del proyecto esté consciente del estado del equipo, sino que también los miembros del equipo tengan toda la información que necesitan de los demás equipos. Esto requiere la selección de comunicadores efectivos como enlaces y la interacción con otros líderes de equipo para asegurar que existan todas las rutas de comunicación necesarias.

Otras organizaciones de equipos

En proyectos basados en equipos la organización cambia con el tiempo. Si se identifican nuevos subsistemas en la arquitectura de software se forman nuevos equipos de subsistemas. Si se requieren nuevas funciones cruzadas se seleccionan enlaces de los equipos de subsistemas y se ensamblan en nuevos equipos de funcionalidad cruzada. Del mismo modo, los equipos se

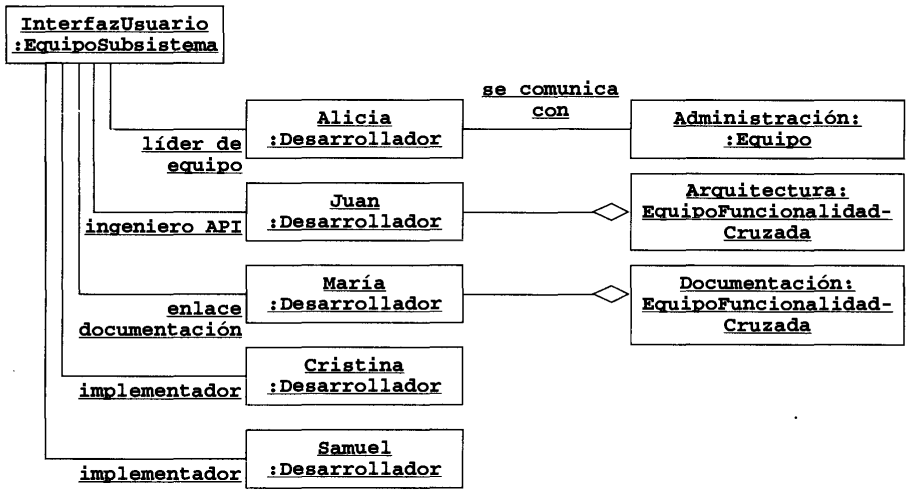


Figura 11-5 Ejemplos de una organización de equipos (diagrama de objetos UML). El equipo está compuesto por cinco desarrolladores. Alicia es la líder del equipo, y también se le llama enlace con el equipo de administración. Juan es el ingeniero de API, y también se le llama enlace con el equipo de arquitectura. María es el enlace con el equipo de documentación. Cristina y Samuel son implementadores e interactúan sólo de manera informal con otros equipos.

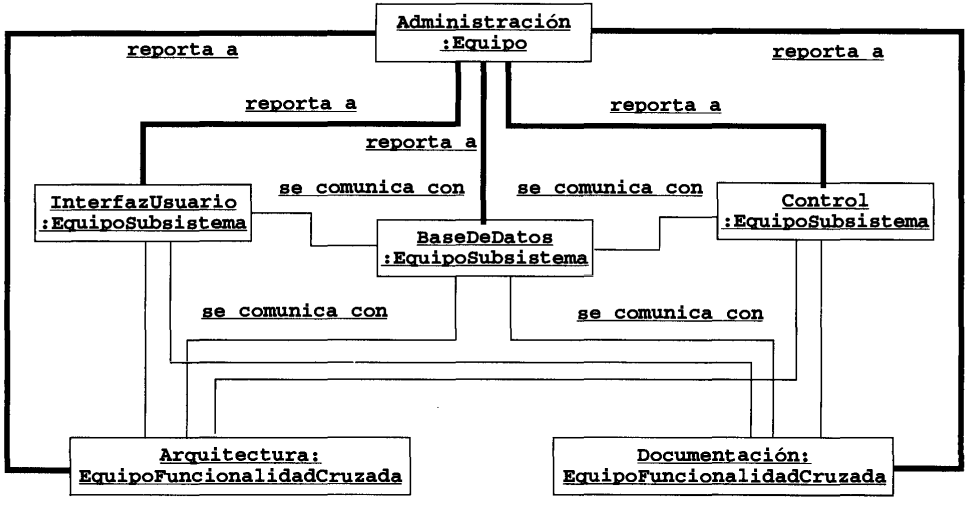


Figura 11-6 Un ejemplo de una organización basada en equipo con cuatro equipos de subsistemas y tres equipos de funcionalidad cruzada (diagrama de objetos UML). Los vínculos entre todos los equipos y el equipo de administración (trazado con líneas gruesas para mayor claridad) representan la estructura de reporte. Todas las demás asociaciones representan comunicación.

eliminan de la organización una vez que terminan su trabajo. El modelo de equipos debe tratarse en la misma forma que el modelo del sistema: se actualiza para que refleje los cambios de la realidad y evoluciona de acuerdo con las necesidades del sistema y del proyecto.

La ingeniería de software introduce un compromiso en la organización de equipos. Los desarrolladores tienen el mayor conocimiento técnico sobre el sistema y pueden tomar decisiones de manera local. Sin embargo, ellos sólo optimizan criterios locales, los cuales pueden dar como resultado decisiones subóptimas en el nivel de proyecto, en especial cuando se dificulta la coordinación entre equipos. Además de una organización basada en equipos, se han propuesto diferentes organizaciones de equipos alternas en la literatura.

La **organización de programador en jefe** [Brooks, 1975] es una organización jerárquica que asigna todas las decisiones del equipo al líder de equipo, llamado **programador en jefe**. El programador en jefe es responsable del diseño y desarrollo del subsistema y delega las tareas de programación a un conjunto de especialistas. El programador en jefe está apoyado por un asistente, quien lo sustituye cuando se necesita, y por un bibliotecario, que es responsable de la documentación y de todas las funciones de oficina. El programador en jefe tiene la última palabra en todo.

Mientras que la organización de programador en jefe toma una posición jerárquica sobre la administración, la **programación sin ego** [Weinberg, 1971] toma una democrática. La responsabilidad se asigna al equipo en vez de a los individuos. Se critican los resultados y no a los participantes. Los problemas se resuelven por votación. La estructura del equipo es informal y la comunicación sucede de manera espontánea.

En general, hay un espectro de organizaciones que van desde una jerarquía estricta, como una organización militar, hasta un conjunto dinámico de grupos colaboradores. Raymond [Raymond, 1998] se refiere al primero como **modelo catedral** y al último como **modelo bazar**. Hasta hace poco se suponía que el modelo bazar sólo era apropiado para sistemas simples desarrollados por un equipo de programadores experimentados. Era común pensar que el modelo catedral era necesario para el manejo de la complejidad de los sistemas. El sistema operativo Linux[1] ha proporcionado un contraejemplo. Linux ha sido desarrollado a partir de cero por miles de desarrolladores, y la mayoría de ellos nunca se ha reunido en persona. Además, la mayoría de los desarrolladores son contribuyentes de tiempo parcial que no reciben ninguna retribución financiera por contribuir con código para Linux.

La complejidad del dominio de aplicación, la experiencia de los desarrolladores y administradores, su ubicación y su infraestructura de comunicaciones determinan, a final de cuentas, en qué punto se encuentra la organización del proyecto en el espectro catedral/bazar. Tal vez encontremos con frecuencia que la organización del proyecto se inicia cerca del extremo catedral del espectro y se mueve hacia el extremo bazar, una vez que los desarrolladores adquieren experiencia, se conocen entre ellos y desarrollan y comunican un modelo compartido del sistema.

Organizaciones corporativas

El énfasis de este libro es en los proyectos de desarrollo de software y, en consecuencia, limitamos el alcance de nuestra discusión a los papeles que van hasta, incluso, el gerente del

1. Linux es un sistema operativo Unix disponible en forma gratuita creado por Linus Torwalds. Para mayor información vea http://www.linux.org/.

proyecto. Necesitamos reconocer que cada proyecto se da en el contexto de una organización más amplia, como una corporación o una universidad. Un proyecto se forma seleccionando entre las habilidades, experiencia y herramientas disponibles dentro de la organización más amplia.

En la **organización funcional** los participantes se agrupan por especialidades en departamentos. Un departamento de analistas construye el modelo de análisis, un departamento de diseñadores lo transforma en un modelo de diseño, el aseguramiento de la calidad lo maneja otro departamento y así sucesivamente. La organización funcional corresponde, por lo general, a un orden secuencial de las actividades del ciclo de vida, que es un modelo de cascada. Los proyectos se entuban por las organizaciones funcionales, dando como resultado que sólo unos cuantos de los participantes estén involucrados durante la existencia completa del proyecto. Este tipo de organización sólo es adecuado para problemas bien definidos donde la tasa de cambio es baja.

La **organización basada en proyecto** es una organización en la que los participantes se asignan a un solo proyecto a la vez durante toda la existencia del proyecto. Los participantes están agrupados, por lo general, por subsistemas. Sin embargo, el flujo de información entre equipos todavía es bajo, ya que no hay equipos de funcionalidad cruzada o enlaces. La comunicación y los reportes se hacen por medio de la estructura de reporte. La organización basada en proyecto tiene mejores capacidades para manejar el cambio que la organización funcional. La organización basada en equipo que describimos es un refinamiento de la organización basada en proyecto, en donde la comunicación entre equipos la manejan los equipos de funcionalidad cruzada. Una compañía virtual es una organización basada en proyecto. Un conjunto de grupos heterogéneos se ensambla bajo un paraguas durante la existencia de un solo proyecto.

La **organización de matriz** es una combinación de las organizaciones funcional y basada en proyecto. Cada participante pertenece a un departamento y está asignado a un proyecto. Cuando el participante deja de estar involucrado con el proyecto se le reasigna a otro proyecto. La organización de matriz sufre el problema del "doble jefe" cuando entran en conflicto los intereses del departamento y del proyecto.

11.3.2 Papeles

Una vez que el proyecto está organizado en equipos necesitamos definir la manera en que se asignan responsabilidades individuales a los miembros del equipo. En esta sección definimos la manera de asignar un conjunto de responsabilidades mediante la asignación de un **papel**. Un papel define los tipos de tareas técnicas y administrativas que se esperan de una persona. Por ejemplo, el papel de probador de un equipo de subsistema tiene la responsabilidad de definir conjuntos de pruebas para el subsistema que se está desarrollando, de ejecutar esas pruebas y de reportar a los desarrolladores los defectos descubiertos.

Distinguimos entre cinco tipos de papeles.

- Los **papeles administrativos** están relacionados con la organización y ejecución del proyecto con restricciones. Estos papeles incluyen al gerente del proyecto y al líder de equipo.
- Los **papeles de desarrollo** están relacionados con la especificación, el diseño y la construcción de subsistemas. Estos papeles incluyen al analista, al arquitecto del sistema, al diseñador de objetos y al implementador.

- Los **papeles de funcionalidad cruzada** están relacionados con la coordinación de más de un equipo.
- Los **papeles de consultor** están relacionados con proporcionar apoyo en áreas donde les falta experiencia a los participantes en el proyecto. Los usuarios y el cliente actúan, en la mayoría de los proyectos, como consultores sobre los problemas del dominio. Los consultores técnicos también pueden proporcionar su experiencia sobre nuevas tecnologías o métodos.
- Los **papeles de promotor** están relacionados con la promoción de los cambios por toda la organización. Una vez que se ha establecido la organización, se tiene listo el plan de tareas y el proyecto está funcionando en estado estable, llega a ser cada vez más difícil introducir cambios en el proyecto. Los promotores son papeles para superar las barreras ante el cambio.

En la sección anterior describimos los papeles de administración. En los capítulos 5, 6 y 7 describimos los papeles de desarrollo. A continuación describimos, con mayor detalle, los papeles de funcionalidad cruzada, de consultor y de promotor.

Papeles de funcionalidad cruzada

Los papeles de funcionalidad cruzada están relacionados con la coordinación entre equipos. Los desarrolladores que ocupan estos papeles son responsables del intercambio de información relevante con los demás equipos y de la negociación de los detalles de interfaces.

El **enlace** es responsable de la diseminación de información de un equipo hacia otro. En algunos casos (como el del ingeniero API), el enlace funciona como un representante del equipo de subsistema y se le puede llamar para resolver problemas entre equipos. Hay seis tipos de papeles de funcionalidad cruzada:

- El **ingeniero API** es responsable de la definición de la interfaz del subsistema que tiene asignado. La interfaz tiene que reflejar la funcionalidad que ya se ha asignado al subsistema y satisfacer las necesidades de los subsistemas dependientes. Cuando hay compromisos entre la funcionalidad y los demás subsistemas, que dan lugar a cambios en los subsistemas, el ingeniero API es responsable de la propagación de los cambios hacia el equipo del subsistema.
- El **editor de documentos** es responsable de la integración de los documentos producidos por un equipo. Puede verse a un editor de documentos como un proveedor de servicios hacia los demás equipos que dependen de un equipo de subsistema dado. Un editor de documentos también administra la información que lanza internamente el equipo, como las agendas y minutas de reuniones.
- El **gerente de configuración** es responsable del manejo de diferentes versiones de los documentos, modelos y código producidos por un equipo. Para políticas simples de administración de la configuración (por ejemplo, plataforma de hardware única, una sola rama), este papel puede desempeñarlo la misma persona que es el líder del proyecto.
- Un **probador** es responsable de asegurar que cada subsistema trabaje como lo ha especificado el diseñador. A menudo, los proyectos de desarrollo tienen un equipo aparte que sólo

es responsable de las pruebas. La separación de los papeles de diseñador, implementador y probador conduce a pruebas más efectivas.

Papeles de consultor

Los papeles de consultor atienden las necesidades de conocimiento a corto plazo. Por lo general, los consultores no están involucrados con el proyecto de tiempo completo. Hay tres tipos de papeles de consultor.

- El **cliente** es responsable de la formulación de escenarios y de los requerimientos. Esto incluye los requerimientos funcionales y no funcionales, así como las restricciones. Se espera que el cliente pueda interactuar con los papeles de los demás desarrolladores. El cliente representa al usuario final del sistema.
- El **especialista del dominio de aplicación** es responsable de proveer el conocimiento del dominio acerca de un área funcional específica del sistema. Mientras que el cliente tiene una visión global de la funcionalidad del sistema, el especialista del dominio de aplicación tiene conocimiento detallado sobre las operaciones diarias de una parte específica del sistema.
- El **consultor técnico** es responsable de proporcionar el conocimiento que les falta a los desarrolladores sobre un tema específico. Este tema puede estar relacionado con el método de desarrollo, el proceso, la tecnología de implementación o el ambiente de desarrollo.

Promotores

Cuando ya está instalada la organización, se han asignado los papeles a los participantes y se les ha entrenado, llega a ser cada vez más difícil introducir cambios en el proyecto. Para manejar este problema, en especial en las organizaciones jerárquicas y en las funcionales, los papeles de promotor son esenciales para realizar un cambio de organización [Hauschildt y Gemuenden, 1998]. Los promotores son individuos autonombrados que se identifican con el resultado del proyecto. Son miembros de la organización corporativa y puede ser que no estén involucrados de manera directa en el proyecto. En vez de ello, son interfaces hacia el resto de la organización corporativa. Debido a su poder, conocimiento de la tecnología o familiaridad con los procesos del proyecto, pueden promover e impulsar cambios específicos por toda la organización. Los promotores se parecen a los revolucionarios: su trabajo principal es personificar el cambio eliminando la organización antigua. Sin embargo, esta visión es incorrecta. Los promotores, a diferencia de los revolucionarios, permanecen dentro de la estructura de la organización actual. Para superar las barreras usan incluso el poder jerárquico para pasar hacia el nuevo proceso. Hay tres promotores principales para que suceda cualquier cambio: el promotor poderoso, el promotor con conocimientos y el promotor de procesos.

El **promotor poderoso**, al que también se llama campeón ejecutivo, presiona el cambio mediante la jerarquía de la organización existente. El promotor poderoso convence, motiva, crea entusiasmo, castiga y recompensa. No es necesario que el promotor poderoso esté en la parte más alta de la organización o cercana a ella, pero debe tener protección de la alta gerencia, ya que de no ser así los que se oponen al proyecto pueden impedir el éxito del mismo. Sin embargo, el promotor poderoso no evita la confrontación con la alta gerencia. Necesita identificar dificultades

en forma constante, resolver problemas y comunicarse con los miembros del proyecto, en especial con los desarrolladores.

El **promotor con conocimientos**, al que también se llama tecnólogo, promueve el cambio que se presenta en el dominio de aplicación o en el de solución. Es un especialista debido a capacidades específicas y, por lo general, está asociado con el promotor poderoso. El promotor con conocimientos adquiere información en forma iterativa, comprende los beneficios y limitaciones de las nuevas tecnologías y argumenta sobre su adopción con los demás desarrolladores.

El **promotor de procesos** tiene un conocimiento profundo de los procesos y procedimientos del proyecto. El promotor de procesos está en interacción constante con el promotor poderoso para obtener consenso sobre los objetivos generales. Además, el promotor de procesos tiene el suficiente conocimiento administrativo y técnico para proporcionar un puente entre los promotores poderoso y con conocimientos, quienes a menudo no hablan o comprenden el mismo lenguaje. Un ejemplo de un promotor de procesos es el dirigente del desarrollo, quien a veces se encuentra en los equipos de desarrollo. Este papel, que no cubrimos con detalle en este capítulo, es responsable de los aspectos administrativos del proyecto, incluyendo la planeación, la definición de indicadores de avance, presupuestación e infraestructura de comunicaciones.

11.3.3 Productos de trabajo

La asignación de un papel a un participante se traduce, por lo general, en la asignación de la responsabilidad sobre un producto de trabajo específico. Un **producto de trabajo** es un artefacto que se produce durante el desarrollo, como un documento interno para los demás participantes en el proyecto o un producto que se va a entregar al cliente. Un analista es responsable de una sección del RAD que describe la funcionalidad del sistema. Un arquitecto de sistema es responsable del SDD que describe la arquitectura de alto nivel del sistema. Un gerente de proyecto es responsable del SPMP que describe los procesos del desarrollo. Por último, el sistema y sus documentos acompañantes también constituyen un conjunto de productos de trabajo que se entregan al cliente. Los productos de trabajo complejos, como el RAD, pueden tratarse como la composición de varios productos de trabajo más pequeños, como los requerimientos para diferentes tipos de usuarios.

Los productos de trabajo son el resultado de tareas y están sujetos a tiempos de entrega. Los productos de trabajo alimentan, por lo general, a otras tareas. Por ejemplo, la actividad de planeación para el subsistema de base de datos da como resultado un producto de trabajo que incluye varias series de pruebas y sus resultados esperados. La serie de pruebas se alimenta luego a la actividad de pruebas del subsistema dado (figura 11-7).

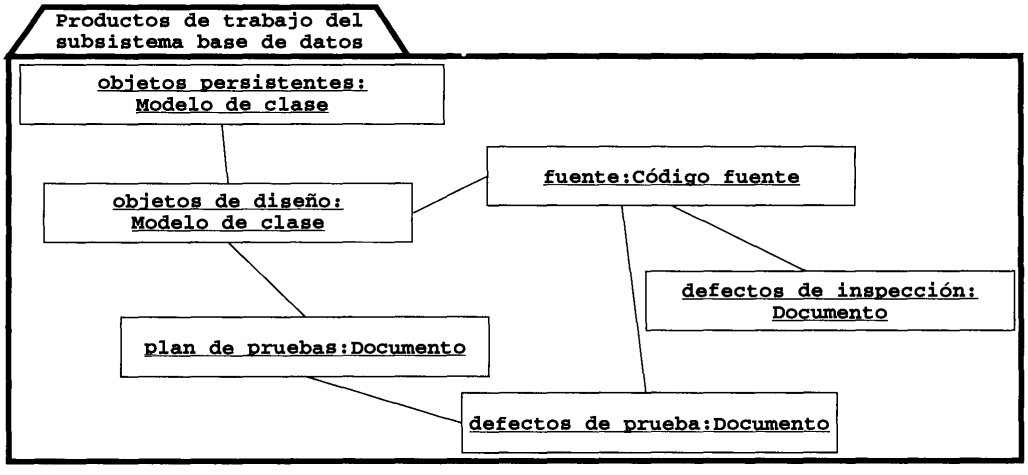


Figura 11-7 Productos de trabajo para un equipo de subsistema de base de datos (diagrama de objetos UML). Las asociaciones representan dependencias entre productos de trabajo.

Los productos de trabajo son artefactos de administración importantes, debido a que podemos valorar su entrega y el inicio de las tareas que dependen de productos de trabajo específicos. La entrega tardía de una serie de pruebas para un subsistema, por ejemplo, retrasará el inicio de sus pruebas. Sin embargo, tome en cuenta que no es suficiente enfocarse en una entrega a tiempo: el apresuramiento de la entrega de series de pruebas cumplirá con la calendarización del proyecto, pero también puede significar que no se descubrirán a tiempo las fallas críticas.

11.3.4 Tareas

Un papel describe al trabajo desde el punto de vista de un conjunto de tareas. Una tarea incluye una descripción, una duración y está asignada a un papel. Una tarea representa una unidad de trabajo atómica que puede administrarse: un gerente la asigna a un desarrollador, el desarrollador la realiza y el gerente supervisa el avance y terminación de la tarea. Las tareas consumen recursos, dan como resultado productos de trabajo y dependen de productos de trabajo originados por otras tareas. La tabla 11-1 proporciona ejemplos simples de tareas.

Tabla 11-1 Ejemplos de tareas para la realización del subsistema de base de datos.

Nombre de la tarea	Papel asignado	Descripción de la tarea	Entrada de la tarea	Salida de la tarea
Obtención de requerimientos del subsistema de base de datos	Diseñador del sistema	Obtiene requerimientos de los equipos de subsistemas acerca de sus necesidades de almacenamiento, incluyendo objetos persistentes, sus atributos y sus relaciones	Enlaces de equipos	API del subsistema de base de datos, modelo de análisis de objetos persistentes (diagrama de clase UML)
Diseño del subsistema de base de datos	Diseñador del subsistema	Diseña el subsistema de base de datos, incluyendo la posible selección de un producto comercial	API del subsistema	Diseño del subsistema de base de datos (diagrama UML)
Implementación del subsistema de base de datos	Implementador	Implementa el subsistema de base de datos	Diseño del subsistema	Código fuente del subsistema de base de datos
Inspección del subsistema de base de datos	Implementador	Realiza una inspección del código del subsistema de base de datos	Código fuente del subsistema	Lista de defectos descubiertos
Plan de pruebas del subsistema de base de datos	Probador	Desarrolla una serie de pruebas para el subsistema de base de datos	Diseño del subsistema, código fuente del subsistema	Pruebas y plan de pruebas
Prueba del subsistema de base de datos	Probador	Ejecuta la serie de pruebas para el subsistema de base de datos	Plan de pruebas del subsistema	Resultados de las pruebas, lista de defectos descubiertos

Las tareas están relacionadas con dependencias. Por ejemplo, en la tabla 11-1 la `Prueba del subsistema de base de datos` no puede comenzar antes de que termine la tarea `Implementación del subsistema de base de datos`. La localización de las dependencias entre tareas nos permite saber cuáles tareas pueden ejecutarse en paralelo. Por ejemplo, en la tabla 11-1 la `Inspección del subsistema de base de datos` y el `Plan de pruebas del subsistema de base de datos` pueden asignarse a dos desarrolladores diferentes y realizarse al mismo tiempo. Al conjunto de tareas y sus dependencias se le llama modelo de tareas. La figura 11-8 muestra en UML el modelo de tareas que corresponde a la tabla 11-1.

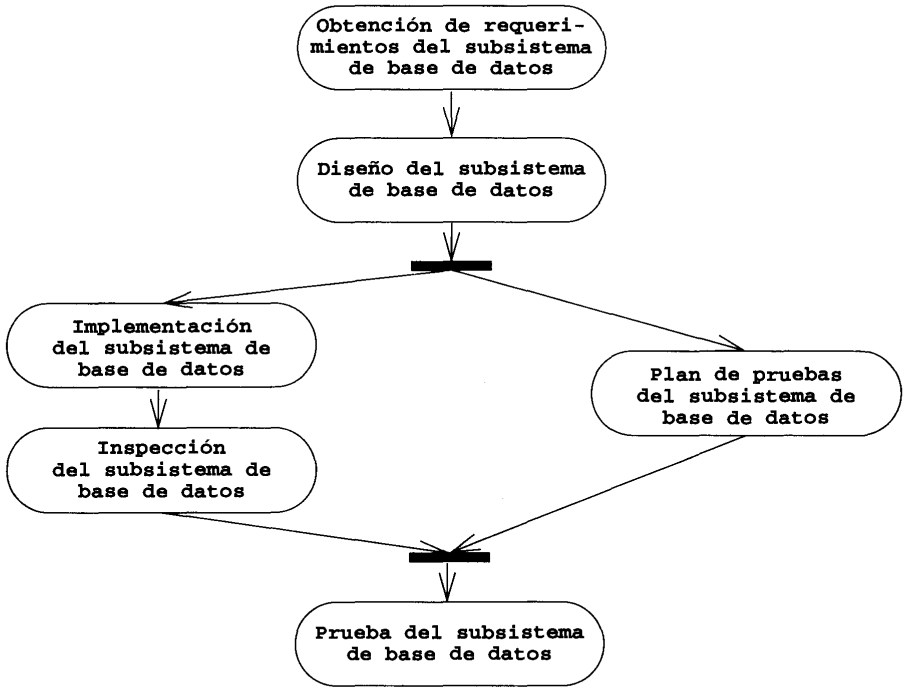


Figura 11-8 Un ejemplo de un modelo de tareas con dependencias de precedencia para el ejemplo de subsistema de base de datos de la tabla 11-1 (diagrama de actividad UML).

Las dependencias entre tareas son relaciones entre las tareas individuales y el tiempo. Pueden usarse restricciones de tareas para asegurar que un proyecto esté a tiempo, independientemente de la relación de la tarea con otras tareas. Por lo general, representan una interacción con el cliente o el usuario. Por ejemplo, si la tarea T tiene que comenzar en un lunes tiene una restricción MSO (vea la tabla 11-2). Si el sistema debe entregarse en la primera semana de diciembre tiene una restricción MFO. ASAP es la restricción más usada por quienes ordenan las tareas (por ejemplo, gerentes de proyecto) y ALAP es la más usada por quienes las reciben.

11.3.5 Calendarización

El modelo de tareas contiene las tareas, sus dependencias, sus restricciones y su duración planeada. En forma similar a los modelos de objetos, donde se ilustran los objetos y sus dependencias, hay muchas formas de representar los modelos de tareas. Una calendarización es la correspondencia entre las tareas y el tiempo: a cada tarea se le asignan tiempos de inicio y terminación. Esto nos permite planear las fechas de entrega de los productos individuales a entregar. Las dos notaciones usadas con mayor frecuencia para la calendarización son los diagramas PERT y Gantt [Hillier y Lieberman, 1967]. Una **gráfica de Gantt** es una forma compacta para presentar la calendarización de un proyecto de software a lo largo del eje del tiempo. Una gráfica de Gantt

Tabla 11-2 Restricciones de las tareas

Restricción	Definición
ASAP	Iniciar la tarea tan pronto como sea posible
ALAP	Tan tarde como sea posible
FNET	No terminar antes que
SNET	No iniciar antes que
FNLT	No terminar después que
SNLT	No iniciar después que
MFO	Debe terminar en
MSO	Debe iniciar en

es una gráfica de barras en donde el eje horizontal representa el tiempo. El eje vertical representa las diferentes tareas a realizar. Las tareas se representan con barras cuya longitud corresponde a la duración planeada de la tarea. En la figura 11-9 se representa como gráfica de Gantt una calendarización para el ejemplo del subsistema de base de datos.

Una **gráfica PERT** representa una calendarización como una gráfica acíclica de tareas. La figura 11-10 es una gráfica PERT para el subsistema de base de datos. El inicio y la duración planeados de la tarea se usan para calcular la ruta crítica, que representa la ruta más corta posible a través de la gráfica. La longitud de la ruta crítica corresponde a la calendarización más corta posible, suponiendo que se tienen los recursos suficientes para realizar en paralelo las tareas que son independientes. Además, las tareas de la ruta crítica son las más importantes, ya que un retraso en cualquiera de estas tareas dará como resultado un retraso en el proyecto general. Las tareas y barras representadas con líneas gruesas pertenecen a la ruta crítica.

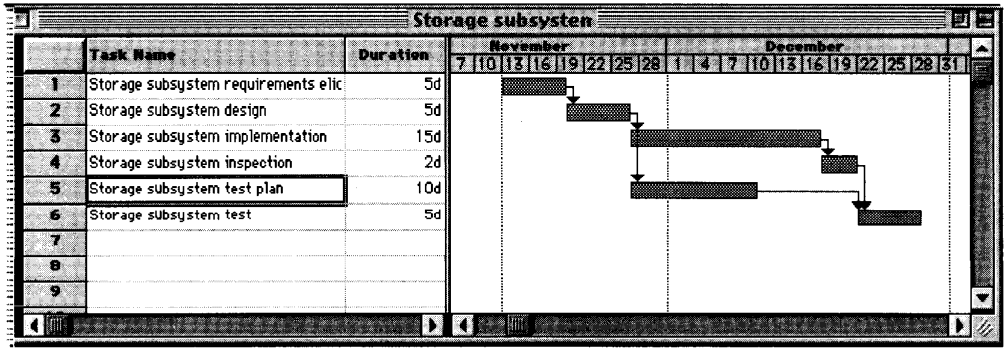


Figura 11-9 Un ejemplo de calendarización para el subsistema de base de datos (gráfica de Gantt).

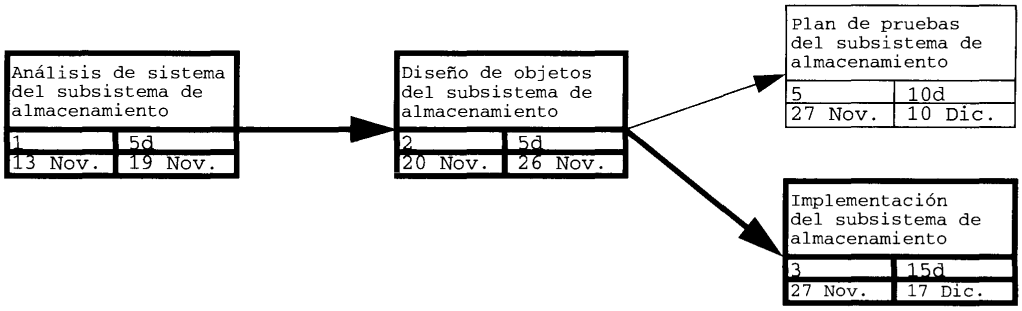


Figura 11-10 Un ejemplo de una calendarización para el subsistema de base de datos (gráfica PERT). Las tareas de la ruta crítica se representan con líneas gruesas.

Las gráficas PERT y Gantt son herramientas útiles para la planeación de un proyecto y el seguimiento de su ejecución. Sin embargo, tome en cuenta que estos modelos sólo son tan buenos como la estimación que representan. La estimación precisa de la duración de una tarea es difícil. Los gerentes de proyecto usan su experiencia, datos de proyectos anteriores y su intuición para estimar las tareas requeridas para la ejecución del proyecto y su duración. La estimación imprecisa, como la omisión de una tarea principal o la subestimación de una tarea de la ruta crítica, da lugar a retrasos considerables en la calendarización. Los eventos inesperados, como los cambios en los requerimientos o el descubrimiento tardío de errores de diseño, desorganizan la calendarización, sin importar lo preciso de la planeación. Un gerente de proyecto debe tratar de anticipar estas situaciones indeseables permitiendo márgenes de error y dejando espacio para los cambios en la calendarización. Un ejemplo de esto es una tarea dedicada a la revisión periódica y al cambio de requerimientos durante el proyecto. En general, las gráficas PERT y Gantt son de utilidad limitada si no se conocen las tareas durante la planeación o si las tareas cambian con frecuencia.

En la siguiente sección describimos la manera en que la administración de proyectos organiza a los participantes en equipos, les asigna papeles, planea el proyecto en función de tareas y, lo más importante, revisa la organización, la asignación de papeles y las tareas conforme avanza el proyecto y encuentra eventos imprevistos.

11.4 Actividad de la administración de proyectos

En esta sección describimos las actividades de la administración de proyectos, las cuales incluyen todas las actividades necesarias para definir y revisar la organización, los papeles, los productos de trabajo, las tareas y la calendarización del proyecto. La mayor parte del esfuerzo del líder de equipo se dedica a supervisar y administrar a un equipo durante el estado estable. Sin embargo, los líderes de equipo deben estar conscientes de las actividades del inicio del proyecto que preceden a su involucramiento con el proyecto y de la interacción con el cliente.

Nos enfocamos en las siguientes actividades administrativas:

- **Inicio del proyecto** (sección 11.4.2), que incluye la definición del problema, la identificación del plan inicial de tareas y la asignación de recursos a las tareas.
- **Estado estable,** que incluye la supervisión del proyecto (sección 11.4.3), la administración del riesgo (sección 11.4.4) y el acuerdo del proyecto (sección 11.4.5).
- **Terminación del proyecto,** que incluye la prueba de aceptación del cliente (sección 11.4.6), la instalación (sección 11.4.7) y el post mortem del proyecto (sección 11.4.8).

Ilustramos las actividades administrativas del proyecto usando como ejemplo el proyecto OWL [Bruegge *et al.*, 1997]. El objetivo del OWL es proporcionar un sistema para edificios inteligentes que permita que los trabajadores ajusten su ambiente, que los administradores detecten con rapidez los componentes que fallan y que las compañías reconfiguren el plano del piso basadas en sus necesidades cambiantes. El OWL supone el uso e integración de nuevas tecnologías, incluyendo sensores para vigilar el ambiente de trabajo, actuadores para controlar la temperatura, humedad e iluminación del ambiente de trabajo, kernels de control en tiempo real distribuidos y tecnología Web. Los requerimientos del OWL todavía no se han estabilizado, ya que todavía se están considerando diferentes fabricantes. Estos factores causarán varios cambios más adelante en el desarrollo e introducirán riesgos significativos en el proyecto, y por eso el OWL sirve como base para los ejemplos de esta sección.

11.4.1 Edificios de oficinas inteligentes

Un edificio de oficinas inteligente pretende proporcionar servicios individualizados a sus trabajadores y, al mismo tiempo, optimizar el consumo de recursos. Los trabajadores de los edificios inteligentes pueden ajustar muchos parámetros de su ambiente individual, como el nivel de iluminación, la temperatura y la velocidad y dirección del flujo de aire, en forma similar a como ya lo hacen en sus automóviles. Además, un diseño modular permite que los espacios de trabajo se reconfiguren con rapidez para satisfacer necesidades de organización o tecnológicas cambiantes. La adición de una computadora o una unidad de videoconferencia a un espacio de trabajo puede hacerse con un realambrado limitado. Por último, los sistemas de control integrados en los edificios inteligentes facilitan su mantenimiento: los focos fundidos o componentes defectuosos se detectan en forma remota y se reemplazan en un tiempo muy corto.

Un componente crítico de los edificios inteligentes es su sistema de control. El sistema de control es un sistema distribuido en tiempo real que:

- Responde a los parámetros especificados por el trabajador controlando las persianas, las unidades de calefacción y enfriamiento individuales y las ventanas con base en el clima actual para ajustar la temperatura y el flujo de aire.
- Registra datos del edificio para análisis futuros.
- Realiza diagnósticos.
- Notifica a los administradores de servicios acerca de cualquier problema o emergencia.

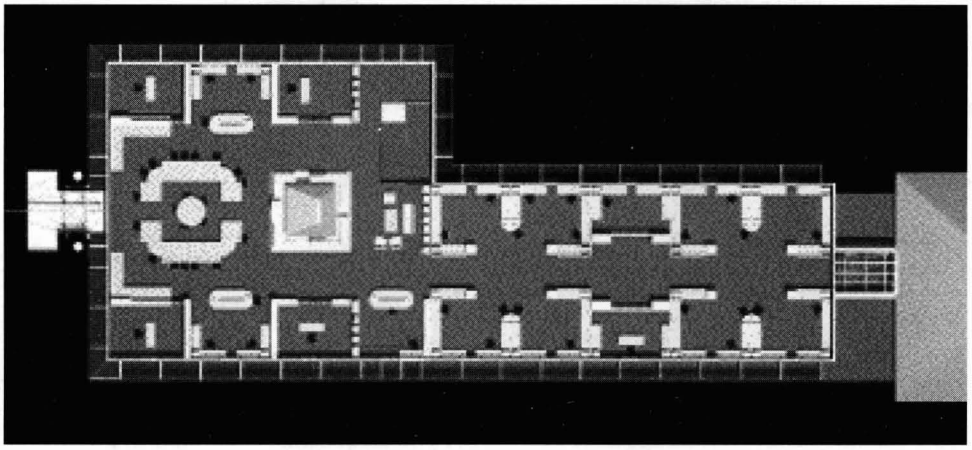


Figura 11-11 Plano de planta del Espacio de trabajo inteligente, un edificio inteligente en el Departamento de Arquitectura de la Universidad Carnegie Mellon.

Tales sistemas de control también necesitan ser modulares y modificables para acomodar nuevos componentes y equipos de una variedad de fabricantes. La figura 11-11 muestra el plano de planta del Espacio de trabajo Inteligente, un edificio inteligente en la Universidad Carnegie Mellon [Hartkopf *et al.*, 1997]. En este capítulo describimos ejemplos de administración de proyectos de OWL, un sistema de control para el Espacio de trabajo inteligente.

11.4.2 Inicio del proyecto

El inicio del proyecto se enfoca en la definición del problema, la planeación de una solución y la asignación de recursos. El inicio del proyecto da como resultado los siguientes productos de trabajo (figura 11-12):

- El **enunciado del problema** es un documento corto que describe el problema que debe resolver el sistema, el ambiente de destino, los productos a entregar al cliente y los criterios de aceptación. El enunciado del problema es una descripción inicial, y es la semilla para el *Acuerdo del proyecto,* que formaliza la comprensión común del proyecto por parte del cliente y la administración, y para el RAD, que es una descripción precisa del sistema que se está desarrollando.
- El **diseño de alto nivel** representa la descomposición inicial del sistema en subsistemas. Se usa para asignar subsistemas a equipos individuales. El diseño de alto nivel también es la semilla para el SDD, el documento de la arquitectura del software.
- La **organización** describe los equipos iniciales del proyecto, sus papeles y sus rutas de comunicación. El **plan inicial de tareas** y la **calendarización inicial** son las descripciones iniciales de la manera de asignar recursos. La organización, el plan inicial de tareas y la calendarización inicial son las semillas para el SPMP, que documenta todos los aspectos administrativos del proyecto.

Durante el inicio del proyecto la gerencia ensambla los equipos de acuerdo con el diseño de alto nivel y las organizaciones seleccionadas, establece la infraestructura de comunicación y luego arranca el proyecto organizando la primera reunión. Una vez que se han asignado las tareas a todos los participantes y se encuentran en su lugar los mecanismos de reporte, se considera que el proyecto está en estado estable.

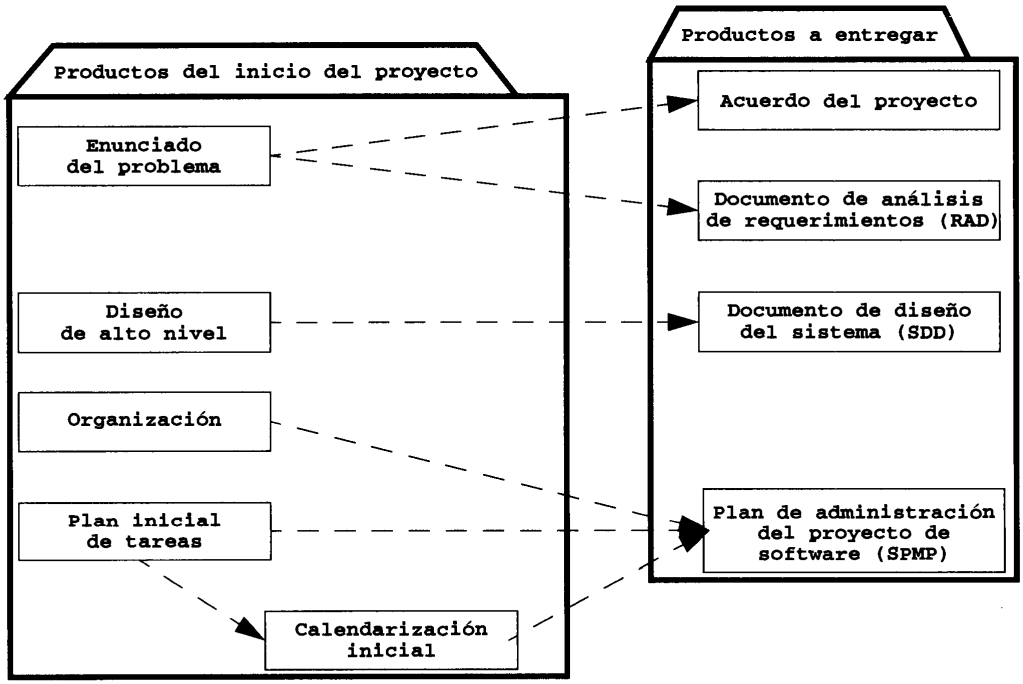


Figura 11-12 Productos de trabajo generados durante el inicio del proyecto y su relación con los productos a entregar típicos de un proyecto (diagrama de clase UML).

Desarrollo del enunciado del problema

El *enunciado del problema* lo desarrollan el gerente del proyecto y el cliente como una comprensión mutua del problema a resolver con el sistema. El enunciado del problema describe la situación actual, la funcionalidad que debe soportar y el ambiente en que se desplegará el sistema. También define los productos a entregar que espera el cliente, junto con las fechas de entrega y un conjunto de criterios de aceptación. El enunciado del problema también puede especificar restricciones en el ambiente de desarrollo, como el lenguaje de programación a usar. El enunciado del problema no es una especificación precisa o completa del sistema. En vez de ello, es un resumen de alto nivel de dos documentos del proyecto que todavía no se desarrollan: el RAD y el SPMP.

El enunciado del problema lo desarrollan de manera iterativa el gerente del proyecto y el cliente. El desarrollo del enunciado del problema es tanto una actividad de negociación como

una de recopilación de información, y durante éste ambas partes aprenden las expectativas y restricciones de la otra. Aunque el enunciado del problema contiene una descripción de alto nivel de la funcionalidad del sistema, también debe proporcionar ejemplos concretos para asegurar que ambas partes comparten la misma visión. Esto se logra mejor usando escenarios (vea el capítulo 4, *Obtención de requerimientos*) para describir la situación actual y la funcionalidad futura.

La figura 11-13 es un ejemplo de esquema para el enunciado del problema. La primera sección describe el dominio del problema. La sección 2 proporciona escenarios de ejemplo que describen la interacción entre los usuarios y el sistema para la funcionalidad esencial. Los escenarios se usan para describir la situación actual y la futura.

ENUNCIADO DEL PROBLEMA

1. Dominio del problema
2. Escenarios
3. Requerimientos funcionales
4. Requerimientos no funcionales
5. Ambiente de destino
6. Productos a entregar y fechas de entrega

Figura 11-13 Esquema del documento de enunciado del problema. Observe que las secciones 2, 3 y 4 no son una especificación completa o precisa del sistema. Proporcionan las bases para la obtención de requerimientos. Sin embargo, la sección 6 describe los productos contractuales a entregar y sus fechas de entrega asociadas.

La figura 11-14 proporciona ejemplos de escenarios usados en el enunciado del problema de OWL. La sección 3 es un resumen de los requerimientos funcionales del sistema. La sección 4 es un resumen de los requerimientos no funcionales, incluyendo las restricciones del cliente, como el lenguaje de programación y la reutilización de componentes. La sección 5 es una descripción del ambiente de la organización, incluyendo la plataforma, el ambiente físico, los usuarios, etc. La sección 6 es una lista de productos a entregar al cliente y sus fechas de entrega asociadas.

Definición del diseño de alto nivel

El diseño de alto nivel describe la arquitectura de software del sistema. En una organización basada en proyecto o equipo, el diseño de alto nivel se usa como base para la organización: cada subsistema se asigna a un equipo y éste es responsable de su definición y realización. La descomposición en subsistemas se refina y modifica más adelante durante el diseño del sistema (vea el capítulo 6, *Diseño del sistema*). Sin embargo, sólo necesitamos identificar los sistemas principales y sus servicios, y en este momento no necesitamos definir sus interfaces. Los equipos que trabajan en subsistemas dependientes negociarán los servicios individuales y sus interfaces conforme lo necesiten.

El diseño de alto nivel del OWL incluye cinco subsistemas principales, que se describen en la figura 11-15. Observe que esta descomposición en subsistemas es de alto nivel y se enfoca

ENUNCIADO DEL PROBLEMA DE OWL

1. Dominio del problema

Una tendencia actual en la industria de la construcción es proporcionar servicios y control distribuidos para los ocupantes individuales como una estrategia para corregir la dependencia excesiva de los grandes sistemas centralizados que caracterizan a los edificios de oficinas construidos durante los últimos 30 años. En el Espacio de trabajo inteligente los trabajadores tendrán más control sobre sus condiciones ambientales, como el ajuste del nivel de brillantez y temperatura de su espacio de trabajo, la reducción de la luminosidad, el control de la velocidad y dirección del flujo de aire que entra al espacio de trabajo. (Ya puede hacerse esto en un automóvil, ¿por qué no en la oficina?) Una fachada con eficiencia energética permitirá la ventilación de aire fresco con ventanas operables e incorporará dispositivos de sombreado movibles que se ajusten para disminuir el brillo y maximizar la iluminación natural del espacio de trabajo.

Es deseable adoptar tres formas de control en el Espacio de trabajo inteligente: sensible, calendarizada y manejada por el usuario. El control sensible se da cuando el sistema reacciona ante un cambio en la lectura de un sensor haciendo actuar a algún componente. El control calendarizado puede adoptarse en presencia de datos predecibles que permiten que los componentes se controlen en forma directa mediante una calendarización diseñada con cuidado. Por ejemplo, debido a que es predecible la posición del sol, puede adoptarse una calendarización para las persianas interiores del Espacio de trabajo inteligente. El sistema de control debe ser lo bastante flexible para que responda a las necesidades de los ocupantes. Si quisieran cambiar la temperatura de su ambiente local debería dárseles la oportunidad de hacerlo.

En este proyecto se le pide que construya un sistema, llamado laboratorio de espacio de trabajo orientado a objetos (OWL, por sus siglas en inglés), que trate de mejorar la manera en que manejamos los edificios.

[. . .]

2. Escenarios

2.1 Control del edificio

Los ocupantes del edificio usan un navegador Web para acceder a su Módulo de ambiente personal (PEM, por sus siglas en inglés). Ahí ajustan la temperatura y velocidad del aire para refrescar su espacio de trabajo. La información de control se envía al equipo PEM. Las acciones de control se registran en la base de datos y el equipo ajusta la calefacción y la ventilación del espacio de trabajo. El sistema revisa los PEM vecinos para ver si el enfriamiento de este espacio particular requiere que se incremente la calefacción de otros espacios de trabajo.

[. . .]

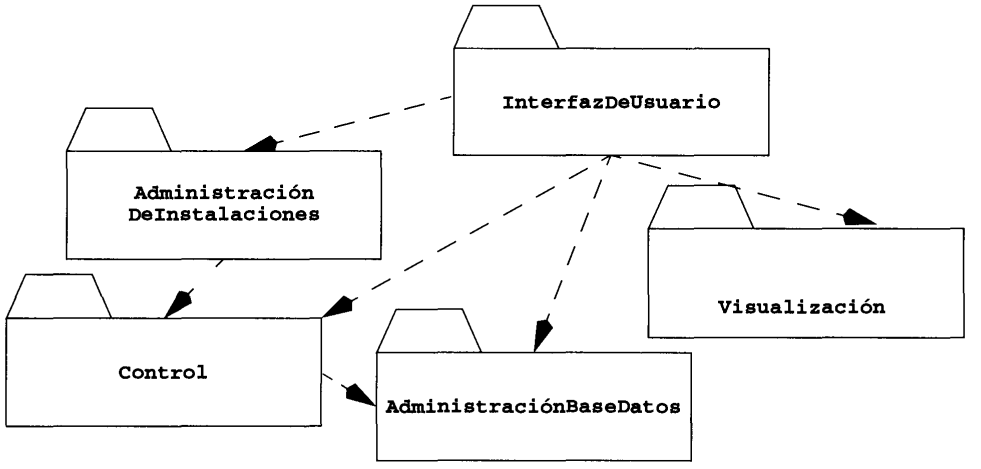
2.5 Mantenimiento del edificio

El sistema vigila el comportamiento de los dispositivos controlados para detectar defectos en el sistema. Los focos defectuosos y las lecturas inusuales de parámetros se reportan al administrador de las instalaciones, quien planea las inspecciones y las reparaciones. La ocurrencia de los defectos de los dispositivos se registra y analiza buscando tendencias, permitiendo que el administrador de las instalaciones anticipe defectos futuros.

[. . .]

Figura 11-14 Fragmentos del enunciado del problema de OWL [OWL, 1996].

en la funcionalidad. Después podrán añadirse otros subsistemas al proyecto, como la capa de comunicaciones para mover objetos entre nodos en la red y un subsistema de notificación para enviar noticias a los usuarios. Una vez que el análisis produce un modelo estable y se revisa la descomposición en subsistemas durante el diseño de subsistemas, durante la fase de diseño del sistema pueden crearse nuevos subsistemas y tal vez tenga que cambiar la organización para que refleje el nuevo diseño.



Subsistema	Servicios
`Control`	• Interfaz con sensores y actuadores
	• Proporcionar algoritmos reactivos para mantener la temperatura y humedad
`Administración-BaseDatos`	• Proporcionar el archivado de datos sobre la operación del edificio (sensor y control)
	• Capturar y almacenar datos de pronóstico del clima
`Administración-DeInstalaciones`	• Mantener el plano de sensores, actuadores y cuartos
	• Mantener los diagramas de cableado y estructura de la red
	• Mantener un plano actualizado de las configuraciones de los espacios de trabajo (por ejemplo, mobiliario)
`InterfazDeUsuario`	• Proporcionar una interfaz de navegador Web para los usuarios a fin de que visualicen y especifiquen los parámetros ambientales
	• Proporcionar una interfaz de voz
	• Proporcionar una interfaz para administrar las instalaciones
`Visualización`	• Proporcionar navegación por un modelo tridimensional del edificio
	• Proporcionar visualización tridimensional de la temperatura y el consumo de energía
	• Proporcionar visualización tridimensional de problemas y emergencias

Figura 11-15 Diseño de alto nivel del OWL (diagrama de clase UML, paquetes colapsados).

Identificación de las tareas iniciales

Los puntos iniciales para la identificación de tareas son el enunciado del problema (incluyendo los productos a entregar y las fechas de entrega), el ciclo de vida de las actividades y la experiencia pasada. El cliente y el gerente del proyecto acuerdan una serie de revisiones cuyo propósito es revisar el avance del proyecto. Cada producto a entregar debe ser cubierto por una tarea, al menos.

La actividad de planeación identifica productos de trabajo adicionales para el consumo propio del proyecto. Éstos incluyen productos de trabajo administrativos, como el SPMP, productos de trabajo para el desarrollo, como las definiciones de interfaz preliminares, prototipos rápidos y evaluaciones, como la hojas de resumen de herramientas, métodos o procedimientos.

El trabajo se descompone al principio en tareas que son lo bastante pequeñas para que se estimen y asignen a una sola persona. A esta descomposición inicial se le llama con frecuencia estructura de división del trabajo (WBS, por sus siglas en inglés) y se parece a una gran lista de pendientes. Por lo general, entre más específicos sean los requerimientos y los productos a entregar será más fácil definir tareas precisas. Cuando los requerimientos no están bien definidos el gerente debe asignar tiempo adicional para la replaneación y revisar el plan de tareas conforme se estabilizan los requerimientos.

Por ejemplo, consideremos las actividades de diseño del equipo que desarrolla el subsistema `AdministraciónBaseDatos` del OWL. A partir de los requerimientos el equipo sabe que el subsistema `AdministraciónBaseDatos` debe mantener con confiabilidad el estado de los sensores y las acciones enviadas a los actuadores. Además, el subsistema `AdministraciónBaseDatos` debe soportar búsquedas de datos históricos para el análisis de tendencias. La tabla 11-3 muestra un ejemplo de una estructura de división del trabajo inicial para un sistema de éstos. Los tiempos estimados se basan en experiencias anteriores del gerente o del estimador.

La estructura de la división del trabajo inicial no debe ser demasiado detallada. Muchos gerentes de proyecto tienen una tendencia a crear listas de pendientes detalladas al inicio del proyecto. Aunque es deseable un modelo detallado del trabajo, es difícil crear al principio una estructura de división del trabajo detallada significativa. El refinamiento de la definición del sistema durante el análisis, y del diseño de la arquitectura de software durante el diseño del sistema genera muchos cambios en la estructura de la división del trabajo original, incluyendo nuevas tareas para investigar tecnología reciente, nuevos subsistemas y funcionalidad adicional. La creación de una estructura de división del trabajo detallada sólo da lugar a trabajo adicional cuando se revisa el plan. En vez de ello, un gerente detalla la estructura de la división del trabajo sólo a corto plazo, y describe las tareas a largo plazo con menor detalle. El gerente revisa y detalla la estructura de la división del trabajo sólo conforme avanza el proyecto.

Identificación de las dependencias de tareas

Una vez que se han identificado las tareas necesitamos encontrar las dependencias entre ellas. La identificación de dependencias nos permite asignar con más eficiencia los recursos a las tareas. Por ejemplo, dos tareas independientes pueden asignarse a diferentes participantes para que las realicen en paralelo. Encontramos dependencias entre tareas examinando los productos de trabajo que requiere cada tarea. Por ejemplo, una tarea de prueba de unidad requiere la especificación de la interfaz de programador de la unidad, generada por la tarea de diseño del sistema. La tarea de diseño del sistema debe terminarse antes de que pueda iniciarse la tarea de prueba. La figura 11-16 muestra las dependencias entre las tareas identificadas en la tabla 11-3. El conjunto de tareas y sus dependencias constituye el modelo de tareas. El siguiente paso es establecer la correspondencia entre el modelo de tareas y los recursos y el tiempo para crear una calendarización.

Creación de la calendarización inicial

La calendarización inicial se crea estableciendo la correspondencia entre el modelo de tareas y el tiempo y los recursos. Las dependencias entre las tareas, la duración estimada de las mismas y

Tabla 11-3 Un ejemplo de la estructura de división del trabajo inicial para `AdministraciónBaseDatos`.

Tarea	Tiempo estimado
1. Selección del sistema de administración de base de datos	2 semanas
2. Identificación de los objetos persistentes y sus atributos	1 semana
3. Identificación de consultas	1 semana
4. Identificación de atributos buscables	1 día
5. Definición del esquema	2 semanas
6. Construcción del prototipo para la evaluación del desempeño	2 semanas
7. Definición de la API para los demás subsistemas	3 días
8. Identificación de requerimientos de concurrencia	2 días
9. Implementación del subsistema de base de datos	2 semanas
10. Prueba unitaria de la base de datos	3 semanas
11. Resolución de los peligros de concurrencia restantes	2 semanas

la cantidad de participantes se usan para crear la calendarización inicial. La calendarización inicial se usa para planear las fechas de las interacciones primarias con el cliente y los usuarios, incluyendo las entrevistas con los usuarios, las revisiones del cliente y las entregas. Estas fechas se hacen parte del enunciado del problema y representan fechas de entrega acordadas mutuamente entre el cliente y el gerente del proyecto. Estas fechas se planean de tal forma que, aunque cambien, todavía satisfagan los tiempos de entrega. La figura 11-17 muestra la calendarización inicial para el OWL.

Así como sucede con el modelo de tareas, no es realista crear una calendarización detallada al inicio del proyecto. La calendarización inicial debe incluir fechas de entrega para todos los productos que se entregan al cliente y una calendarización detallada a corto plazo (por ejemplo, los dos primeros meses). La calendarización detallada para el resto del proyecto se realiza conforme avanza el mismo. Además, una vez que está en marcha el proyecto cada equipo puede contribuir a la planeación de sus tareas, tomando en cuenta que tiene una visión más detallada del trabajo a realizar. La calendarización general se realiza como parte de un compromiso continuo entre recursos, tiempo y funcionalidad implementada, y se actualiza en forma constante.

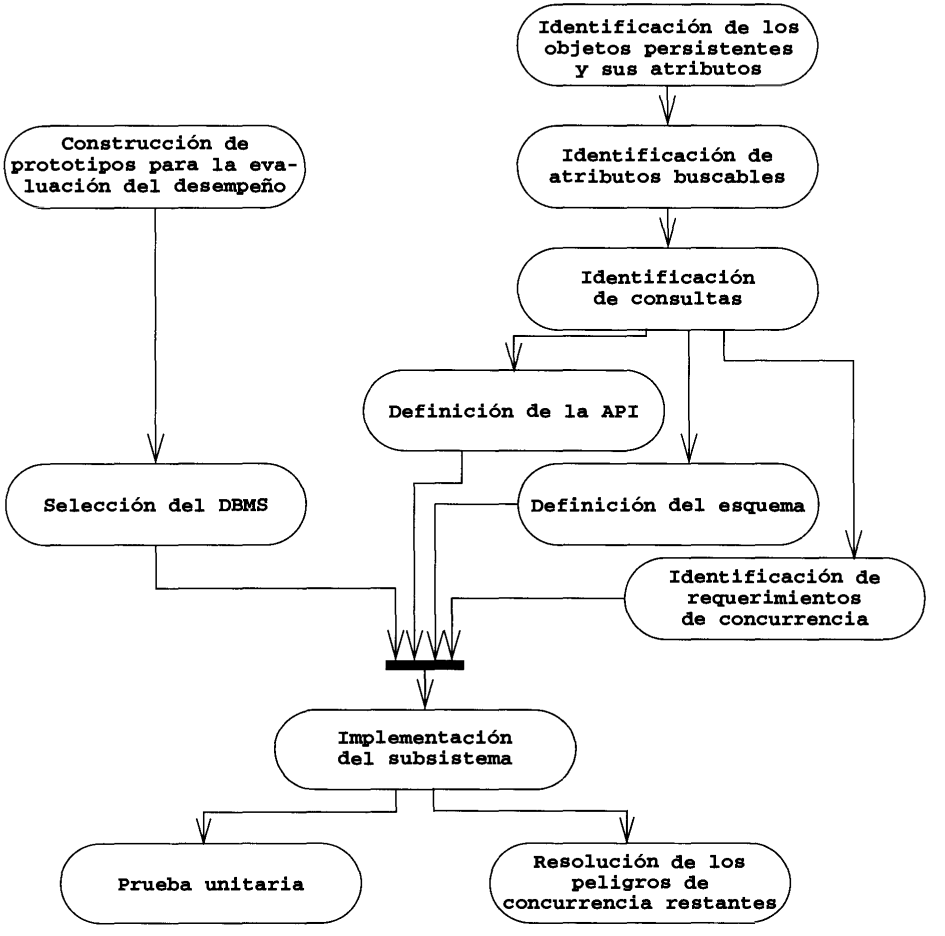


Figura 11-16 Modelo de tareas para la estructura de división del trabajo de la tabla 11-3.

Preparación de los equipos

El siguiente paso en el inicio del proyecto es armar los equipos que elaborarán los productos a entregar. Pueden asignarse todos los desarrolladores que trabajarán en el proyecto de una vez (contratación llana) o el proyecto puede crecer en forma gradual contratando personal conforme se necesita (contratación gradual). Por un lado, la contratación gradual está motivada por el ahorro de recursos en la primera parte del proyecto. La obtención de requerimientos y el análisis no requieren tantas personas como la codificación y las pruebas. Además, analista e implementador son papeles que requieren diferentes habilidades y, por lo tanto, deben asignarse a personas diferentes. Por otro lado, la contratación llana tiene la ventaja del pronto establecimiento de los equipos y del ambiente social necesario para la comunicación espontánea. Algunos de los desarrolladores pueden asignarse a las actividades de análisis con los analistas, mientras que los demás pueden iniciar otras activi-

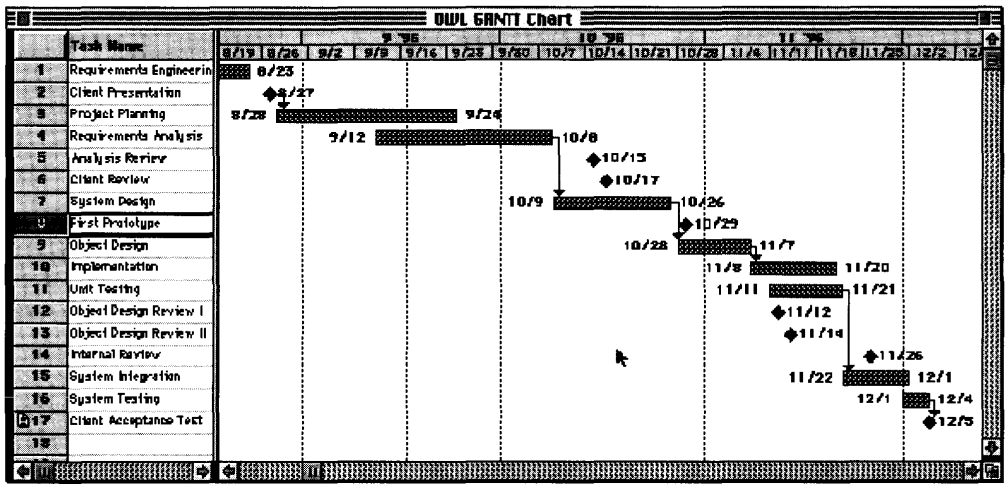


Figura 11-17 Calendarización del proyecto inicial para el OWL (gráfica de Gantt). La calendarización inicial sirve para estimar las fechas de entrega y la interacción con el cliente. La calendarización se detalla y revisa conforme avanza el proyecto.

dades, como la especificación del ambiente de administración de la configuración, investigaciones tecnológicas y evaluación y entrenamiento. Con proyectos y fechas de entrega a mercado más cortos, la contratación llana está siendo el esquema de contratación preferido. El dilema entre la contratación gradual y la llana lo ha tratado ampliamente Brooks [Brooks, 1975].

El gerente del proyecto asigna personal a los equipos examinando el diseño de alto nivel y considerando las habilidades requeridas para cada subsistema. A menudo no se tienen disponibles los suficientes desarrolladores con las habilidades necesarias, y en ese caso el gerente del proyecto incluye tiempo en el modelo de tareas para el entrenamiento técnico del nuevo personal. El gerente del proyecto distribuye al nuevo personal por todo el proyecto en forma tal que cada equipo tenga, al menos, un desarrollador experimentado. El desarrollador experimentado proporciona el liderazgo técnico y sirve como mentor para los nuevos desarrolladores quienes obtienen experiencia más rápido cuando ponen en práctica su nuevo entrenamiento. El entrenamiento formal y el aprendizaje por la práctica es un compromiso que evalúa el gerente del proyecto tomando en cuenta las restricciones de tiempo del proyecto. El entrenamiento formal permite que los desarrolladores lleguen más pronto a ser más productivos, pero es caro y constituye una inversión a más largo plazo que beneficia a los proyectos futuros.

El gerente del proyecto selecciona a los líderes de equipo antes de que se formen los equipos. Además de poder comprender el estado del equipo, el líder del mismo necesita poder comunicarse de manera efectiva, reconocer crisis pendientes (sociales o técnicas) y resolver compromisos tomando en cuenta los asuntos en el nivel de proyecto. El inicio del proyecto también es un momento ideal para el entrenamiento de los líderes de equipo, para que se familiaricen con los procedimientos y las reglas básicas del proyecto. El papel de líder de equipo es distinto al de líder técnico. El líder técnico, por lo general el enlace con el equipo de arquitectura, interactúa

con el arquitecto en jefe y los enlaces de arquitectura de los demás equipos y tiene la última palabra en las decisiones técnicas dentro del equipo. El papel de enlace de arquitectura requiere habilidades técnicas excelentes y experiencia en el desarrollo.

En un contexto de entrenamiento, el papel de líder del equipo puede dividirse en dos. Un entrenador instruye, explica y da consejos al equipo. El entrenador también actúa como enlace con la administración. Un aprendiz de líder de equipo desempeña todas las demás responsabilidades del líder de equipo, incluyendo la organización de reuniones de estado y la asignación de tareas. En un proyecto de curso, este papel lo desempeña el instructor o el asistente de enseñanza, mientras que el enlace con la arquitectura lo ejerce un estudiante. En un proyecto de desarrollo, funge como líder del equipo un desarrollador que tiene experiencia y conocimiento de los procedimientos administrativos de la organización, mientras que el enlace con la arquitectura lo desempeña un desarrollador que tenga grandes habilidades técnicas.

La tabla 11-4 muestra un ejemplo de la asignación de papeles a los participantes en el equipo de base de datos del proyecto OWL. El gerente del proyecto y los líderes de equipo asignan los papeles con base en las habilidades y determinan las necesidades de entrenamiento de cada uno de los participantes.

Tabla 11-4 Asignación de papeles, habilidades y necesidades de entrenamiento del equipo de base de datos de OWL.

Participante	Papeles	Habilidades	Necesidades de entrenamiento
Alicia	Líder de equipo	Administración: Líder de equipo Programación: C Administración de la configuración	UML Habilidades de comunicación
Juan	Enlace de arquitectura Implementador	Programación: C++ Modelado: UML	Java
María	Gerente de configuración Implementadora	Programación: C++, Java Modelado: Relación de entidades Bases de datos: relacional Administración de la configuración	Bases de datos orientadas a objetos Modelado UML
Cristina	Implementadora	Programación: C++, Java Modelado: Relación de entidades Bases de datos: orientadas a objetos	Modelado UML
Samuel	Enlace de administración de instalaciones Probador	Programación: C++ Pruebas: caja blanca, caja negra	Inspecciones Java

Organización de la infraestructura de comunicación

La comunicación clara y precisa es esencial para el éxito de un proyecto de desarrollo. Además, la comunicación llega a ser crucial cuando se incrementa la cantidad de participantes en el proyecto. En consecuencia, la organización de la infraestructura de comunicación de un proyecto se hace lo más pronto posible; esto es, durante el inicio del proyecto. El gerente del proyecto necesita resolver los siguientes modos de comunicación, como se describió en el capítulo 3, *Comunicación de proyectos*:

- *Modos de comunicación calendarizados.* Éstos incluyen los indicadores de avance planeados, como las revisiones del cliente y del proyecto, las reuniones de estado del equipo, las inspecciones, etc. Los modos de comunicación calendarizados son los medios formales mediante los cuales recopilan y comparten información los participantes en el proyecto. Estos modos tienen mejor soporte mediante la comunicación síncrona o frente a frente, como las reuniones, las presentaciones formales, las videoconferencias y las conferencias telefónicas. El gerente calendariza las fechas y designa a los organizadores de cada una de estas interacciones.
- *Modos de comunicación basados en eventos.* Éstos incluyen los reportes de problemas, las peticiones de cambio, la discusión de asuntos y las resoluciones. Los modos basados en eventos se presentan, por lo general, a partir de problemas y crisis no previstos. Los mecanismos asíncronos, como el correo electrónico, el groupware y las bases de datos de problemas necesitan organizarse desde el principio, y es necesario entrenar a los participantes para que los usen. Cuando la cantidad de participantes es grande son preferibles las infraestructuras centralizadas, como los sitios Web y los tableros de noticias, ya que ponen más información a la vista de más personas que el correo electrónico o las conversaciones bilaterales. Por ejemplo, la figura 11-18 es la página de inicio del proyecto OWL, y proporciona acceso al directorio del proyecto, a los tableros de noticias, a los documentos y al código fuente. Esta infraestructura también tiene la ventaja de ser accesible con requerimientos mínimos de software y plataforma.

El gerente del proyecto usa mecanismos de comunicación diferentes con el cliente y los usuarios finales de los que usa con los desarrolladores. Los desarrolladores, por un lado, y los clientes y usuarios finales, por el otro, tienen terminología, necesidades de información y disponibilidad diferentes. El cliente y los usuarios finales hablan el lenguaje del dominio de aplicación y puede ser que no tengan muchos conocimientos de ingeniería de software. Necesitan información acerca del estado del proyecto y del modelo de análisis. Su involucramiento con el proyecto es episódico y se enfoca en su propio trabajo. Los desarrolladores el lenguaje del dominio de solución, y puede ser que no tengan mucho conocimiento del dominio de aplicación. Su necesidad de información es técnica y diaria. Una buena heurística es realizar la comunicación frente a frente varias veces durante el proyecto y establecer un tablero de noticias para los clientes para las necesidades de comunicación no calendarizadas (por ejemplo, para la aclaración de requerimientos). Esto asume que el cliente está familiarizado con el uso de los tableros de noticias y la comunicación en línea. Si no es así, se usa entrenamiento u otros mecanismos de comunicación. Sin embargo, tome en cuenta que el canal de comunicación entre el cliente y los usuarios tiene un ancho de banda pequeño y es necesario usarlo en forma eficiente.

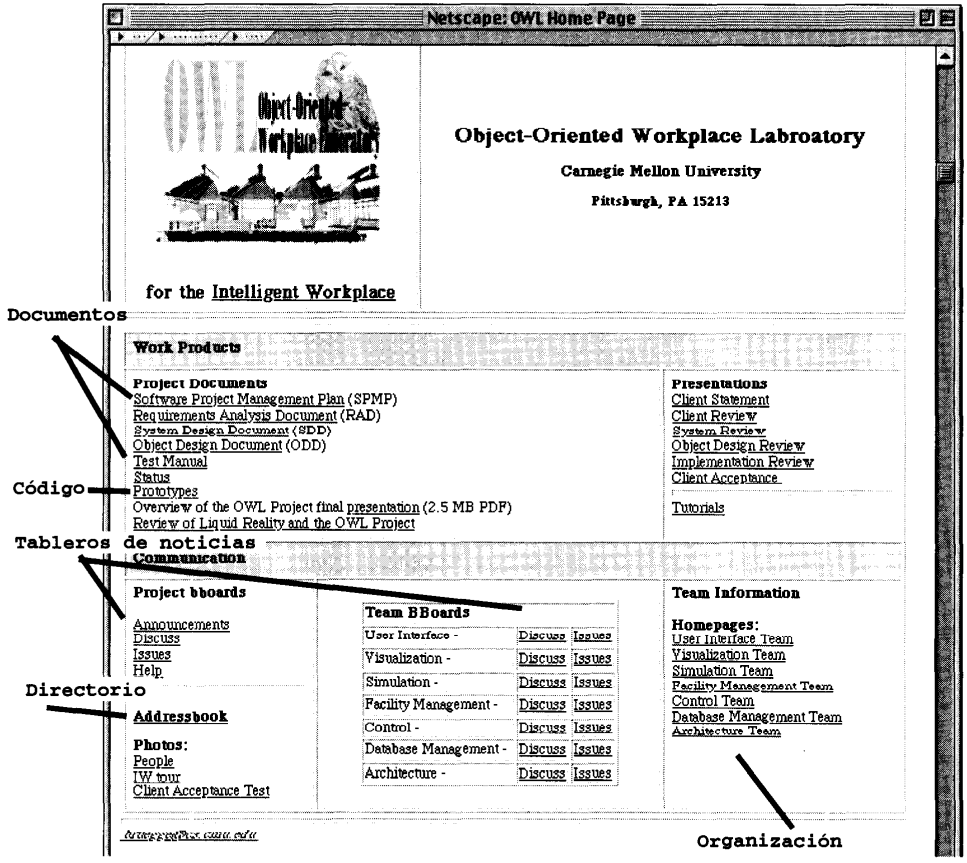


Figura 11-18 Página de inicio del OWL. Los desarrolladores tienen acceso a toda la información relevante desde una página Web central.

Arranque del proyecto

La reunión de arranque del proyecto marca el final de la fase de inicio del proyecto y el principio de la fase de desarrollo. El gerente del proyecto ha terminado gran parte de la planeación inicial y el diseño de alto nivel, ha seleccionado una organización, ha contratado desarrolladores y ha designado a los líderes de equipo. El arranque del proyecto consta de tres reuniones.

- *La presentación al cliente.* En esta reunión están incluidos todos los participantes en el proyecto. Incluye una presentación, por parte del cliente, de los requerimientos del sistema, y una presentación, por parte del gerente, de la organización y calendarización iniciales.
- *La reunión de arranque de la administración.* El gerente del proyecto organiza esta reunión que incluye a todos los líderes de equipo. La reunión sirve para definir los procedimientos administrativos, como los procedimientos de reunión, la administración de la configuración y el reporte de problemas, y dar entrenamiento sobre ellos.

- *Reuniones de arranque de equipos individuales.* Estas reuniones las organizan los líderes de equipo e incluyen a los miembros del equipo. A los participantes se les comunican los procedimientos definidos durante la reunión de administración. Se presentan entre sí los miembros del equipo. Aquí es donde comienza el involucramiento del líder del equipo. La figura 11-19 es un ejemplo de una agenda para la reunión de arranque del equipo de base de datos.

El objetivo de las tres reuniones de arranque es que los participantes se conozcan y que se inicie la comunicación. Los participantes se familiarizan con los procedimientos de reunión básicos, la organización y la asignación de papeles y los mecanismos generales del proyecto. Poco después se distribuyen las tareas y se realizan las reuniones de estado normales. Después de unas cuantas reuniones de estado los aspectos calendarizados de la comunicación ya deben estar bien cimentados.

Luego describimos las actividades de supervisión del proyecto durante el estado estable. Las actividades de administración durante el estado estable se enfocan sobre todo en eventos inesperados y planes de contingencia.

11.4.3 Supervisión del proyecto

Para tomar decisiones efectivas, el gerente del proyecto necesita información de estado precisa. Por desgraciada, es difícil la recopilación del estado preciso. Es difícil comprender un sistema complejo en sí, y es todavía más difícil comprender el estado de sus componentes y su impacto sobre los productos entregables futuros. Los desarrolladores no reportarán a su líder de equipo cualquier problema que consideren que puedan resolver a tiempo. Sin embargo, las pequeñas desviaciones con respecto al calendario, que no vale la pena reportar en forma individual, se acumulan y degeneran en grandes desviaciones mucho más adelante en el calendario. Para cuando los líderes de equipo descubren y reportan un problema grande al gerente del proyecto, dicho problema ya ha causado un retraso significativo en el proyecto.

Se dispone de varias herramientas para recopilar información de estado. Debido a que ninguna es precisa o confiable por sí sola, los líderes de equipo y los gerentes de proyecto necesitan usar una combinación de ellas. A continuación revisamos sus ventajas y desventajas.

Reuniones

- *Reuniones de estado periódicas.* Las reuniones de estado realizadas en el nivel de equipo tienen el mejor potencial para reportar el estado y la información requerida para tomar decisiones correctivas. Sin embargo, las reuniones de estado también pueden proporcionar información de estado poco precisa si los miembros del equipo no cooperan. Los participantes son renuentes por naturaleza para reportar problemas o errores. Cooperan sólo si pueden confiar en que un gerente no se meterá en los problemas que pueden resolver por sí solos. Los gerentes necesitan lograr que el reporte de problemas sea benéfico para los desarrolladores al intervenir sólo cuando es necesario.
- *Indicadores de avance precisos.* El avance puede medirse determinando si los desarrolladores entregan sus productos de trabajo a tiempo. Los gerentes pueden incrementar la precisión de este método definiendo indicadores de avance precisos de tal forma que puedan supervisarse con precisión. Por ejemplo, el indicador de avance "código terminado" no es preciso, debido a que no toma en cuenta la calidad del código entregado. El código puede

AGENDA: Reunión de arranque del equipo de base de datos

Cuándo y dónde		**Papel**
Fecha:	13/11/1998	**Moderador principal:** María
Inicio:	4:30 p.m.	**Tomador de tiempo:** Juan
Terminación:	5:30 p.m.	**Secretario de actas:** Cristina
Edificio: AC Hall		**Salón:** 3421

1. Propósito
Familiarizarse con los papeles de administración del proyecto para un proyecto de mediana escala con una jerarquía de dos niveles.

2. Resultado deseado
- Los papeles del grupo se asignan a las personas
- Se terminan los tiempos de reunión
- Primer conjunto de conceptos de acción para la siguiente reunión

3. Compartir la información [tiempo asignado: 25 minutos]
Procedimientos de reunión
- Lineamientos
- Agendas/minutas
- Papeles

Papeles de equipo
- Líder de equipo
- Enlace de arquitectura
- Enlace de documentación
- Gerente de configuración
- Encargado de herramientas

Tareas del equipo
Calendarización del equipo
Reporte de problemas

4. Discusión [tiempo asignado: 25 minutos]
Definición de las reglas básicas
Asignación inicial de papeles y tareas

5. Cierre [tiempo asignado: 5 minutos]
Revisión y asignación de nuevos conceptos de acción
Crítica de la reunión

Figura 11-19 Un ejemplo de agenda para la reunión de arranque del equipo de base de datos del proyecto OWL.

contener pocos o muchos errores, y el indicador de avance se consideraría terminado en ambos casos. En forma alterna, la definición de un indicador de avance como la terminación de la prueba, la demostración, la documentación y la integración de una característica específica produce una mejor información de estado. Cuando se definen y supervisan indicadores de avance precisos, los gerentes no necesitan la cooperación de los desarrolladores.

- *Revisiones del proyecto.* Una revisión del proyecto es un medio para que todos los participantes intercambien información de estado y del plan acerca de todos los equipos con una presentación formal. Las revisiones del proyecto sufren problemas similares a los de las reuniones de estado: si los participantes ya han sido renuentes a admitir problemas enfrente de su líder de equipo, definitivamente no admitirán problemas en un foro público.
- *Inspecciones del código.* Las revisiones formales del código entre iguales son un método efectivo para el descubrimiento temprano de defectos. Cuando se realizan con frecuencia sus resultados también pueden usarse como un indicador de avance.
- *Demostraciones de prototipos.* Las demostraciones de prototipos son implementaciones parciales del sistema para evaluar la tecnología o la funcionalidad. Por lo general, no ilustran cómo se manejan los casos de frontera o qué tan robusta es la implementación. Son buenas para medir el avance inicial, pero no son mediciones precisas de terminación.

Mediciones

- *Defectos a resolver.* Una vez que existe una versión inicial del sistema, la cantidad de defectos a resolver es una medida intuitiva de qué tanto esfuerzo hay que realizar antes de que se pueda entregar el sistema. Sin embargo, la medición de la cantidad de defectos pendientes motiva su rápida resolución y activa la introducción de nuevos defectos, invalidando el propósito original de medir la cantidad de defectos.
- *Medición del código fuente.* Para la estimación se han propuesto diversas mediciones del código fuente, que van desde la cantidad de líneas de código hasta relaciones de la cantidad de operadores y operandos y puntos de función. Estas mediciones son sensibles a muchos factores del ambiente, incluyendo el estilo de codificación de los implementadores. Además, los resultados no parecen ser transferibles de una organización a otra. Estas medidas sólo deberán usarse si se ha organizado un esfuerzo sistemático a lo largo de varios proyectos dentro de la organización.

Por último, independientemente de los métodos que se usen para determinar el estado, el gerente del proyecto, los líderes de equipo y los desarrolladores necesitan comunicar la información de estado en términos comprensibles. Luego describimos la administración del riesgo como un método para controlar los proyectos, comunicar problemas potenciales y planear contingencias.

11.4.4 Administración del riesgo

El enfoque de la administración del riesgo es identificar posibles problemas en el proyecto y resolverlos antes de que puedan tener un impacto significativo en la fecha de entrega o el presupuesto. El elemento principal de la administración del riesgo es el ajuste de los flujos de información adecuados, de tal forma que se reporten con precisión y a tiempo los riesgos y problemas. Muchos proyectos fallan debido a que se reportaron muy tarde problemas simples o porque se resolvió el problema equivocado. En esta sección nos enfocamos en las actividades de la administración del riesgo para la identificación, el análisis y la resolución de riesgos. Para mayores

detalles sobre la administración del riesgo en la ingeniería de software, remitimos al lector a la literatura especializada, por ejemplo [Boehm, 1991] y [Charette, 1989].

Identificación de los riesgos

El primer paso de la administración del riesgo es identificar los riesgos. Los riesgos son problemas potenciales que resultan de un área de incertidumbre. Los riesgos pueden ser administrativos o técnicos. Los riesgos administrativos incluyen cualquier incertidumbre relacionada con la organización, los productos de trabajo, los papeles o el plan de tareas. Los riesgos técnicos incluyen cualquier incertidumbre relacionada con los modelos del sistema, incluyendo los cambios de la funcionalidad del sistema, los requerimientos no funcionales, la arquitectura o la implementación del sistema. En particular, los riesgos técnicos incluyen los defectos descubiertos tardíamente en el desarrollo. La tabla 11-5 muestra ejemplos de riesgos en el proyecto OWL.

Tabla 11-5 Ejemplos de identificación de riesgos en el proyecto OWL.

Riesgo	Tipo de riesgo
El módulo de ambiente personal (PEM) del fabricante no se apega al estándar publicado	Técnico
La fecha de entrega del PEM del fabricante rebasa lo planeado	Administrativo
Los usuarios no desean usar el navegador Web para ajustar la temperatura	Técnico
El middleware seleccionado es demasiado lento para satisfacer los requerimientos de desempeño para el registro de datos	Técnico
El desarrollo de la capa del PEM abstracto se lleva más tiempo que el calendarizado	Administrativo

A menudo los desarrolladores están conscientes de los riesgos que corresponden a sus tareas. El reto es motivar a los desarrolladores para que reporten los riesgos a fin de que puedan aflorar y se administren. Esto implica recompensar a los desarrolladores para que reporten los riesgos y problemas, y que hagan que las actividades de administración del riesgo sean benéficas en forma directa para los desarrolladores. Por lo general, el reporte espontáneo del riesgo no es suficiente: los participantes en el proyecto, incluyendo al cliente, a los desarrolladores y a los gerentes, se resisten, por lo general, a comunicar fallas o limitaciones potenciales y son demasiado optimistas acerca de sus resultados. Un enfoque más sistemático para la identificación del riesgo es entrevistar a los participantes del proyecto usando un cuestionario estructurado: a los participantes se les pide en sesiones de grupo que listen los riesgos que anticipan con respecto a tareas específicas. La figura 11-20 muestra preguntas de ejemplo del proceso de identificación de riesgos basado en taxonomía del SEI [Carr *et al.*, 1993]. En este ejemplo el entrevistador trata de determinar si hay algún riesgo relacionado con los requerimientos no funcionales del desempeño. Dependiendo de la respuesta a la primera pregunta, el entrevistador puede hacer preguntas de seguimiento para asegurarse que no estén presentes otros riesgos relacionados. El fundamento que hay tras este cuestionario es cubrir todas las áreas del desarrollo en donde se encuentran los riesgos en forma típica.

1. Requerimientos

...

d. Desempeño

...

[23] ¿Se ha realizado un análisis de desempeño?

(Sí) (23.a) ¿Cuál es su nivel de confianza en el análisis de desempeño?

(Sí) (23.b) ¿Tiene un modelo para el seguimiento del desempeño a lo largo del diseño y la implementación?

Figura 11-20 Preguntas para obtener los riesgos del desempeño en el proceso de identificación de riesgos basado en taxonomía. Si la respuesta a la pregunta 23 es positiva, se hacen las preguntas 23.a y 23.b a los desarrolladores [Carr *et al.*, 1993].

Priorización de los riesgos

La identificación sistemática de los riesgos produce gran cantidad de riesgos administrativos y técnicos, algunos críticos y otros sin importancia. La priorización de los riesgos permite que los gerentes se enfoquen sólo en los riesgos críticos. Los riesgos se caracterizan por la probabilidad de que puedan convertirse en problemas y por su impacto potencial al proyecto cuando se convierten en problemas. Usando estos dos atributos, los riesgos pueden asignarse a una de cuatro categorías:

- Probable, alto impacto potencial
- Poco probable, alto impacto potencial
- Probable, bajo impacto potencial
- Poco probable, bajo impacto potencial

Los riesgos de la primera categoría son los que deben preocupar a los gerentes (probable, alto impacto potencial). Para estos riesgos, los desarrolladores y gerentes deben elaborar planes de contingencia y supervisar el riesgo en forma meticulosa. Si se incrementa la probabilidad del riesgo, los gerentes pueden activar el plan de contingencia y resolver el problema a tiempo. Además, los gerentes deben supervisar los riesgos de la segunda categoría (poco probable, alto impacto potencial). No es necesario elaborar planes de contingencia para ellos a menos que se incremente su probabilidad. Por último, los riesgos de la tercera y cuarta categorías pueden

ignorarse, a menos que se disponga de recursos suficientes para supervisarlos. La tabla 11-6 es el ordenamiento de los riesgos presentados en la tabla 11-5.

Tabla 11-6 Riesgos priorizados del proyecto OWL.

Riesgo	Probabilidad	Impacto potencial
El módulo de ambiente personal (PEM) del fabricante no se apega al estándar publicado	Poco probable	Alto
La fecha de entrega del PEM del fabricante rebasa lo planeado	Poco probable	Alto
Los usuarios no desean usar el navegador Web para ajustar la temperatura	Probable para los usuarios que pasan menos de dos horas al día frente a una computadora	Alto
El middleware seleccionado es demasiado lento para satisfacer los requerimientos de desempeño para el registro de datos	Poco probable, baja frecuencia de muestreo	Alto
El desarrollo de la capa del PEM abstracto se lleva más tiempo que el calendarizado	Probable, primera vez que sucede	Alto

Mitigación del riesgo

Una vez que se han identificado y priorizado los riesgos es necesario diseñar estrategias de mitigación para los riesgos críticos. Las estrategias de mitigación pueden incluir la disminución de la probabilidad del riesgo o la disminución de su impacto potencial. Los riesgos son causados, en general, por un área de incertidumbre, como información faltante o falta de confianza en alguna información. Los desarrolladores disminuyen la probabilidad de un riesgo investigando más las causas del riesgo, cambiando proveedores o componentes, o seleccionando un proveedor o componente redundante. Del mismo modo, los desarrolladores disminuyen el impacto del riesgo en el proyecto desarrollando una solución alterna o redundante. Sin embargo, en la mayoría de los casos la mitigación del riesgo hace que se incurra en recursos y costos adicionales. La tabla 11-7 describe las estrategias de mitigación para los riesgos presentados en la tabla 11-5.

Tabla 11-7 Estrategias de mitigación para riesgos OWL.

Riesgo	Estrategia de mitigación
El módulo de ambiente personal (PEM) del fabricante no se apega al estándar publicado	• Ejecutar pruebas de desempeño para identificar dónde no se apega • Investigar si se pueden evitar las funciones que no se apegan
La fecha de entrega del PEM del fabricante rebasa lo planeado	• Supervisar el riesgo pidiendo al fabricante reportes de estado intermedios
Los usuarios no desean usar el navegador Web para ajustar la temperatura	• Realizar estudios de usabilidad usando maquetas • Desarrollar una interfaz alterna
El middleware seleccionado es demasiado lento para satisfacer los requerimientos de desempeño para el registro de datos	• Supervisar el riesgo. Planear un prototipo para la evaluación del desempeño
El desarrollo de la capa del PEM abstracto se lleva más tiempo que el calendarizado	• Incrementar la prioridad de esta tarea con respecto a las demás tareas de implementación • Asignar desarrolladores principales a esta tarea

Comunicación de los riesgos

Una vez que se han identificado, priorizado y mitigado los riesgos, es necesario comunicar el plan de administración de riesgos a todos los interesados. La administración del riesgo se apoya en una comunicación a tiempo. Deben motivarse los reportes espontáneos y a tiempo de los problemas potenciales. Los planes de administración de riesgos y los documentos técnicos se comunican usando los mismos canales. Los desarrolladores y demás participantes en el proyecto revisan los riesgos al mismo tiempo que revisan el contenido técnico del proyecto. Como se dijo antes, la comunicación es la barrera más significativa cuando se maneja la incertidumbre.

11.4.5 Acuerdo del proyecto

El *Acuerdo del proyecto* es un documento que define de manera formal el alcance, la duración, el costo y los productos a entregar del proyecto. La forma del *Acuerdo del proyecto* puede ser un contrato, una declaración de trabajo, un plan de negocios o una carta de proyecto. El *Acuerdo del proyecto* se termina, por lo general, poco después de que se estabiliza el modelo de análisis y está en marcha la planeación del resto del proyecto.

Un *Acuerdo del proyecto* debe contener, al menos, lo siguiente:

- Una lista de los documentos a entregar.
- Los criterios para la demostración de los requerimientos funcionales.

- Los criterios para la demostración de los requerimientos no funcionales, incluyendo precisión, confiabilidad, tiempo de respuesta y seguridad.
- Los criterios para la aceptación.

El *Acuerdo del proyecto* representa la línea base de las pruebas para la aceptación del cliente. Cualquier cambio en la funcionalidad a entregar, los requerimientos no funcionales, los tiempos de entrega o el presupuesto del proyecto requiere la renegociación del *Acuerdo del proyecto.*

11.4.6 Prueba de aceptación del cliente

El propósito de la prueba de aceptación del cliente es la presentación del sistema y la aprobación del cliente de acuerdo a los criterios de aceptación establecidos en el *Acuerdo del proyecto.* El resultado de la prueba de aceptación del cliente es la aceptación formal (o el rechazo) del sistema por parte del cliente. Puede ser que la instalación del sistema y las pruebas de campo hechas por el cliente ya hayan sucedido antes de esta prueba. La prueba de aceptación del cliente constituye el final visible del proyecto.

La prueba de aceptación del cliente se realiza como una serie de presentaciones de la funcionalidad y características novedosas del sistema. Se ejercitan los escenarios importantes del *Enunciado del problema* y los demuestran los desarrolladores o los futuros usuarios. Las demostraciones adicionales se enfocan en los requerimientos no funcionales del sistema, como la precisión, la confiabilidad, el tiempo de respuesta o la seguridad. Si la instalación y las evaluaciones del usuario sucedieron antes de la prueba de aceptación del cliente, se presentan y resumen los resultados. Por último, la prueba de aceptación del cliente también sirve como un foro de discusión para actividades subsiguientes, como el mantenimiento, la transferencia de conocimientos o la mejora del sistema.

11.4.7 Instalación

La fase de instalación del proyecto incluye la prueba de campo del sistema, la comparación de los resultados del sistema con el sistema heredado, la eliminación de sistema heredado y el entrenamiento de los usuarios. El proveedor o el cliente pueden realizar instalación, dependiendo del *Acuerdo del proyecto.*

Para minimizar los riesgos, la instalación y la prueba de campo se realizan incrementalmente, usando sitios no críticos como ambiente de pruebas de campo. Sólo hasta que el cliente está convencido que la interrupción de su negocio se mantendrá al mínimo es cuando el sistema entregado entra en operación a escala completa. Los sistemas de reemplazo y las mejoras rara vez se introducen en forma de "gran explosión", ya que la mayor cantidad de problemas se descubre durante los primeros días de operación.

11.4.8 Post mortem

Todo proyecto descubre nuevos problemas, eventos no previstos y fallas inesperadas. Por tanto, cada proyecto constituye una oportunidad para el aprendizaje y la anticipación de nuevos riesgos. Muchas compañías de software realizan un estudio post mortem de cada proyecto después de que termina. El post mortem incluye la recopilación de datos acerca de las fechas de entrega planeadas al inicio y las reales, la cantidad de defectos descubiertos, información cualitativa acerca

de problemas técnicos y administrativos descubiertos y sugerencias para proyectos futuros. Aunque esta fase es la menos visible del ciclo de vida, la compañía depende mucho de ella para el aprendizaje y mejora de su eficiencia.

11.5 Administración de los modelos y actividades de la administración del proyecto

Las actividades de administración del proyecto, así como las actividades técnicas, necesitan documentarse, asignarse, comunicarse y revisarse cuando se presentan eventos inesperados. En esta sección describimos los problemas administrativos que se aplican a las actividades de administración del proyecto, incluyendo.

- La documentación de los modelos de administración (sección 11.5.1).
- La asignación de responsabilidades para controlar y supervisar el proyecto (sección 11.5.2).
- La comunicación acerca de la administración del proyecto (sección 11.5.3).

11.5.1 Documentación de la administración del proyecto

Los modelos de administración descritos en la sección 11.3 se documentan en el SPMP [IEEE Std. 1058.1-1993]. La audiencia del SPMP incluye a la gerencia y a los desarrolladores. El SPMP documenta todos los asuntos relacionados con los requerimientos del cliente (como los productos a entregar y los criterios de aceptación), los objetivos del proyecto, la organización del proyecto, la división del trabajo en tareas y la asignación de recursos y responsabilidades. La figura 11-21 muestra un ejemplo de un esquema para el SPMP.

La primera sección del SPMP, *Introducción*, proporciona información de fondo para el resto del documento. Describe en forma breve el proyecto, los productos a entregar al cliente, los indicadores de avance del proyecto y los cambios que se espera que sufran los documentos. Esta sección contiene las restricciones fuertes que se encuentran en el *Acuerdo del proyecto* y que son relevantes para los desarrolladores.

La segunda sección del SPMP describe la *Organización del proyecto*. Se describe el diseño de alto nivel del sistema, junto con los equipos de subsistemas y de funcionalidad cruzada que conforman el proyecto. Se definen las fronteras de cada equipo y de la administración y se asignan responsabilidades. En esta sección se describen los papeles de comunicación, como los enlaces. Leyendo esta sección los desarrolladores pueden identificar a los participantes en otros equipos con los que necesitan comunicarse.

La tercera sección del SPMP, *Proceso administrativo*, describe la manera en que la administración supervisa el estado del proyecto y como ataca los problemas no previstos. Se describen las dependencias entre equipos y subsistemas, y se hacen públicos los riesgos anticipados y planes de contingencia. Al documentar lo que puede salir mal se facilita que los desarrolladores lo reporten cuando sucedan los problemas. Esta sección deberá actualizarse con regularidad para que incluya los riesgos recién identificados.

La cuarta sección del SPMP, *Proceso técnico*, describe los estándares técnicos que se requiere que adopten todos los equipos. Éstos van desde la metodología de desarrollo hasta la política de administración de la configuración de documentos y código, los lineamientos de codificación y la selección de componentes hechos estándar. Algunos pueden ser impuestos por intereses al nivel de la compañía y otros por el cliente.

Plan de administración de proyectos de software (SPMP)

1. Introducción
   - 1.1 Panorama del proyecto
   - 1.2 Productos a entregar del proyecto
   - 1.3 Evolución de este documento
   - 1.4 Referencias
   - 1.5 Definiciones y siglas
2. Organización del proyecto
   - 2.1 Modelo del proceso
   - 2.2 Estructura de organización
   - 2.3 Fronteras e interfaces de organización
   - 2.4 Responsabilidades del proyecto
3. Proceso administrativo
   - 3.1 Objetivos y prioridades de la administración
   - 3.2 Suposiciones, dependencias y restricciones
   - 3.3 Administración del riesgo
   - 3.4 Mecanismos de supervisión y control
4. Proceso técnico
   - 4.1 Métodos, herramientas y técnicas
   - 4.2 Documentación del software
   - 4.3 Funciones de soporte del proyecto
5. Elementos de trabajo, calendarización y presupuesto

Figura 11-21 Un ejemplo de plantilla para el SPMP.

La quinta sección del SPMP, *Elementos de trabajo, calendarización y presupuesto*, representa el producto más visible de la administración. Esta sección detalla la manera en que se realizará el trabajo y quien deberá llevarlo a cabo. Las primeras versiones del SPMP contienen planes detallados sólo para las primeras fases del proyecto. Los planes detallados para las fases posteriores se actualizan conforme avanza el proyecto y se comprenden mejor los riesgos importantes.

El SPMP se escribe al inicio del proceso, antes que se finalice el *Acuerdo del proyecto*. Lo revisan la administración y los desarrolladores para asegurarse que la calendarización sea factible y que no se hayan omitido riesgos importantes. Luego se actualiza el SPMP a lo largo del proceso cuando se toman decisiones o se descubren problemas. El SPMP, una vez que se publica, es la línea base y se pone bajo la administración de la configuración. La sección de historia de revisiones del SPMP proporciona una historia de los cambios en la forma de una lista de cambios, incluyendo al autor responsable del cambio, la fecha del cambio y una breve descripción.

11.5.2 Asignación de responsabilidades

La administración asigna dos tipos de papeles a los desarrolladores, papeles administrativos, como líder de equipo y enlace, y papeles técnicos, como arquitecto, analista o probador. La administración asigna estos papeles a un individuo o a un equipo, dependiendo de los requerimientos de esfuerzo y el tipo de papel asignado.

Asignación de papeles administrativos

Los papeles administrativos se asignan a participantes individuales. Un papel, como el de líder de equipo, no puede desempeñarse de manera efectiva si está compartido por dos o más participantes. Primero, un líder de equipo cubre una función de comunicación actuando como un enlace entre los altos niveles de administración y los desarrolladores y, por tanto, debe poder comunicar una imagen consistente del estado en ambos sentidos. Segundo, aunque un líder de equipo toma decisiones buscando, por lo general, el consenso de los desarrolladores, en ocasiones tiene que imponer decisiones críticas en el tiempo. Otros papeles administrativos, como un enlace entre equipos, también requiere el mismo grado de consistencia y, por tanto, se asignan a un individuo.

La administración busca tres cualidades cuando asigna papeles administrativos: la capacidad para comunicarse, la capacidad para reconocer pronto los riesgos y la disciplina para separar las decisiones administrativas de las técnicas. Tales cualidades con frecuencia provienen de la experiencia en el establecimiento de proyectos.

Asignación de papeles técnicos

Los papeles técnicos, a diferencia de los administrativos, pueden asignarse a un equipo de participantes. Los proyectos complejos tienen un equipo de arquitectura y un equipo de pruebas, compuesto únicamente por arquitectos de sistema y probadores, respectivamente. Una vez que la administración ha definido las responsabilidades del equipo, el líder del equipo asigna tareas individuales a cada miembro con base en su disponibilidad o habilidad. Los papeles técnicos no requieren el mismo nivel de consistencia que los administrativos, ya que sus tareas están mejor definidas y se apoyan en la administración y los enlaces para la coordinación.

Las habilidades requeridas para cada papel técnico varían de manera significativa con el papel. Sin embargo, conforme evolucionan con rapidez los métodos de diseño y tecnologías de implementación hay mucha más escasez de habilidades técnicas que administrativas, y en este caso se requiere mucho entrenamiento al inicio del proyecto. La asignación de papeles técnicos a los equipos permite también la mezcla de expertos con principiantes, quienes pueden complementar su entrenamiento con la disponibilidad de mentores experimentados.

Selección del tamaño de los equipos

Aunque los papeles técnicos pueden asignarse a los equipos, el tamaño de un solo equipo está restringido por la sobrecarga administrativa y de comunicación. Entre más participantes haya en un equipo, mayor es la sobrecarga de comunicación y menor la efectividad del equipo. Las observaciones que se dan a continuación están adaptadas de [Kayser, 1990], quien aplicó originalmente esta heurística a la formación de subgrupos durante reuniones. Tomando en cuenta que las reuniones de equipos son una herramienta significativa para la recopilación del estado, la negociación y la toma de decisiones, las restricciones en la cantidad de participantes en las reuniones son el límite superior para el tamaño del equipo.

- *Tres miembros.* Durante las reuniones cada miembro tiene oportunidad de hablar. Los asuntos se discuten a profundidad y se resuelven con facilidad. Un posible problema con este

tamaño de equipo es que un miembro puede dominar a los otros dos. Además, los equipos pequeños sufren la acumulación de varios papeles para cada participante (síndrome de "demasiados sombreros").

- *Cuatro miembros.* Al comenzar con equipos de este tamaño se permite que siga funcionando aunque un participante salga del equipo. Sin embargo, la obtención de consenso con un equipo de miembros pares puede ser problemática, ya que puede haber empate en las votaciones. En consecuencia, la resolución de problemas puede llevarse relativamente mucho tiempo.
- *Cinco y seis miembros.* Éste es el tamaño ideal para un proyecto de desarrollo de software. Los miembros todavía pueden reunirse frente a frente. Las diversas perspectivas, soportadas por una mezcla de ideas, opiniones y actitudes, promueven el pensamiento creativo. Se pueden asignar papeles únicos a cada miembro, creando una miniorganización en la que cada miembro complementa a los demás.
- *Siete miembros.* Éste es todavía un tamaño de equipo relativamente efectivo, pero las reuniones de equipo tienden a ser largas. La revisión del estado requiere más de media hora. Si se necesitan equipos de siete, es recomendable formar subequipos durante las reuniones de equipo formales, teniendo cada uno la tarea de discutir y resolver un subconjunto de los asuntos pendientes de la agenda.
- *Ocho y más miembros.* Los equipos de este tamaño llegan a ser difíciles de manejar. La estructura interna con frecuencia comienza a dividirse en subequipos. A menudo ocurren coaliciones y conversaciones laterales durante las reuniones formales. Los miembros tienen más oportunidades de competir que de cooperar. Aunque los resultados son satisfactorios, por lo general el tiempo para alcanzar estos resultados es más largo que con un equipo de menor tamaño.

11.5.3 Comunicación acerca de la administración del proyecto

La comunicación acerca de la administración del proyecto es difícil por varias razones. Como sucede con el análisis, los administradores, los desarrolladores y el cliente tienen diferentes conocimientos y no discuten el sistema en el mismo nivel de abstracción. Los administradores, los desarrolladores y el cliente tienen intereses creados sobre el sistema, y esto forma una barrera durante la comunicación. El cliente lucha por un sistema útil al menor costo. Los administradores luchan contra las restricciones respecto al tiempo y el presupuesto, ya sea pidiendo un precio más alto al cliente o presionando a los desarrolladores para que hagan el desarrollo más rápido para bajar el costo. Los desarrolladores tienen el conocimiento técnico y toman las intervenciones del cliente o de los administradores como una incursión en su territorio. Estos factores, ya presentes durante el análisis, empeoran cuando la comunicación es acerca de la administración, incluyendo la asignación de recursos y la revisión del estado.

A continuación exponemos heurísticas para facilitar la comunicación entre las tres partes.

Comunicación con el cliente

La comunicación con el cliente se caracteriza por su naturaleza episódica. El cliente está presente en el arranque, durante las revisiones externas y durante la prueba de aceptación. En los demás momentos el cliente no está disponible, por lo general. Esto motiva una comunicación frente a frente para la toma de decisiones seguida de unos documentos cortos escritos como registro de estas decisiones. Incluyen el *Enunciado del problema*, que se usa cuando arranca el proyecto, el *Acuerdo del proyecto*, cuando se pone la línea base del alcance del proyecto y la *Aceptación del cliente*, una vez que se entrega el producto. La información de estado detallada y los borradores parciales de la información técnica, por lo general, no se comparten con el cliente.

Comunicación con los líderes de equipo

La comunicación entre la administración del proyecto y los líderes de equipo determina la calidad de la información de estado disponible para la administración del proyecto. Si se ve que la administración del proyecto está reaccionando en forma prematura a los reportes de estado, un líder de equipo dejará de reportar los problemas que considera que puede manejar solo. Si el gerente del proyecto cree que los líderes de equipo toman demasiadas iniciativas ocultará información crítica. El establecimiento de una reunión semanal con la administración del proyecto y los líderes de equipo, en donde la información fluya en forma libre, es un elemento esencial para la resolución de problemas a tiempo. El gerente del proyecto puede apoyar esta comunicación separando las reuniones de estado de las reuniones de decisiones. Además, el gerente del proyecto puede recompensar el reporte de problemas poniendo a disposición del líder del equipo que reporta los recursos adecuados para resolver el problema.

Comunicación con los desarrolladores

El gerente del proyecto, por lo general, no se comunica de manera directa con los desarrolladores en la misma forma en que se reúne con los líderes de equipo. Los canales de comunicación principales son el SPMP, las revisiones del proyecto y los líderes de equipo. Con más frecuencia, los líderes de equipo actúan como un aislamiento entre la administración del proyecto y los desarrolladores, llenando el hueco entre los niveles de abstracción más alto y más bajo. La comunicación entre el líder del equipo y los desarrolladores es esencial, ya que el líder del equipo tiene, por lo general, la experiencia técnica para valorar los problemas que encuentran los desarrolladores.

11.6 Ejercicios

1. ¿Cuáles son las ventajas relativas de la contratación llana contra la contratación gradual?
2. ¿Cuál es la diferencia entre las reuniones de estado y las de decisión? ¿Por qué deben mantenerse separadas?
3. ¿Por qué deben asignarse los papeles de arquitecto y líder de equipo a personas distintas?
4. Trace un modelo UML de la organización de equipos del proyecto OWL para cada una de las tres fases (es decir, inicio, estado estable, terminación).
5. Trace un plan de tareas detallado del diseño de sistema del sistema MiViaje que se presentó en el capítulo 6, *Diseño del sistema.*
6. Estime el tiempo para terminar cada tarea del modelo de tareas producido en el ejercicio 5 y determine la ruta crítica.
7. Identifique, priorice y planee los cinco riesgos mayores relacionados con el diseño del sistema MiViaje que se presentó en el capítulo 6, *Diseño del sistema.*
8. Identifique, priorice y planee los cinco riesgos mayores relacionados con el subsistema de interfaz de usuario del CTC que se presentó en el capítulo 8, *Administración de la fundamentación.*
9. Linux, desarrollado usando el modelo bazar, es más confiable y más sensible que muchos sistemas operativos que ejecutan en PC Intel. Sin embargo, discuta en forma detallada por qué no debe usarse el modelo bazar para el software de control del transbordador espacial.
10. Compare los papeles de gerente de proyecto, jefe de desarrollo y jefe técnico, como se definieron en la sección 11.3.2. En particular, describa las tareas de las que es responsable cada jefe y sobre cuáles decisiones tiene la última palabra.

Referencias

[Allen, 1985] T. J. Allen, *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization*, 2a. ed., MIT Press, Cambridge, MA, 1995.

[Boehm, 1987] B. Boehm, "A spiral model of software development and enhancement", *Software Engineering Project Management*, 1987, págs. 128–142.

[Boehm, 1991] B. Boehm, "Software risk management: Principles and practices", *IEEE Software*, vol. 1, 1991, págs. 32–41.

[Brooks, 1975] F. P. Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, MA, 1975.

[Bruegge *et al.*, 1997] B. Bruegge, S. Chang, T. Fenton, B. Fernandes, V. Hartkopf, T. Kim, R. Pravia y A. Sharma, "Turning lightbulbs into objects", *OOPSLA'97*, Ponencia de experiencia, Atlanta, 1997.

[Carr *et al.*, 1993] M. J. Carr, S. L. Konda, I. Monarch, F. C. Ulrich y C. F. Walker, "Taxonomy-based risk identification", *Technical Report CMU/SEI-93-TR-6*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[Charette, 1989] R. N. Charette, *Software Engineering Risk Analysis and Management*, McGraw-Hill, Nueva York, 1989.

[Hartkopf *et al.*, 1997] V. Hartkopf, V. Loftness, A. Mahdavi, S. Lee y J. Shankavarm, "An integrated approach to design and engineering of intelligent buildings—The Intelligent Workplace at Carnegie Mellon University", *Automation in Construction*, vol. 6, 1997, págs. 401–415.

[Hauschildt y Gemuenden, 1998] J. Hauschildt y H. G. Gemuenden, *Promotoren*, Gabler, Wiesbaden, Alemania, 1998.

[Hillier y Lieberman, 1967] F. S. Hillier y G. J. Lieberman, *Introduction to Operation Research*, Holden-Day, San Francisco, 1967.

[IEEE Std. 1058.1-1993] *IEEE Standard for Software Project Management Plans*, IEEE Computer Society, Nueva York, 1993.

[Kayser, 1990] T. A. Kayser, *Mining Group Gold*, Serif, El Segundo, CA, 1990.

[Kemerer, 1997] C. F. Kemerer, *Project Management: Readings and Cases*, Irwin/McGraw-Hill, Boston, MA, 1997.

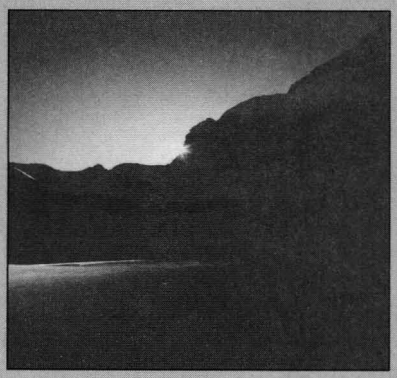
[OWL, 1996] *OWL Project Documentation*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Raymond, 1998] E. Raymond, "The cathedral and the bazaar", disponible en http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html, 1998.

[Rogers *et al.*, 1986] *The Presidential Commission on the Space Shuttle Challenger Accident Report*, Washington, DC, 6 de junio de 1986.

[Vaughan, 1996] D. Vaughan, *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA*, The University of Chicago, Chicago, 1996.

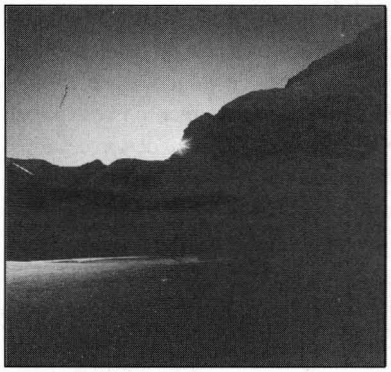
[Weinberg, 1971] G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand, Nueva York, 1971.



PARTE IV
Vuelta a empezar

12

12



Ciclo de vida del software

Siempre habrá una discrepancia entre los conceptos y la realidad, debido a que los primeros son estáticos y la segunda es dinámica y cambiante.
—Robert Pirsig, en Lila

Un modelo de ciclo de vida del software representa todas las actividades y productos de trabajo necesarios para desarrollar un sistema de software. Los modelos de ciclo de vida permiten que los gerentes y desarrolladores manejen la complejidad del proceso de desarrollo de software en la misma forma que un modelo de análisis o un modelo de diseño del sistema permite que los desarrolladores manejen la complejidad de un sistema de software. En el caso de los sistemas de software, la realidad que se está modelando incluye fenómenos como los relojes, los accidentes, los trenes, los sensores y los edificios. En el caso del desarrollo de software, la realidad incluye fenómenos como los participantes, los equipos, las actividades y los productos de trabajo. En la literatura se han publicado muchos modelos de ciclo de vida como intentos para comprender, medir y controlar mejor el proceso de desarrollo. Los modelos de ciclo de vida hacen que las actividades de desarrollo de software y sus dependencias sean visibles y manejables.

En este capítulo describimos modelos de ciclo de vida seleccionados y bien conocidos, y revisamos las actividades descritas en capítulos anteriores desde la perspectiva del modelado del ciclo de vida. Las técnicas de modelado que usamos para el modelado de los sistemas de software también pueden usarse para la representación de los ciclos de vida. Aunque los modelos de ciclo de vida se representan, por lo general, con notaciones *ad hoc*, nosotros usamos diagramas de clase UML y diagramas de actividad UML siempre que es posible. Primero describimos las actividades y productos de trabajo típicos de un ciclo de vida del software, como los define el estándar IEEE 1074 [IEEE Std. 1074-1995]. Luego presentamos el modelo de madurez de capacidades, un marco para la valoración de la madurez de las organizaciones y sus ciclos de vida. Luego tratamos diferentes modelos de ciclos de vida que se han propuesto para manejar el desarrollo de sistemas de software complejos. También presentamos un nuevo modelo de ciclo de vida llamado modelo basado en problemas, en donde los productos y las actividades se modelan como un conjunto de problemas guardados en una base de conocimiento. El modelo basado en problemas es una visión centrada en entidades del ciclo de vida del software que maneja mejor los cambios frecuentes dentro de la duración de un proyecto. Concluimos con dos ejemplos de los modelos de ciclo de vida del software que pueden usarse para un proyecto de curso de uno o dos semestres.

12.1 Introducción

En el capítulo 2, *Modelado con UML*, describimos al modelado como la clasificación de fenómenos del mundo real en conceptos. Hasta ahora nos han interesado los fenómenos relacionados con los relojes, el manejo de emergencias, el control de tráfico, los catálogos de partes de vehículos y el modelado ambiental. Sin embargo, este capítulo trata el proceso de desarrollo de software como la realidad que nos interesa. Los fenómenos que queremos modelar incluyen todas las actividades necesarias para construir un sistema de software y los productos de trabajo producidos durante el desarrollo. A los modelos del proceso de desarrollo de software les llamamos **modelos del ciclo de vida del software**. El modelado del ciclo de vida del software es una empresa difícil, debido a que es un sistema cambiante y complejo. Al igual que los sistemas de software, los ciclos de vida del software pueden describirse mediante modelos diferentes. La mayoría de los modelos de ciclo de vida del software propuestos se han enfocado en las actividades del desarrollo del software y las representan de manera explícita como objetos de primera clase. A esta perspectiva del ciclo de vida del software se le llama **centrada en actividad**. Una visión alterna del ciclo de vida del software es enfocarse en los productos de trabajo creados por estas actividades. A esta perspectiva alterna se le llama **centrada en entidad**. La perspectiva centrada en actividad conduce a los participantes a que se enfoquen en la manera en que se crean los productos de trabajo. La visión centrada en entidad conduce a los participantes a que se enfoquen en el contenido y estructura de los productos de trabajo.

La figura 12-1 muestra un ciclo de vida simple para el desarrollo de software usando tres actividades: `Definición del problema`, `Desarrollo del sistema` y `Operación del sistema`.

La figura 12-2 muestra una visión centrada en actividad de este modelo de ciclo de vida simplista. Las asociaciones entre las actividades muestran una dependencia lineal en el tiempo, que está implícita en el uso de la notación de diagrama de actividad: el enunciado del problema precede al desarrollo del sistema, que a su vez precede a la operación del sistema. Son posibles dependencias de tiempo alternas.

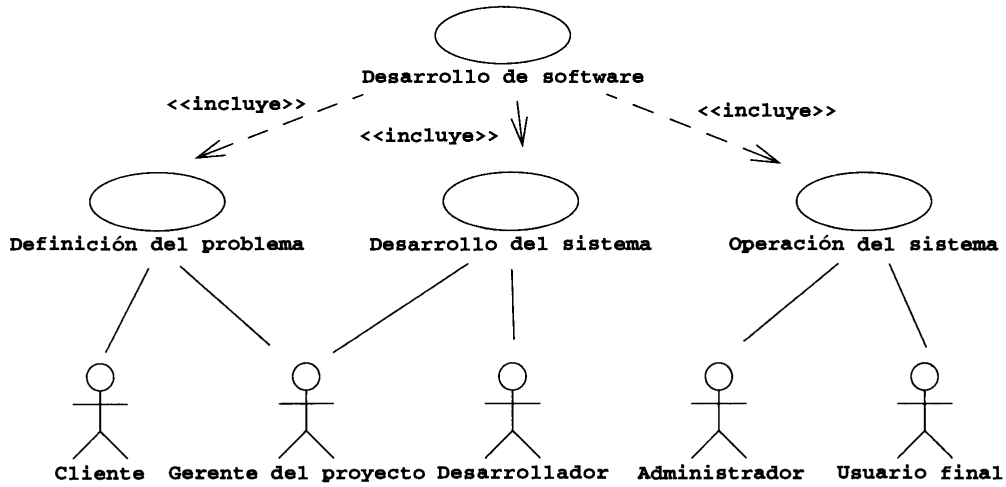


Figura 12-1 Ciclo de vida simple del desarrollo de software (diagrama de caso de uso UML).

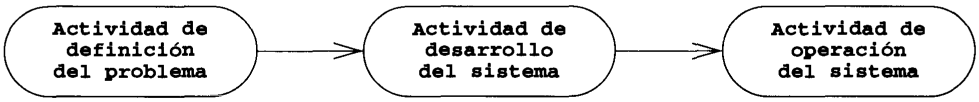


Figura 12-2 Ciclo de vida simple para el desarrollo de software (diagrama de actividad UML).

Por ejemplo, en el ciclo de vida del software de la figura 12-3 las actividades `Desarrollo del sistema` y `Creación de mercado` pueden realizarse de manera concurrente.

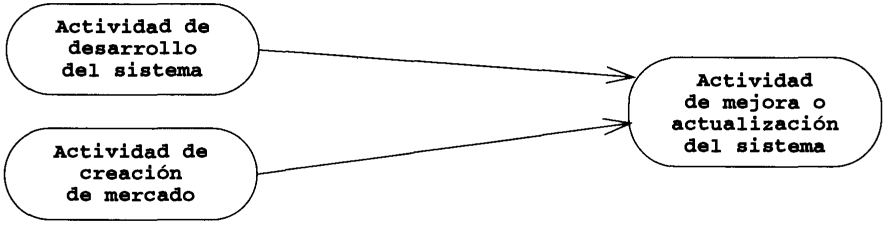


Figura 12-3 Otro ciclo de vida simple (diagrama de actividad UML).

La figura 12-4 es una visión centrada en entidad del modelo que se muestra en la figura 12-2. El desarrollo del software produce cuatro entidades, un `Documento de estudio de mercado`, un `Documento de especificación del sistema`, un `Sistema ejecutable` y un `Documento de lecciones aprendidas`.

Las perspectivas centrada en actividad y centrada en entidad son complementarias, como lo ilustra la figura 12-5. Para cada producto hay una o más actividades que lo generan. La `Actividad de definición del problema` usa un `Documento de estudio de mercado` como entrada y genera un `Documento de especificación del sistema`. La actividad `Desarrollo del sistema` toma el `Documento de especificación del sistema` como entrada y produce un `Sistema ejecutable`. La `Operación del sistema` genera un `Documento de lecciones aprendidas` que podrá usarse durante el desarrollo del siguiente producto. En forma alternativa, cada actividad genera uno o más productos.

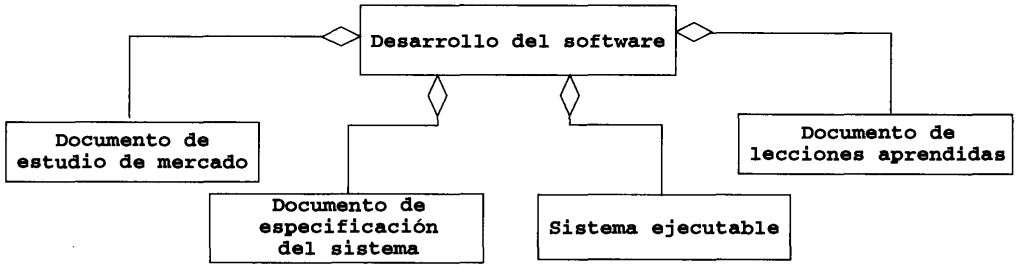


Figura 12-4 Visión centrada en entidad del desarrollo de software (diagrama de clase UML).

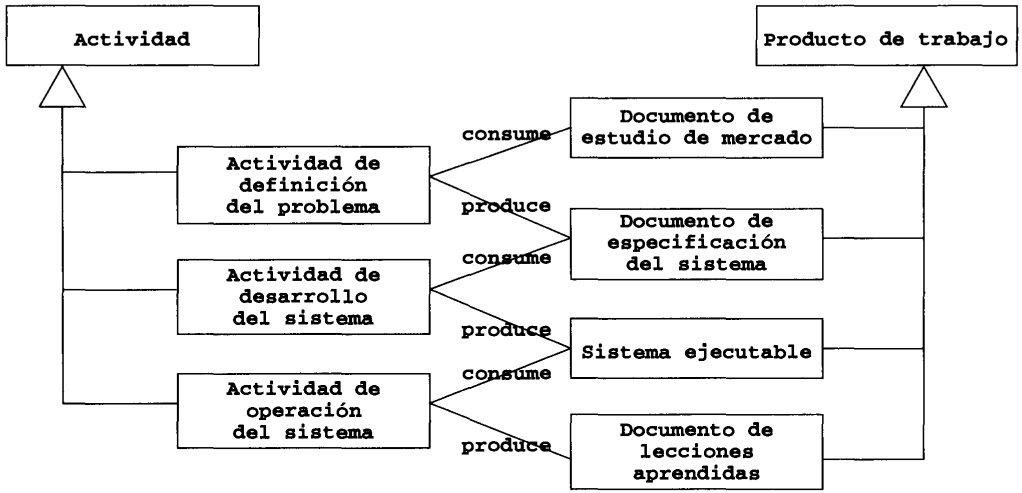


Figura 12-5 Actividades y productos del ciclo de vida simple de la figura 12-2.

En la sección 12.2 describimos las actividades del ciclo de vida definidas por el estándar IEEE 1074 [IEEE Std. 1074-1995]. Este estándar presenta definiciones precisas que permiten a los participantes en el proyecto comprender y comunicarse de manera efectiva acerca del ciclo de vida. En esta sección también describimos los flujos de información entre actividades.

En la sección 12.3 describimos el modelo de madurez de capacidades, un marco para la valoración de la madurez de las organizaciones y sus ciclos de vida. Este marco permite que se comparen las organizaciones y los proyectos con base en las actividades de sus ciclos de vida.

En la sección 12.4 investigamos varios modelos de ciclo de vida centrados en actividad que proponen diferentes ordenamientos de actividades. En particular exponemos el modelo de cascada [Royse, 1970], el modelo de espiral [Boehm, 1987], el modelo V [Jensen y Tonies, 1979], el modelo de diente de sierra [Rowen, 1990] y el proceso de software unificado [Jacobson *et al.*, 1999].

En la sección 12.5 describimos dos instancias específicas del modelo de ciclo de vida para un proyecto basado en equipo, incluyendo sus actividades, productos y flujo de información.

12.2 IEEE 1074: el estándar para el desarrollo de procesos del ciclo de vida

El *estándar IEEE para los procesos del ciclo de vida del software* describe el conjunto de actividades y procesos obligatorios para el desarrollo y mantenimiento del software [IEEE Std. 1074-1995]. Su objetivo es establecer un marco común para el desarrollo de modelos de ciclo de vida y proporciona ejemplos de situaciones típicas. En esta sección describimos las actividades principales presentadas por el estándar y aclaramos sus conceptos fundamentales usando diagramas UML.

12.2.1 Procesos y actividades

Un **proceso** es un conjunto de actividades que se realiza para un propósito específico (por ejemplo, requerimientos, administración, entrega). El estándar IEEE lista un total de 17 procesos (tabla 12-1). Los procesos están agrupados en niveles de abstracción más elevados llamados **grupos de procesos**. Son ejemplos de grupos de procesos la administración del proyecto, el predesarrollo, el desarrollo y el posdesarrollo. Los ejemplos de procesos en el grupo de procesos de desarrollo incluyen:

- El *Proceso de requerimientos,* durante el cual los desarrolladores desarrollan los modelos del sistema.
- El *Proceso de diseño,* durante el cual los desarrolladores separan al sistema en componentes.
- El *Proceso de implementación,* durante el cual los desarrolladores realizan cada componente.

Tabla 12-1 Procesos de software en el IEEE 1074.

Grupo de procesos	Procesos
Modelado del ciclo de vida	Selección de un modelo de ciclo de vida
Administración del proyecto	Inicio del proyecto Supervisión y control del proyecto Administración de la calidad del software
Predesarrollo	Exploración de conceptos Asignación del sistema
Desarrollo	Requerimientos Diseño Implementación
Posdesarrollo	Instalación Operación y soporte Mantenimiento Retiro
Procesos integrales	Verificación y validación Administración de la configuración del software Desarrollo de la documentación Entrenamiento

Cada proceso está compuesto por actividades. Una **actividad** es una tarea o grupo de subactividades que se asignan a un equipo o a un participante del proyecto para lograr un propósito especifico. El *Proceso de requerimientos*, por ejemplo, está compuesto por tres actividades:

- *Definir y desarrollar los requerimientos de software,* durante la cual se define con precisión la funcionalidad del sistema.
- *Definir los requerimientos de interfaz,* durante la cual se definen con precisión las interacciones entre el sistema y el usuario.

- *Establecer prioridades e integrar los requerimientos de software,* durante la cual se integran todos los requerimientos para que tengan consistencia y se establecen prioridades de acuerdo con la preferencia del cliente.

Las tareas consumen recursos (por ejemplo, personal, tiempo, dinero) y crean un producto de trabajo. Durante la planeación las actividades se descomponen en tareas específicas del proyecto, se les dan fechas de inicio y terminación y se asignan a un equipo o a un participante en el proyecto (figura 12-6). Durante el proyecto se da seguimiento al trabajo real contra las tareas planeadas, y los recursos se reasignan para responder a los problemas.

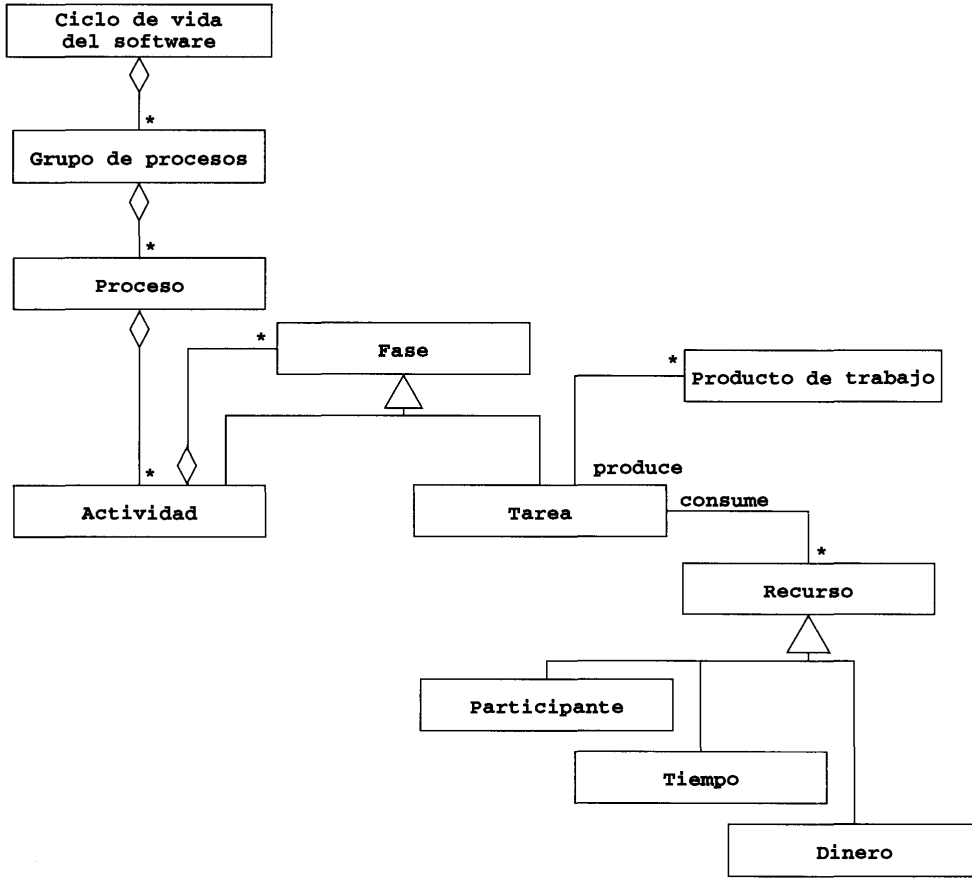


Figura 12-6 Modelo del ciclo de vida del software (diagrama de clase UML). Un ciclo de vida del software consta de grupos de procesos, que a su vez constan de procesos. Un proceso logra un propósito específico (por ejemplo, requerimientos, diseño, instalación). Un proceso consta de actividades, que a su vez constan de subactividades o tareas. Las tareas representan la parte de trabajo más pequeña que es relevante para la administración. Las tareas consumen recursos y crean uno o más productos de trabajo. Un proyecto es una instancia de un ciclo de vida del software.

Los procesos requeridos por el IEEE 1074 se listan en la tabla 12-1. Desde el punto de vista del desarrollador, los primeros seis procesos (es decir, el *Proceso de modelado del ciclo de vida*, los *Procesos de administración del proyecto* y los *Procesos de predesarrollo*) a menudo ya se han iniciado antes de su involucramiento con el proyecto.

12.2.2 Modelado del ciclo de vida

Durante el *Modelado del ciclo de vida*, el gerente del proyecto personaliza las actividades definidas en el IEEE 1074 para un proyecto especifico (es decir, para una instancia del modelo del ciclo de vida). No todos los proyectos requieren las mismas actividades y la misma secuencia de actividades. Por ejemplo, los proyectos que no manejan el almacenamiento persistente no necesitan ejecutar la actividad *Diseño de la base de datos*. El modelo del ciclo de vida seleccionado sirve como entrada al *Proceso de inicio del proyecto* que se describe en la siguiente sección. En la sección 12.5 proporcionamos los ejemplos del *Modelado del ciclo de vida*.

12.2.3 Administración del proyecto

Durante el grupo de *Administración del proyecto*, el gerente del proyecto inicia, supervisa y controla el proyecto por todo el ciclo de vida del software. La *Administración del proyecto* consta de tres procesos (tabla 12-2).

El *Proceso de inicio del proyecto* crea la infraestructura para el proyecto. Durante este proceso se define el plan de tareas, la calendarización, el presupuesto, la organización y el ambiente del proyecto. El ambiente del proyecto incluye los estándares del proyecto, la infraestructura de comuni-

Tabla 12-2 Procesos de la administración del proyecto.

Proceso	Cláusula[a]	Actividades
Inicio del proyecto	3.1.3	Establecimiento de la correspondencia entre las actividades y el modelo del ciclo de vida del software.
	3.1.4	Asignación de recursos al proyecto.
	3.1.5	Establecimiento del ambiente del proyecto.
	3.1.6	Planeación de la administración del proyecto.
Supervisión y control del proyecto	3.2.3	Analizar riesgos.
	3.2.4	Realizar planeación de contingencias.
	3.2.5	Administrar el proyecto.
	3.2.6	Conservar registros.
	3.2.7	Implementar el modelo de reporte de problemas.
Administración de la calidad del software	3.3.3	Planear la administración de la calidad del software.
	3.3.4	Definir medidas.
	3.3.5	Administrar la calidad del software.
	3.3.6	Identificar las necesidades de mejora de calidad.

a. La columna "Cláusula" de esta tabla y las demás de este capítulo se refiere a un número de cláusula del IEEE 1074. Es una referencia cruzada hacia el documento de estándares como se publicó en [IEEE Std. 1074-1995].

cación, los procedimientos de reunión y reporte, la metodología de desarrollo y las herramientas de desarrollo. La mayor parte de la información generada durante este proceso se documenta en el *Plan de administración del proyecto de software* (SPMP, por sus siglas en inglés). El *Proceso de inicio del proyecto* se termina tan pronto como se establece un ambiente estable para el proyecto.

El *Proceso de supervisión y control del proyecto* asegura que el proyecto se ejecute de acuerdo con el plan de tareas y el presupuesto. Si el gerente del proyecto observa alguna desviación con respecto al calendario tomará acciones correctivas, como la reasignación de algunos de los recursos, el cambio de algunos procedimientos o la replaneación de la calendarización. El SPMP se actualiza para reflejar cualquiera de estos cambios. El *Proceso de supervisión y control del proyecto* está activo durante todo el ciclo de vida.

El *Proceso de administración de la calidad del software* asegura que el sistema que se está construyendo satisfaga los estándares de calidad requeridos (que se seleccionaron durante el inicio del proyecto). Este proceso lo ejecuta un equipo de administración de la calidad separado para evitar conflictos de intereses (es decir, el objetivo de los desarrolladores es terminar el sistema a tiempo, y el objetivo del equipo de administración de la calidad es asegurarse que al sistema no se le considere terminado sino hasta que satisfaga el estándar de calidad requerido). El *Proceso de administración de la calidad del software* está activo durante la mayor parte del ciclo de vida.

En el capítulo 11, *Administración del proyecto*, describimos las actividades de *Inicio del proyecto* y *Supervisión y control del proyecto* que están relacionadas con la planeación, la organización y el seguimiento. La actividad *Establecer el ambiente del proyecto* requiere particular atención en el contexto de un proyecto basado en equipos. Una de las partes críticas del ambiente del proyecto es la infraestructura de comunicación que soportará la diseminación de información entre los participantes. Para poder reaccionar con rapidez ante los cambios y reportar problemas sin introducir una sobrecarga irrazonable, todos los participantes en el proyecto necesitan estar conscientes del flujo de información por el proyecto y de los mecanismos para la diseminación de la información. En el capítulo 3, *Comunicación de proyectos*, describimos las actividades relacionadas con la definición y uso de la infraestructura de comunicación. Tome en cuenta que para definir la estructura del equipo de desarrollo y, por tanto, la infraestructura de comunicación, necesita definirse la arquitectura inicial del sistema (producida por el *Proceso de asignación del sistema*, que se describe en la siguiente sección).

12.2.4 Predesarrollo

Durante el *predesarrollo*, la administración (o marketing) y el cliente identifican una idea o una necesidad. Esto puede resolverse con un nuevo esfuerzo de desarrollo (ingeniería *greenfield*), con un cambio a la interfaz de un sistema existente (ingeniería de interfaz) o con un reemplazo de software de un proceso de negocios existente (reingeniería). El *Proceso de asignación del sistema* establece la arquitectura inicial del sistema e identifica el hardware, el software y los requerimientos funcionales. Tome en cuenta que la descomposición en subsistemas es la base de la infraestructura de comunicación entre los miembros del proyecto. Los requerimientos, la descomposición en subsistemas y la infraestructura de comunicación se describen en el *Enunciado del problema*,[1] que sirve como entrada para el proceso *Desarrollo*. Los procesos del predesarrollo se muestran en la tabla 12-3.

1. El *Enunciado de necesidades* mencionado en el estándar IEEE 1074 es similar al *Enunciado del problema*, pero no contiene ninguna información sobre la organización del proyecto.

Tabla 12-3 Procesos del predesarrollo.

Proceso	Cláusula	Actividades
Exploración del concepto	4.1.3	Identificar ideas o necesidades
	4.1.4	Formular enfoques potenciales
	4.1.5	Realizar estudios de factibilidad
	4.1.6	Planear la transición del sistema (si es aplicable)
	4.1.7	Refinar y finalizar la idea o necesidad
Asignación del sistema	4.2.3	Analizar funciones
	4.2.4	Desarrollar la arquitectura del sistema
	4.2.5	Descomponer los requerimientos del sistema

12.2.5 Desarrollo

El *Desarrollo* consiste en los procesos que se dirigen a la construcción del sistema.

El *Proceso de requerimientos* se inicia con la descripción informal de los requerimientos y define los requerimientos del sistema desde el punto de vista de los requerimientos funcionales de alto nivel, produciendo una especificación completa del sistema y estableciendo la prioridad de los requerimientos. En los capítulos 4, *Obtención de requerimientos*, y 5, *Análisis*, describimos el *Proceso de requerimientos.*

El *Proceso de diseño* toma la arquitectura producida durante el *Proceso de asignación del sistema* y las especificaciones de los *Requerimientos* y produce una representación del sistema coherente y bien organizada. Las actividades *Realizar el diseño arquitectónico* y *Diseñar interfaces* dan como resultado un refinamiento de la descomposición en subsistemas. Esto también incluye la asignación de los requerimientos a los sistemas de hardware y software, la descripción de las condiciones de frontera, la selección de componentes hechos y la definición de los objetivos de diseño. El diseño detallado de cada subsistema se realiza durante la actividad *Realizar el diseño detallado.* El *Proceso de diseño* da como resultado la definición de los objetos de diseño, sus atributos y operaciones y su organización en paquetes. Al término de esta actividad se tienen

Tabla 12-4 Procesos de desarrollo.

Proceso	Cláusula	Actividades
Requerimientos	5.1.3	Definir y desarrollar los requerimientos de software
	5.1.4	Definir los requerimientos de la interfaz
	5.1.5	Establecer la prioridad e integrar los requerimientos de software

Tabla 12-4 Procesos de desarrollo.

Proceso	Cláusula	Actividades
Diseño	5.2.3	Realizar el diseño arquitectónico
	5.2.4	Diseñar la base de datos (si es aplicable)
	5.2.5	Diseñar interfaces
	5.2.6	Seleccionar o desarrollar algoritmos (si es aplicable)
	5.2.7	Realizar el diseño detallado
Implementación	5.3.3	Crear datos de prueba
	5.3.4	Crear código fuente
	5.3.5	Crear código objeto
	5.3.6	Crear la documentación operativa
	5.3.7	Planear la integración
	5.3.8	Realizar la integración

definidos todos los métodos y sus firmas de tipo. Se introducen nuevas clases para tomar en cuenta los requerimientos no funcionales y los detalles específicos de los componentes. El proceso de *Diseño* usado en este libro se describe en los capítulos 6, *Diseño del sistema*, y 7, *Diseño de objetos*.

El *Proceso de implementación* toma el modelo de diseño y produce una representación ejecutable equivalente. El *Proceso de implementación* incluye la planeación de la integración y las actividades de la integración. Tome en cuenta que las pruebas que se realizan durante este proceso son independientes de las realizadas durante el control de calidad o la *Verificación y validación*. En el capítulo 9, *Pruebas*, describimos los aspectos de las pruebas y la integración de la *Implementación*. Los procesos de desarrollo se muestran en la tabla 12-4.

12.2.6 Posdesarrollo

El *Posdesarrollo* consta de los procesos de instalación, mantenimiento, operación y soporte y retiro (tabla 12-5).

Durante la *Instalación* se distribuye e instala el software del sistema en el sitio del cliente. La instalación culmina con la prueba de aceptación del cliente de acuerdo con los criterios definidos en el *Acuerdo del proyecto*. La *Operación y soporte* se refieren al entrenamiento de los usuarios y a la operación del sistema. El *Mantenimiento* se encarga de la resolución de errores, defectos y fallas del software después de la entrega del sistema. El *Mantenimiento* requiere la elevación de los procesos y actividades del ciclo de vida del software hacia un nuevo proyecto. El *Retiro* elimina un sistema existente, dando por terminadas sus operaciones o soporte. El *Retiro* sucede cuando se mejora el sistema o se reemplaza con uno nuevo. Para asegurar una transición suave entre los dos sistemas, a menudo se ejecutan ambos en paralelo hasta que los usuarios se han acostumbrado al nuevo sistema. A excepción de la entrega y aceptación por parte del cliente, en este libro no tratamos los procesos del posdesarrollo.

Tabla 12-5 Procesos del posdesarrollo.

Proceso	Cláusula	Actividades
Instalación	6.1.3	Planear la instalación
	6.1.4	Distribuir el software
	6.1.5	Instalar el software
	6.1.7	Aceptar el software en el ambiente operacional
Operación y soporte	6.2.3	Operar el sistema
	6.2.4	Proporcionar asistencia técnica y consultoría
	6.2.5	Mantener la bitácora de peticiones de soporte
Mantenimiento	6.3.3	Volver a aplicar el ciclo de vida del software
Retiro	6.4.3	Notificar a los usuarios
	6.4.4	Realizar operaciones paralelas (si es aplicable)
	6.4.5	Retirar el sistema

12.2.7 Procesos integrales (desarrollo cruzado)

Durante toda la extensión del proyecto se realizan varios procesos. A éstos se les llama procesos integrales (también usamos el término *procesos de desarrollo cruzado*) e incluyen la *Validación y verificación*, la *Administración de la configuración del software*, el *Desarrollo de la documentación* y el *Entrenamiento* (tabla 12-6).

La *Verificación y validación* incluye las tareas de verificación y validación. Las tareas de verificación se enfocan en mostrar que los modelos del sistema se apegan a las especificaciones. La verificación incluye revisiones, auditorías e inspecciones. Las tareas de validación aseguran que el sistema resuelva las necesidades del cliente e incluyen la prueba del sistema, las pruebas beta y la prueba de aceptación del cliente. Las actividades de *Verificación y validación* suceden durante todo el ciclo de vida tratando de detectar anomalías lo más pronto posible. Por ejemplo, cada modelo puede revisarse contra una lista de verificación al final del proceso que lo generó. La revisión de un modelo, digamos el modelo de diseño, puede dar como resultado la modificación de un modelo generado en otros procesos, digamos el modelo de análisis. La actividad *Recopilar y analizar los datos de mediciones* genera datos del proyecto que también pueden servir para proyectos futuros y contribuyen al conocimiento de la organización. Las actividades *Planear pruebas* y *Desarrollar requerimientos de pruebas* pueden iniciarse de inmediato después de la terminación de los requerimientos. En proyectos grandes, estas tareas las realizan participantes diferentes a los desarrolladores. En el capítulo 3, *Comunicación de proyectos*, describimos mecanismos para las revisiones, auditorías e inspecciones. En los capítulos 4, 5 y 6 describimos revisiones específicas asociadas con la obtención de requerimientos, el análisis y el diseño del sistema, respectivamente. En el capítulo 9, *Pruebas*, describimos las actividades de pruebas.

Los *Procesos de administración de la configuración* se enfocan en el seguimiento y control de los cambios a los productos de trabajo. Los elementos en la administración de la configuración incluyen el código fuente del sistema, todos los modelos de desarrollo, el plan de administración del proyecto de software y todos los documentos visibles para los participantes en el proyecto. En

Tabla 12-6 Procesos integrales (llamados también procesos de desarrollo cruzado).

Proceso	Cláusula	Actividad
Verificación y validación	7.1.3	Planear la verificación y validación
	7.1.4	Ejecutar las tareas de verificación y validación
	7.1.5	Recopilar y analizar datos de medidas
	7.1.6	Planear las pruebas
	7.1.7	Desarrollar los requerimientos de las pruebas
	7.1.8	Ejecutar las pruebas
Administración de la configuración del software	7.2.3	Planear la administración de la configuración
	7.2.4	Desarrollar la identificación de la configuración
	7.2.5	Realizar el control de la configuración
	7.2.6	Realizar la contabilización del estado
Desarrollo de la documentación	7.3.3	Planear la documentación
	7.3.4	Implementar la documentación
	7.3.5	Producir y distribuir la documentación
Entrenamiento	7.4.3	Planear el programa de entrenamiento
	7.4.4	Desarrollar los materiales de entrenamiento
	7.4.5	Validar el programa de entrenamiento
	7.4.6	Implementar el programa de entrenamiento

el capítulo 10, *Administración de la configuración del software*, describimos la administración de la configuración.

Los *Procesos de documentación* tratan con los productos de trabajo (excluyendo al código) que documentan los resultados producidos por los demás procesos. Las plantillas de documentos se seleccionan durante esta actividad. Sin embargo, el estándar IEEE 1074 no prescribe ningún documento o plantilla específicos. Los asuntos específicos de la documentación del desarrollo y el desarrollo cruzado se tratan en los capítulos donde se producen los documentos (por ejemplo, el capítulo 5, *Análisis*, expone el *Documento de análisis de requerimientos*, el capítulo 10, *Administración de la configuración del software*, trata el *Plan de administración de la configuración*).

12.3 Caracterización de la madurez de los modelos del ciclo de vida del software

En la sección anterior presentamos el conjunto de actividades y artefactos que constituyen el ciclo de vida del software. Las actividades y artefactos que se escogen para un proyecto específico no están definidos por el estándar. Uno de los objetivos del modelo de madurez de capacidades (CMM, por sus siglas en inglés) es proporcionar lineamientos para la selección de las actividades del ciclo de vida. El CMM asume que el desarrollo de los sistemas de software se hace más predecible cuando una organización usa un proceso de ciclo de vida bien estructurado, visible para todos los participantes en el proyecto y que se adapta al cambio. El CMM usa los siguientes cinco niveles para caracterizar la madurez de una organización [Paulk *et al.*, 1995].

Nivel 1: Inicial. Una organización que está en el nivel inicial aplica actividades *ad hoc* para desarrollar software. Pocas de estas actividades están bien definidas. El éxito de un proyecto en este nivel de madurez depende, por lo general, de los esfuerzos heroicos y habilidades de individuos clave. Desde el punto de vista del cliente, el modelo de ciclo de vida del software, si es que existe, es una caja negra: después de proporcionar el enunciado del problema y negociar el acuerdo del proyecto, el cliente debe esperar hasta el final del proyecto para inspeccionar los productos a entregar del proyecto. Durante todo el proyecto el cliente no tiene una forma efectiva para interactuar con la administración del proyecto.

Nivel 2: Repetible. Cada proyecto tiene un modelo de ciclo de vida del software bien definido. Sin embargo, los modelos difieren entre proyectos, reduciendo la oportunidad del trabajo en equipo y la reutilización del conocimiento. Los procesos básicos de administración del proyecto se usan para el seguimiento de los costos y la calendarización. Los nuevos proyectos se basan en la experiencia de la organización con proyectos similares anteriores, y el éxito es predecible para proyectos en dominios de aplicación similares. El cliente interactúa con la organización en momentos bien definidos, como las revisiones del cliente y la prueba de aceptación del cliente, permitiendo algunas correcciones antes de la entrega.

Nivel 3: Definido. Este nivel usa un modelo de ciclo de vida del software documentado para todas las actividades administrativas y técnicas por toda la organización. Al inicio de cada proyecto se produce una versión personalizada del modelo durante la actividad *Modelado del ciclo de vida. El cliente conoce el modelo estándar y el modelo seleccionado para el proyecto específico.*

Nivel 4: Administrado. Este nivel define las medidas para las actividades y los productos a entregar. Los datos se recopilan en forma constante durante todo el proyecto. En consecuencia, el modelo de ciclo de vida del software puede comprenderse y analizarse de modo cuantitativo. Al cliente se le informa de los riesgos antes de que comience el proyecto y conoce las medidas utilizadas por la organización.

Nivel 5: Optimizado. Los datos de las mediciones se usan en un mecanismo de retroalimentación para mejorar al modelo de ciclo de vida del software a lo largo de la vida de la organización. El cliente, los gerentes del proyecto y los desarrolladores se comunican y trabajan juntos durante el desarrollo del proyecto.

Para poder medir la madurez de una organización, el SEI ha definido un conjunto de áreas de proceso claves (KPA, por sus siglas en inglés). Para alcanzar un nivel de madurez específico, la organización debe demostrar que maneja todas las áreas de proceso claves definidas para ese nivel. Algunas de estas áreas de proceso claves van más allá de las actividades definidas en el estándar IEEE 1074. La tabla 12-7 muestra la correspondencia entre el nivel de madurez y las áreas de proceso claves.

Tabla 12-7 Correspondencia entre los niveles de madurez de los procesos y las áreas de proceso claves.

Nivel de madurez	Área de proceso clave
Inicial	No es aplicable
Repetible	Enfoque: establecer los controles de administración de proyectos básicos. • **Administración de los requerimientos:** los requerimientos se establecen como línea base en un acuerdo de proyecto y se mantienen durante el proyecto. • **Planeación y seguimiento del proyecto:** se establece un plan de administración del proyecto de software al inicio del proyecto y se le da seguimiento durante la ejecución del proyecto. • **Administración del subcontratista:** la organización selecciona y administra de manera efectiva a subcontratistas de software calificados. • **Administración del aseguramiento de la calidad:** todos los productos a entregar y las actividades del proceso se revisan y auditan para verificar que se apeguen a los estándares y lineamientos adoptados por la organización. • **Administración de la configuración:** un conjunto de artículos de la administración de la configuración se define y mantiene durante todo el proyecto.
Definido	Enfoque: establecer una infraestructura que permita un solo modelo de ciclo de vida del software efectivo en todos los proyectos. • **Enfoque del proceso de la organización:** la organización tiene un equipo permanente para mantener y mejorar el ciclo de vida del software. • **Definición del proceso de la organización:** se usa un modelo de ciclo de vida del software estándar para todos los proyectos de la organización. Se usa una base de datos para la información y documentación relacionadas con el ciclo de vida del software. • **Programa de entrenamiento:** la organización identifica las necesidades de entrenamiento para proyectos específicos y desarrolla programas de entrenamiento. • **Administración integrada del software:** cada proyecto tiene la posibilidad de ajustar sus procesos específicos a partir del proceso estándar. • **Ingeniería de productos de software:** el software se construye de acuerdo con el ciclo de vida del software y con los métodos y herramientas definidos. • **Coordinación entre grupos:** los equipos del proyecto interactúan con los demás equipos para resolver los requerimientos y problemas. • **Revisiones entre iguales:** el desarrollador examina los productos a entregar en un nivel entre iguales para identificar defectos potenciales y áreas donde se necesiten cambios.

Tabla 12-7 Correspondencia entre los niveles de madurez de los procesos y las áreas de proceso claves.

Nivel de madurez	Área de proceso clave
Administrado	Enfoque: comprensión cuantitativa de los procesos y productos a entregar del ciclo de vida del software. • **Administración de procesos cuantitativa:** las medidas de productividad y calidad se definen y miden en forma constante a lo largo del proyecto. Es esencial que estos datos no se usen inmediatamente en el proyecto, en particular para valorar el desempeño de los desarrolladores, sino que se guarden en una base de datos para que puedan compararse con otros proyectos. • **Administración de la calidad:** la organización ha definido un conjunto de objetivos de calidad para los productos de software. Supervisa y ajusta los objetivos y productos para entregar productos de alta calidad al usuario.
Optimizado	Enfoque: dar seguimiento a los cambios de tecnología y procesos que pueden causar cambios en el modelo del sistema o en los productos a entregar, incluso durante todo un proyecto. • **Prevención de defectos:** se analizan las fallas de proyectos anteriores usando datos de la base de datos de mediciones. Si es necesario se toman acciones específicas para prevenir que vuelvan a suceder esos defectos. • **Administración del cambio de tecnología:** se investiga en forma constante a los facilitadores de tecnología y a las innovaciones, y se comparten en toda la organización. • **Administración del cambio de procesos:** se refina y cambia en forma constante el proceso del software para manejar las ineficiencias identificadas por las mediciones del proceso del software. El cambio constante es la norma y no la excepción.

12.4 Modelos del ciclo de vida

La figura 12-7 muestra el flujo de información entre los procesos en el estándar IEEE 1074. La complejidad del estándar es significativa, como puede verse por las muchas dependencias que hay entre los procesos. Cada asociación representa un producto de trabajo generado por un proceso y consumido por otro proceso. Cada asociación representa también un canal de comunicación formal entre los participantes en el proyecto, soportado por el intercambio de documentos, modelos o código.

Cuando se selecciona un modelo de ciclo de vida el modelador tiene que abordar dos cuestiones: ¿es necesario modelar todas estas dependencias? y ¿en qué orden deben calendarizarse? Esto es, ¿cuáles actividades pueden suprimirse? y ¿en qué orden deben realizarse y administrarse las actividades restantes para entregar un sistema de alta calidad dentro del presupuesto y a tiempo mientras suceden cambios durante todo el proyecto?

No hay una respuesta única a estas preguntas. Primero, diferentes proyectos requieren diferentes procesos y diferentes dependencias. Por ejemplo, si el dominio de aplicación es bien conocido, como en el caso de un proyecto de reingeniería de negocios, el ordenamiento de las actividades de desarrollo puede ser secuencial, en especial si los desarrolladores sólo requieren entrenamiento mínimo. Un proyecto primero en su tipo puede requerir de una elaboración considerable de prototipos y, en ese caso, todos los procesos de desarrollo deben ejecutarse en forma concurrente. Esto, a su vez, hace que los procesos de administración y administración de la configuración sean mucho más complejos.

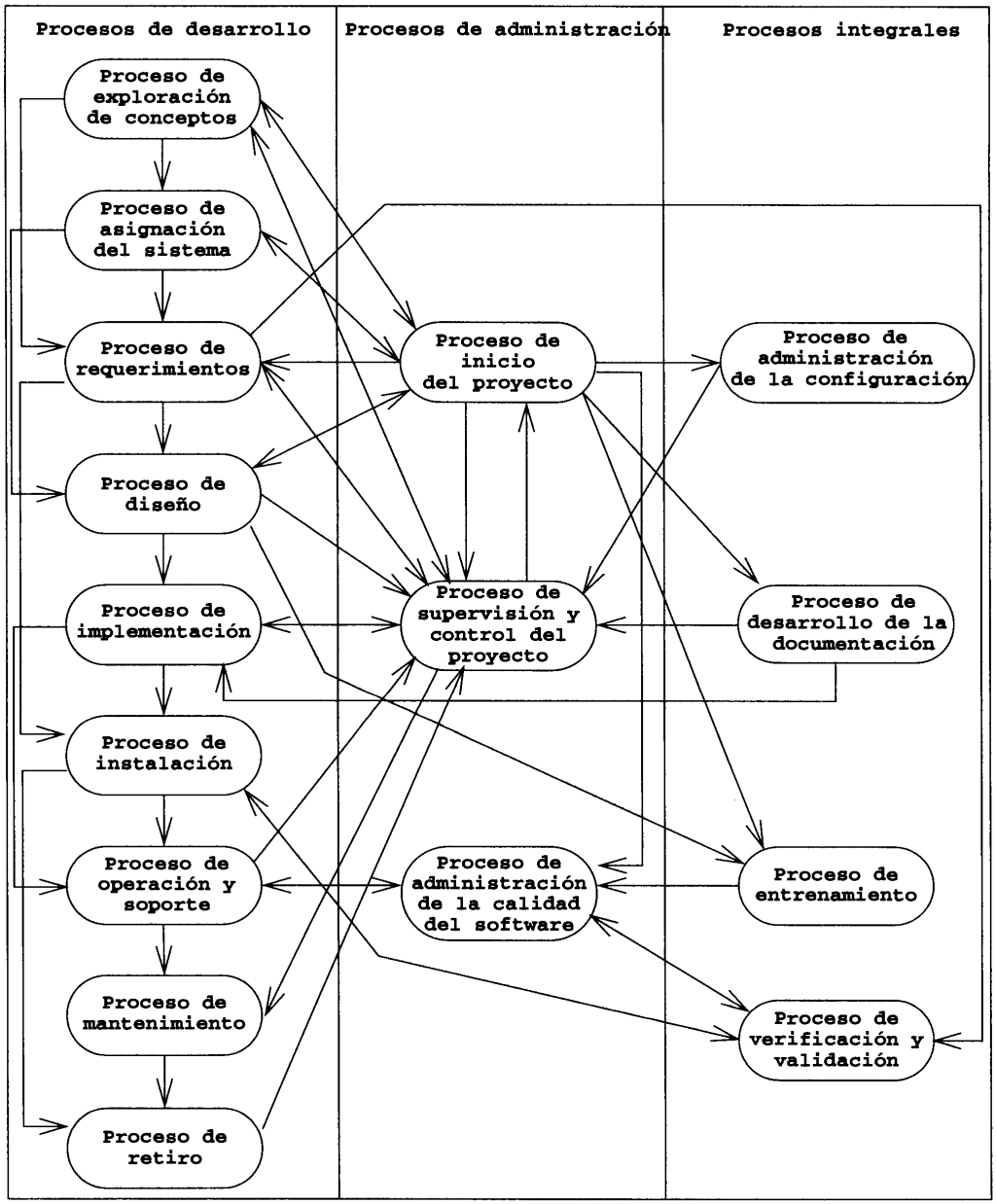


Figura 12-7 Interrelaciones de procesos en el estándar IEEE 1074 (diagrama de actividad UML, adaptado de [IEEE Std. 1074-1995]). Como lo sugiere esta ilustración, las dependencias entre procesos y actividades son complejas y rara vez permiten una ejecución secuencial de los procesos.

La selección del modelo del ciclo de vida también depende del modelo de sistema. La reingeniería de un sistema de software con casos de uso y modelos de objetos existentes requiere un conjunto de actividades diferentes que la construcción de un sistema a partir de cero. En un proyecto con desarrolladores experimentados, soporte con buenas herramientas CASE y un tiempo de entrega justo, el gerente del proyecto puede seleccionar un modelo del ciclo de vida centrado en entidad en vez de uno centrado en actividad.

Por estas razones, el IEEE 1074 no dicta un modelo del ciclo de vida específico, sino que proporciona una plantilla que puede personalizarse de muchas maneras diferentes. En esta sección revisamos modelos del ciclo de vida seleccionados. La mayoría de estos modelos se enfocan exclusivamente en los procesos de desarrollo.

12.4.1 Modelo de cascada

El **modelo de cascada** lo describió por primera vez Royse [Royse, 1970]. El modelo de cascada es un modelo del ciclo de vida centrado en actividad que prescribe una ejecución secuencial de un subconjunto de los procesos de desarrollo y administrativos que se describieron en la sección anterior (figura 12-8).

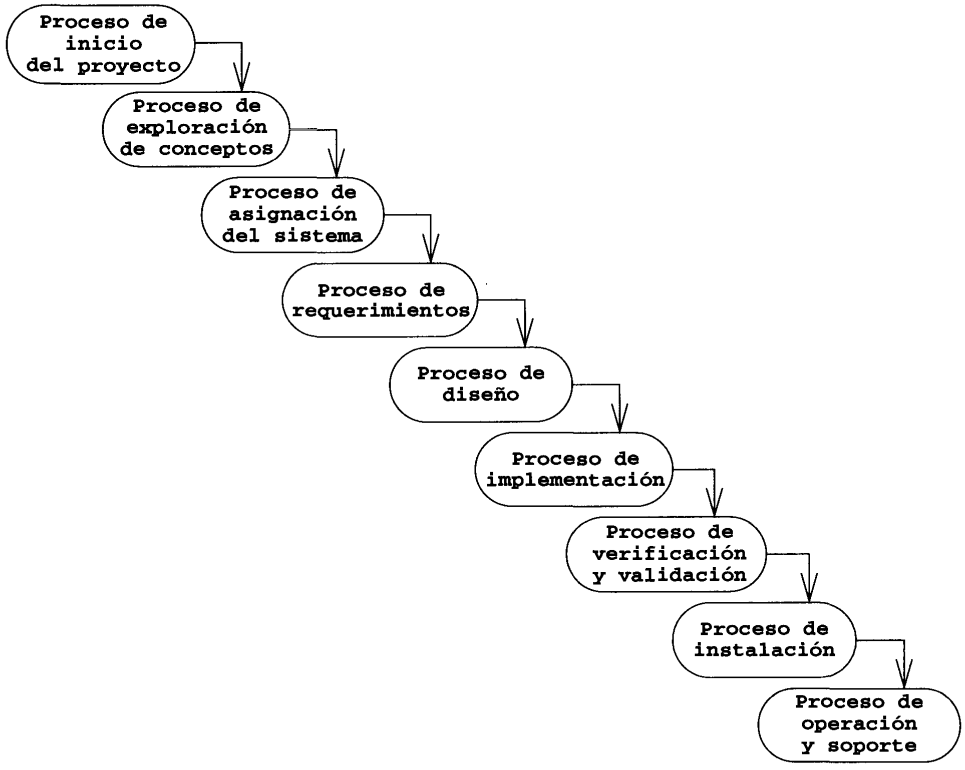


Figura 12-8 El modelo de cascada del desarrollo de software es una perspectiva centrada en actividad del ciclo de vida del software. Las actividades de desarrollo de software se realizan en secuencia (diagrama de actividad UML adaptado de [Royse, 1970] usando los nombres del IEEE 1074; se omiten los procesos de administración del proyecto y de desarrollo cruzado).

Todas las actividades de requerimientos se terminan antes de que comience la actividad de diseño del sistema. El objetivo es nunca regresar una vez que se termina una actividad. La característica principal de este modelo es la actividad de verificación constante (llamada "paso de verificación" por Royse) que asegura que cada actividad de desarrollo no introduzca requerimientos no deseados o elimine los obligatorios. Este modelo proporciona una visión simple (o hasta simplista) del desarrollo de software que mide el avance por el número de tareas que se han terminado. El modelo asume que el desarrollo de software puede calendarizarse como un proceso gradual que transforma en código las necesidades del usuario.

La figura 12-9 muestra un modelo de cascada ampliamente usado, el modelo del ciclo de vida estándar DOD 2167A.

La característica principal de este modelo es que cada actividad de desarrollo es seguida por una revisión. El punto inicial de este modelo es la actividad de análisis de requerimientos del sistema, cuyo objetivo es generar requerimientos del sistema no ambiguos. La primera revisión es la *Revisión de requerimientos del sistema*, durante la cual se revisan la suficiencia, consistencia y claridad de los requerimientos. Los requerimientos del sistema son la base para la actividad de diseño del sistema, la cual genera el diseño del sistema. El diseño del sistema se revisa durante la actividad de *Revisión del diseño del sistema.*

El diseño del sistema se toma como base una vez que se termina en forma satisfactoria la revisión del diseño del sistema. La línea base funcional sirve como punto de partida para el análisis de los requerimientos de software, que crea los requerimientos de software. Los requerimientos de software se revisan luego y se hacen línea base antes de que sirvan como base para la implementación. La implementación se inicia con el diseño preliminar, seguido por la actividad de diseño detallado. Una revisión importante es la revisión del diseño crítico (CDR, por sus siglas en inglés). La codificación no empieza antes de que la CDR se termine en forma satisfactoria.

12.4.2 Modelo V

El **modelo V** es una variación del modelo de cascada que hace explícita la dependencia entre las actividades de desarrollo y las de verificación. La diferencia entre el modelo de cascada y el V es que este último hace explícita la noción de nivel de abstracción. Todas las actividades, desde los requerimientos hasta la implementación, se enfocan en la construcción de una representación del sistema cada vez más detallada, mientras que todas las actividades desde la implementación hasta la operación se enfocan en la validación del sistema.

Los niveles de abstracción más elevados del modelo V manejan los requerimientos desde el punto de vista de la obtención y operación. La parte media del modelo V se enfoca en establecer la correspondencia entre la comprensión del problema y la arquitectura del software. El nivel más bajo del modelo V se enfoca en detalles como el ensamble de los componentes del software y la codificación de otros nuevos. Por ejemplo, el objetivo de la actividad *Prueba unitaria* es validar la unidad contra su descripción en el diseño detallado. La actividad *Integración y prueba de componentes* valida los componentes funcionales contra el diseño preliminar (o de alto nivel).

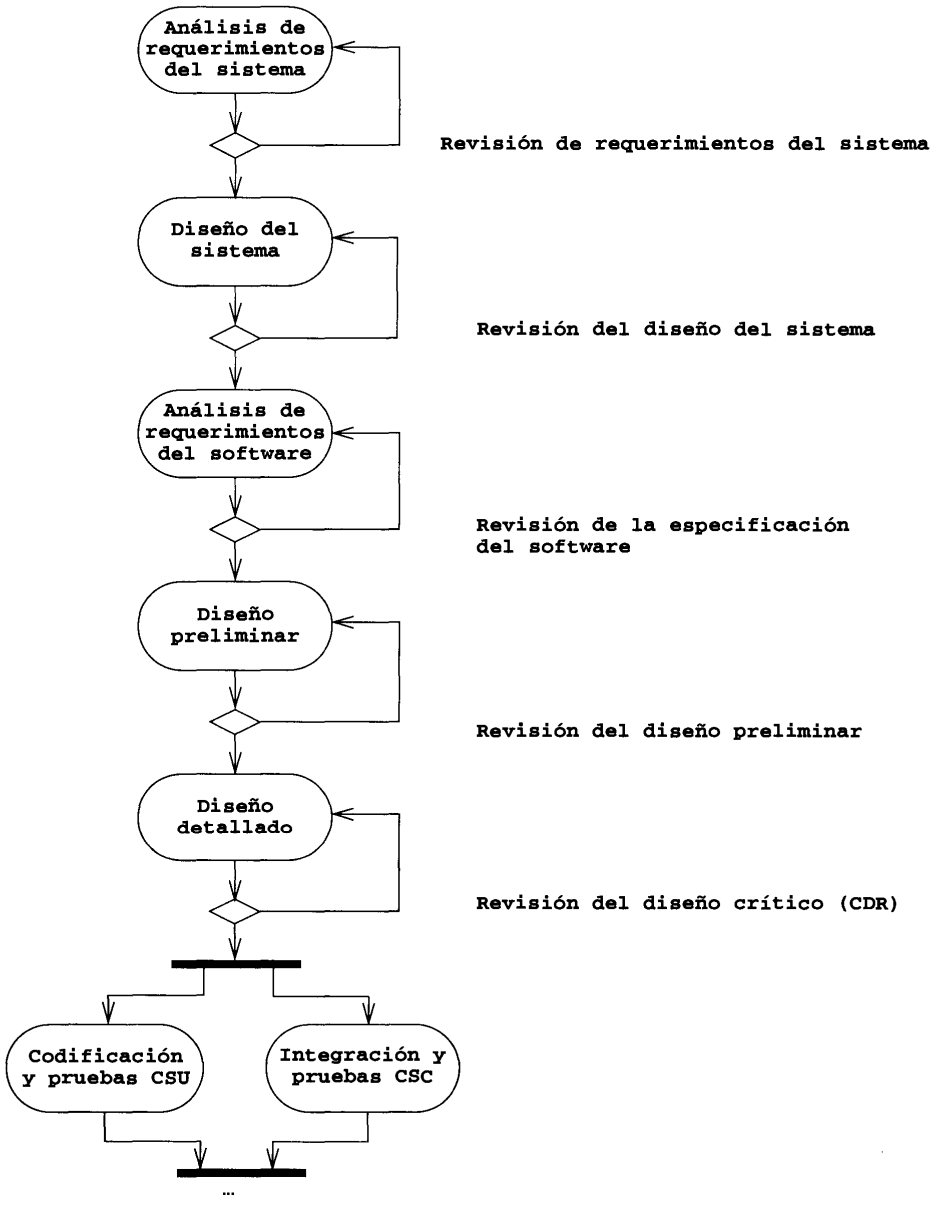


Figura 12-9 Modelo de cascada para el estándar DOD 2167A (diagrama de actividad UML). Observe que se usan las actividades específicas del DOD en vez de las actividades del IEEE 1074. Los puntos de decisión indican revisiones: la actividad subsiguiente se inicia sólo si la revisión es satisfactoria.

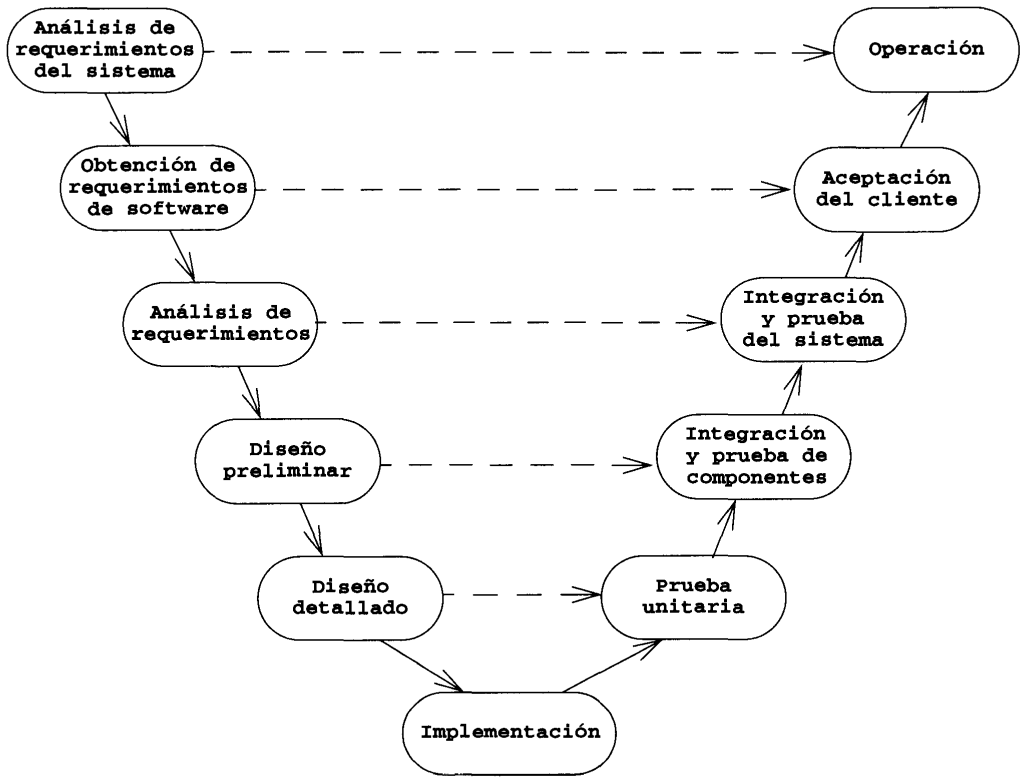


Figura 12-10 Modelo V de desarrollo de software (diagrama de actividad UML, adaptado de [Jensen y Tonies, 1979]). El flujo de objetos horizontal indica el flujo de información entre las actividades del mismo nivel de abstracción (por ejemplo, los resultados del análisis de requerimientos del sistema se validan durante la integración y prueba del sistema). Se conservó la disposición de las actividades en forma de V para reflejar el trazo original. Sin embargo, la disposición de las actividades no tiene semántica en UML.

En muchos aspectos, el modelo de cascada y sus variantes son abstracciones simplistas del proceso de desarrollo de software. La debilidad de estos modelos es que asumen que después de que una actividad se termina y se revisa, el producto de trabajo asociado puede tomarse como línea base. Tal modelo idealizado sólo es apropiado si la especificación de los requerimientos tiene un alto nivel de garantía y no cambia durante el desarrollo. En la práctica, el desarrollo de sistemas rara vez se apega a este modelo ideal. Los cambios durante una actividad a menudo requieren que se revise el trabajo de una actividad anterior. La figura 12-10 muestra el modelo V.

12.4.3 El modelo espiral de Boehm's

El **modelo espiral de Boehm** [Boehm, 1987] es un modelo de ciclo de vida centrado en actividad que se inventó para resolver la causa de la debilidad del modelo de cascada, en particular para acomodar los cambios poco frecuentes durante el desarrollo del software. Se basa en las mismas actividades que el modelo de cascada, pero añade varias tareas, como la administración del riesgo, la reutilización y la elaboración de prototipos, a cada actividad. A estas actividades extendidas se les llama **ciclos** o **rondas**.

El modelo espiral se enfoca en abordar los riesgos en forma incremental, de acuerdo a su prioridad. Cada ronda está compuesta por cuatro fases (figura 12-11). Durante la primera fase (cuadrante superior izquierdo) los desarrolladores exploran alternativas, definen restricciones e identifican objetivos. Durante la segunda fase (cuadrante superior derecho) los desarrolladores manejan los riesgos asociados con las soluciones definidas durante la primera fase. Durante la tercera fase (cuadrante inferior derecho) los desarrolladores realizan y validan un prototipo o la parte del sistema asociada con los riesgos que se tratan en esta ronda. La cuarta fase (cuadrante inferior izquierdo) se enfoca en la planeación de la siguiente ronda con base en los resultados producidos en la ronda actual. La última fase de la ronda se realiza, por lo general, como una revisión que involucra a los participantes en el proyecto, incluyendo a los desarrolladores, clientes y usuarios. Esta revisión abarca los productos desarrollados durante las rondas anteriores y actual y los planes para la siguiente ronda. El modelo espiral de Boehm distingue entre las siguientes rondas: *Conceptos de operación*, *Requerimientos de software*, *Diseño de productos de software*, *Diseño detallado*, *Código*, *Prueba unitaria*, *Integración y pruebas*, *Prueba de aceptación*, *Implementación*.[2]

Cada ronda sigue el modelo de cascada e incluye las siguientes actividades:

1. Determinar objetivos
2. Especificar restricciones
3. Generar alternativas
4. Identificar los riesgos
5. Resolver los riesgos
6. Desarrollar y verificar el producto del siguiente nivel
7. Planear

Las dos primeras actividades definen el problema que se trata en el ciclo actual. La tercera actividad, *Generar alternativas*, define el espacio de solución. Las actividades *Identificar riesgos* y *Resolver riesgos* sirven para identificar problemas futuros que pueden causar costos elevados o la cancelación del proyecto. La actividad *Desarrollar y verificar el producto del siguiente nivel* es la realización del ciclo. La actividad *Planear* es una actividad administrativa que prepara el siguiente ciclo.

2. Observe que la figura ilustra sólo las tres primeras actividades (*Conceptos de operación, Requerimientos de software y Diseño de productos de software).* Las rondas para las actividades restantes, *Diseño detallado, Código, Prueba unitaria, Integración y pruebas, y Prueba* de aceptación, no se muestran en forma detallada sino sólo como bloques al final de la última capa de la espiral.

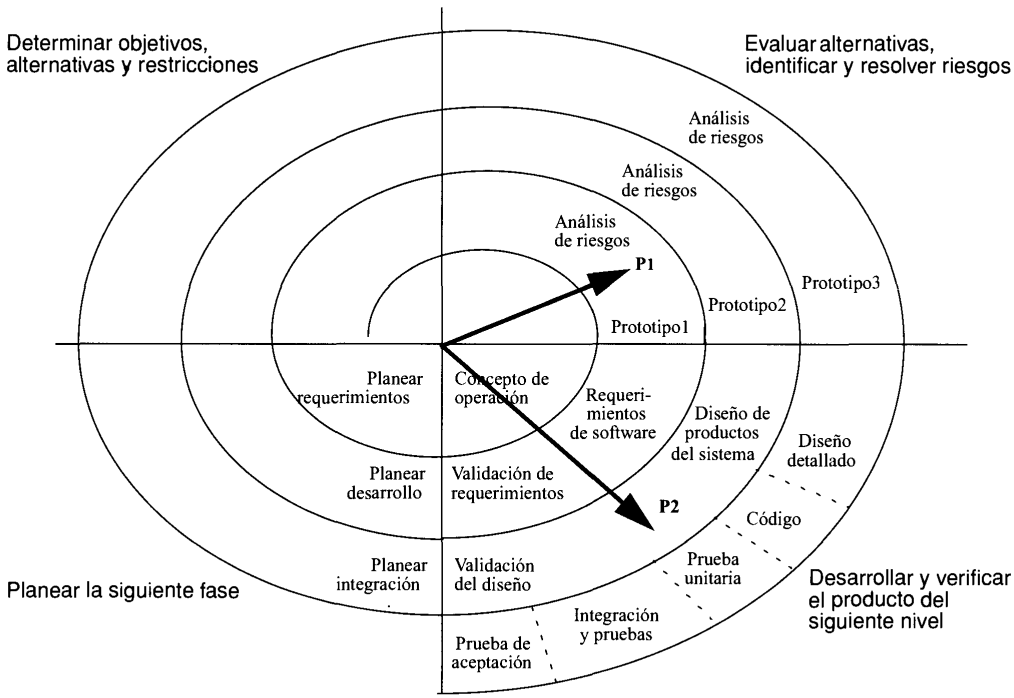


Figura 12-11 Modelo espiral de Boehm (adaptado de [Boehm, 1987]). La distancia con respecto al origen representa el costo acumulado del proyecto. El ángulo con respecto a la horizontal representa el tipo de actividad. Por ejemplo, el proyecto P1 está actualmente en la actividad de análisis de riesgos asociados con los requerimientos de software. El proyecto P2 está en el desarrollo del diseño de productos del sistema.

Estas rondas pueden verse en un sistema de coordenadas polares, como se muestra en la figura 12-11. La primera ronda, *Conceptos de operación*, se inicia en el cuadrante superior izquierdo. Las rondas subsiguientes se representan como capas adicionales de la espiral. La notación permite que se determine con facilidad el estado del proyecto a lo largo del tiempo. La distancia con respecto al origen es el costo acumulado del proyecto. La coordenada angular indica el avance logrado en cada fase.

12.4.4 Modelo de diente de sierra

Los modelos de ciclo de vida que hemos descrito hasta ahora enfatizan la administración de los desarrolladores de software. No tratan las necesidades del cliente o los usuarios. El modelo de cascada, el modelo V y el modelo espiral asumen que los requerimientos de software no cambiarán en forma drástica durante todo el proyecto y que es suficiente mostrar el avance de acuerdo al plan de desarrollo. La desventaja de este enfoque es que el cliente y el usuario no ven un sistema en ejecución sino hasta antes de la prueba de aceptación del cliente y, por tanto, no pueden corregir ningún problema de requerimientos.

Sin embargo, los usuarios y los implementadores tienen diferentes necesidades cuando tratan de comprender los sistemas de software. El **modelo de diente de sierra** [Rowen, 1990] trata de resolver estas discrepancias mostrando las percepciones del sistema por parte del usuario y el desarrollador de software en diferentes niveles de abstracción a lo largo del tiempo.

Al inicio del proyecto los desarrolladores y el cliente están en el mismo nivel de abstracción, es decir, los requerimientos del sistema como se describen en el enunciado del problema. Durante el desarrollo estos puntos de vista difieren un poco. El usuario permanece en el nivel de los requerimientos, mientras que los desarrolladores se enfocan en la factibilidad. El proceso de desarrollo de software tiene que asegurar que ambos puntos de vista se reúnan al final del proyecto. El modelo de diente de sierra logra este objetivo introduciendo nuevas actividades. El espacio entre estos niveles de abstracción corresponde al hueco entre la percepción del sistema que tiene el usuario y la que tiene el desarrollador. Para asegurar que se reúnan al final, durante el desarrollo se introducen puntos de revisión. Esto se logra, por lo general, haciendo que el cliente se involucre *en su nivel de abstracción*.

Por ejemplo, después de las fases de requerimientos y diseño del sistema de un sistema de software interactivo, los desarrolladores pueden elaborar prototipos de las secuencias de pantallas desde el punto de vista de los casos de uso que describen los requerimientos funcionales. Mostrando este prototipo ante el cliente puede evaluar muy pronto en el desarrollo si el prototipo satisface los requerimientos funcionales. Repitiendo este proceso varias veces durante el desarrollo, el gerente se asegura que las trayectorias se intersecten varias veces durante el desarrollo. Esto hace que sea mucho más probable que se encuentren al final del desarrollo.

El modelo de diente de sierra es un modelo V modificado que incluye estas intersecciones. Se le llama modelo de diente de sierra porque cada demostración del prototipo da como resultado un "diente". La punta de cada diente es una intersección con el nivel de abstracción del cliente. La figura 12-12 muestra el modelo de diente de sierra para un proyecto de desarrollo con dos prototipos, uno revolucionario y otro evolutivo.

A menudo, el prototipo revolucionario es ilustrativo, debido a que necesita construirse rápido para mostrar la funcionalidad del sistema. Hay pocos intentos de entregar este prototipo para uso en producción.[3] Sin importar qué tan realista sea, este prototipo todavía es sólo un modelo del sistema. Los escenarios que se muestren serán artificiales y representan solamente una pequeña fracción de la funcionalidad requerida. Los atajos que se toman para desarrollar una versión rápida serían una pesadilla de mantenimiento si se promoviera al prototipo para usarlo en producción.

El segundo prototipo es, por lo general, evolutivo. Se muestra tarde en el desarrollo, cuando ya se ha implementado alguna funcionalidad. La distinción principal entre los dos tipos de elaboración de prototipos es que el revolucionario no necesita un diseño general, mientras que el evolutivo sí lo necesita. Por lo general, es contraproducente insistir en una descomposición en subsistemas completa cuando se muestra el primer prototipo al usuario.

Las tareas y actividades del proceso de desarrollo de prototipos en el modelo de diente de sierra se muestran en la tabla 12-8.

3. Si el ambiente de elaboración de prototipos revolucionario es idéntico al ambiente de desarrollo, es posible la reutilización de algunas partes del prototipo revolucionario durante la implementación. Pero la reutilización no es el objetivo cuando se produce un prototipo revolucionario.

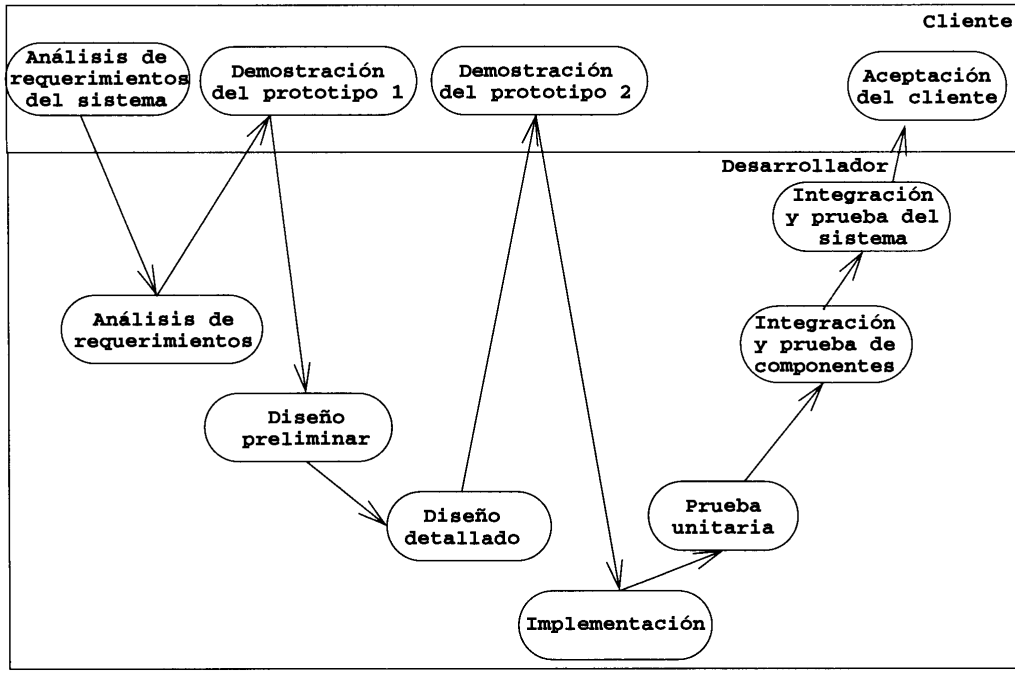


Figura 12-12 Modelo de diente de sierra con dos demostraciones de prototipos (diagrama de actividad UML). El carril `Cliente` encierra las actividades que son visibles para el cliente, mientras que el carril `Desarrollador` encierra las actividades que están en un nivel de abstracción menor.

Tabla 12-8 Actividades del proceso de desarrollo de prototipo ("diente") en el modelo de diente de sierra.

Actividad	Tarea
Preparación	Seleccionar el ambiente de elaboración de prototipos Seleccionar la funcionalidad que se va a demostrar Desarrollar el prototipo Desarrollar la agenda para la demostración del prototipo Notificar al cliente y revisar la agenda
Demostración	Instalar el prototipo Demostrar la funcionalidad seleccionada Registrar las minutas
Evaluación de la retroalimentación	Revisar las minutas de la demostración Identificar problemas Discutir problemas
Corrección del proyecto	Resolver asuntos pendientes

12.4.5 Modelo de diente de tiburón

El **modelo de diente de tiburón** es un refinamiento del modelo de diente de sierra. Además de las demostraciones al cliente también se introducen revisiones y demostraciones para la gerencia.

Por ejemplo, los dientes grandes pueden incluir un prototipo funcional y una maqueta de interfaz de usuario. Lo primero demuestra la factibilidad de las funciones que se implementarán en el sistema, mientras que lo segundo ilustra la disposición (o apariencia) de la interfaz de usuario. Los dientes pequeños pueden incluir un prototipo de la integración del sistema que muestra la interacción entre los componentes del sistema. Tal prototipo de integración puede construirse tan pronto como se seleccionan los componentes, y no necesita implementar ninguna funcionalidad. El cliente no es un buen destino para la demostración de un prototipo de integración del sistema. En general, el gerente del proyecto es la audiencia de destino para la demostración del prototipo de integración del sistema, y podemos decir que la revisión asociada es una revisión interna. El prototipo de integración del sistema puede mostrarse varias veces durante el proyecto, dando lugar cada una de las demostraciones a un diente pequeño. Para describir la demostración ante la gerencia añadimos otro carril al modelo de diente de sierra que muestra el nivel de comprensión del gerente del proyecto (vea la figura 12-13).

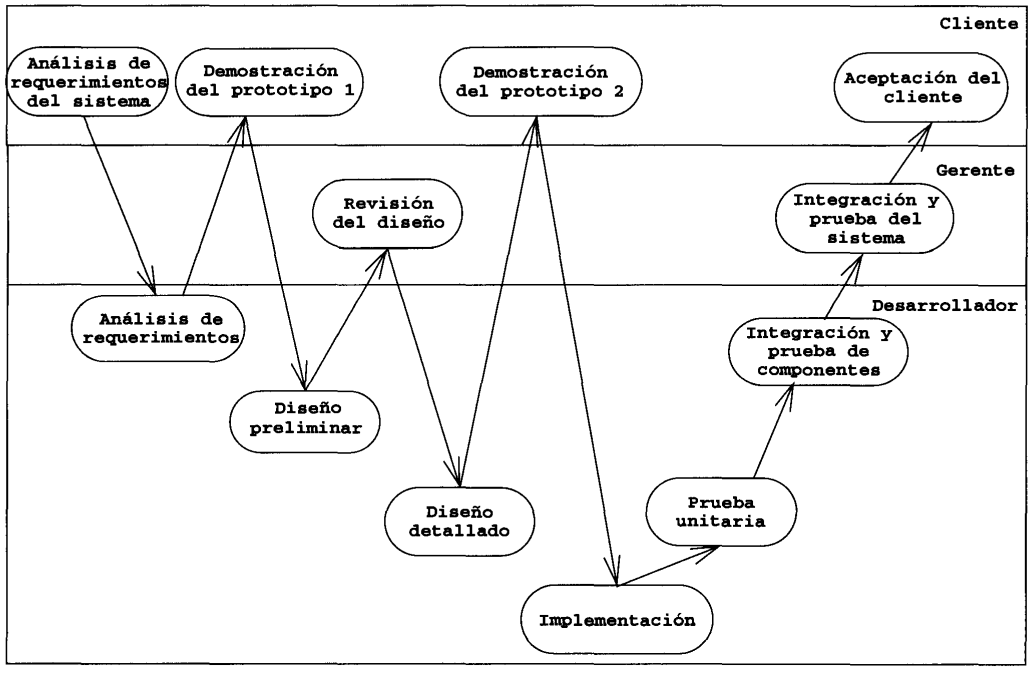


Figura 12-13 El modelo de diente de tiburón con dos demostraciones de prototipo y una revisión (diagrama de actividad UML, los niveles de abstracción están representados con carriles). Los dientes pequeños que llegan al carril del gerente son revisiones internas que involucran una demostración de prototipo hecha por los desarrolladores para el gerente del proyecto. Los dientes grandes que llegan al carril del cliente son demostraciones de prototipo para el cliente.

El carril del gerente se muestra entre los carriles del cliente y el desarrollador. La demostración del prototipo de integración del sistema es un diente que involucra a los desarrolladores y al gerente del proyecto. El modelo de diente de tiburón asume que el gerente del proyecto está interesado en los asuntos del diseño del sistema y el diseño de objetos y, por tanto, quiere alcanzar un nivel de comprensión del sistema más profundo que el del cliente, pero no quiere seguir a los desarrolladores hasta los niveles de detalle más profundos del sistema. Los dientes pequeños son revisiones internas que involucran una demostración de prototipo para el gerente del proyecto. Los dientes grandes son demostraciones de prototipos para el cliente.

12.4.6 Proceso de desarrollo de software unificado

El **proceso de desarrollo de software unificado** (llamado también **proceso unificado**) es un modelo de ciclo de vida propuesto por Booch, Jacobson y Rumbaugh [Jacobson *et al.*, 1999]. En forma similar al modelo espiral de Boehm, un proyecto consta de varios ciclos y cada uno de ellos termina con la entrega de un producto al cliente. Cada ciclo consta de cuatro fases: *Comienzo o inicio*, *Elaboración*, *Construcción* y *Transición*. Cada fase consta de varias iteraciones. Cada iteración trata un conjunto de casos de uso relacionados o mitiga algunos de los riesgos identificados al inicio de la iteración.

La fase de comienzo o inicio corresponde a la actividad *Exploración de conceptos* del IEEE 1074. Durante esta fase se define una necesidad o idea y se evalúa su factibilidad. La fase de elaboración corresponde al proceso de inicio del proyecto, y durante ella se planea el proyecto, se define el sistema y se le asignan recursos. La fase de construcción corresponde a los procesos de desarrollo. La fase de transición corresponde a los procesos de instalación y posdesarrollo.

El proceso unificado enfatiza el escalonamiento de recursos, un aspecto del desarrollo que no capturan los demás modelos de ciclo de vida. El proceso unificado asume que las actividades *Requerimientos*, *Análisis*, *Diseño*, *Implementación* y *Pruebas* participan en cada una de las iteraciones. Sin embargo, estas actividades tienen diferentes necesidades específicas de la fase. Por ejemplo, durante la fase de elaboración se asigna la mayoría de los recursos a las actividades de requerimientos y análisis. Durante la fase de construcción disminuyen los requerimientos de recursos para las actividades de recursos y análisis, pero se asignan más recursos a las actividades de diseño e implementación.

La figura 12-14 muestra una vista centrada en entidad del modelo de proceso unificado como un conjunto de modelos. Los requerimientos se capturan en el modelo de casos de uso. El modelo de análisis describe el sistema como un conjunto de clases. El modelo de diseño define la estructura del sistema como un conjunto de subsistemas e interfaces, y el modelo de organización define la distribución. El modelo de implementación establece la correspondencia entre las clases y los componentes, y el modelo de pruebas verifica que el sistema ejecutable proporcione la funcionalidad descrita en el modelo de casos de uso.

Todos los modelos están relacionados entre sí mediante dependencias de rastreabilidad. Un elemento del modelo puede rastrearse, al menos, hacia un elemento de un modelo asociado. Por ejemplo, todos los casos de uso tienen una relación de rastreabilidad con, al menos, una clase del modelo de análisis. La rastreabilidad nos permite comprender el efecto del cambio en un modelo sobre otros modelos y, en particular, nos permite proporcionar rastreabilidad en los requerimientos.

El mantenimiento de las dependencias entre los modelos de la figura 12-14 puede realizarse de formas diferentes. Durante la **ingeniería hacia delante** los modelos de análisis y diseño se establecen a partir del modelo de casos de uso, y los modelos de implementación y pruebas se generan luego a partir de estos modelos. Durante la **ingeniería inversa** los modelos de análisis y diseño se extraen o actualizan mediante el código existente. La **ingeniería de viaje redondo** es una combinación de ingeniería inversa y hacia delante. Permite que el desarrollador cambie entre estos dos modos de desarrollo en cualquier momento, dependiendo de cuál modelo esté sufriendo la mayor cantidad de cambios.

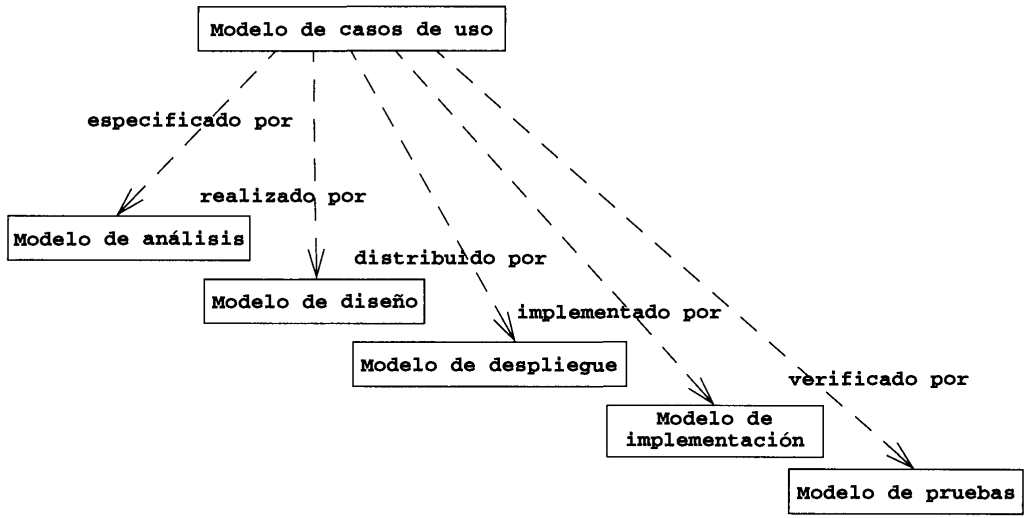


Figura 12-14 Vista del ciclo de vida centrado en entidad de los modelos del proceso unificado (diagrama de clase UML, las dependencias muestran la rastreabilidad). Hay dependencias entre todos los modelos. Sólo se muestran las dependencias entre el modelo de casos de uso y los demás modelos [Jacobson *et al.*, 1999].

12.4.7 Modelo de ciclo de vida basado en problemas

Si el tiempo entre cambios es significativamente más pequeño que la duración de una actividad, y si los cambios pueden ocurrir tanto en el dominio de aplicación como en el de solución, los modelos en cascada y en espiral presentan problemas. Por ejemplo, supongamos que el equipo de desarrollo selecciona la plataforma de hardware y que una nueva plataforma aparece cada tres o cuatro años. Mientras el tiempo del proyecto sea significativamente menor a tres años, el conocimiento del equipo de desarrollo al inicio del proyecto es suficiente, por lo general, para seleccionar una plataforma una vez durante el diseño del sistema. Pero si una nueva plataforma aparece cada tres o cuatro meses, y la duración del proyecto también es de tres o cuatro meses, hay una probabilidad muy alta de que las decisiones sobre la plataforma necesitarán reconsiderarse durante el proyecto.

En esta sección describimos al modelo de ciclo de vida centrado en entidad llamado **modelo de ciclo de vida basado en problemas**, que está orientado al manejo de los cambios frecuentes.

Este modelo se basa en la fundamentación que hay tras el sistema como un modelo de problemas (vea el capítulo 8, *Administración de la fundamentación*). Cada proyecto comienza con un conjunto de problemas. Si el proyecto empieza a partir de cero, estos problemas se trazan a partir de la experiencia del gerente del proyecto o de una plantilla estándar. En un proyecto de reingeniería o de interfaz, los problemas pueden estar disponibles a partir del modelo de problemas del proyecto anterior. Si el proyecto tiene una larga historia, la fundamentación debe estar bien poblada. Ejemplos de problemas son "¿cómo formamos los equipos iniciales?", "¿debe tener acceso el mecánico a la información específica del manejador?", "¿cuál es el middleware apropiado?", "qué arquitectura de software debemos usar?" y "¿cuál debemos usar como lenguaje de implementación?" Todos los problemas están guardados en una base de problemas a la que tienen acceso todos los participantes en el proyecto.

El estado de un problema puede ser abierto o cerrado. Un problema cerrado es aquel que ha sido resuelto. Por ejemplo, un problema cerrado puede ser una decisión acerca de la plataforma en la que debe ejecutar el sistema (por ejemplo, Solaris). Sin embargo, los problemas cerrados pueden volver a abrirse si suceden cambios en los dominios de aplicación o de solución. Por ejemplo, si necesitamos dar soporte a plataformas adicionales (por ejemplo, Linux y Windows NT), volvemos a abrir el problema, volvemos a evaluar las alternativas y proporcionamos nuevas resoluciones. Los problemas abiertos se resuelven mediante discusión y negociación entre los participantes en el proyecto (vea el capítulo 3, *Comunicación de proyectos*). Un problema `p2` depende de otro problema `p1` si la resolución de `p1` restringe las alternativas disponibles para `p2`. El rastreo de las dependencias entre problemas permite a los desarrolladores valorar el impacto de revisar un problema dado. La base de problemas también rastrea las dependencias entre problemas. La figura 12-15 muestra una instantánea de la base de problemas de un proyecto, mostrando el estado del problema y las dependencias.

Puede establecerse la correspondencia de los problemas con las actividades de los modelos de ciclo de vida descritos antes. Por ejemplo, supongamos que las actividades *Planeación* y *Diseño del sistema* son parte del modelo de ciclo de vida seleccionado. La cuestión "¿cómo ajustamos los

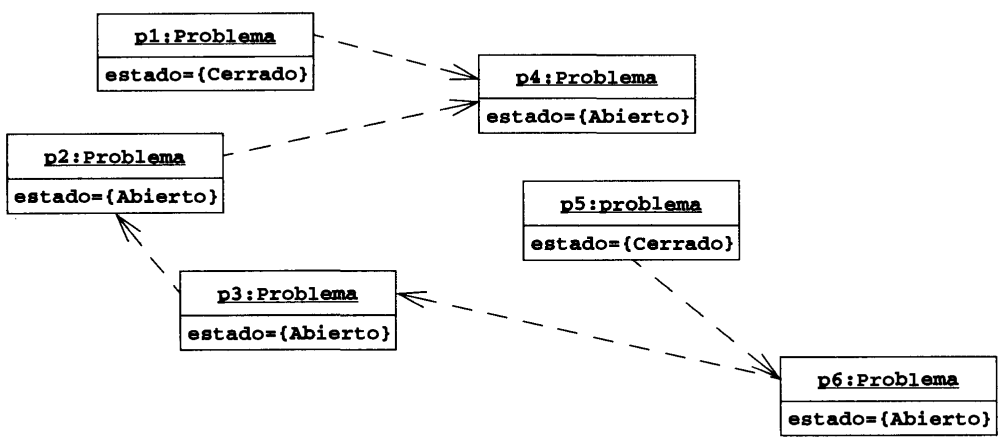


Figura 12-15 Instantánea de una base de problemas de proyecto (diagrama de objetos UML). Los problemas `p1` y `p5` ya se han resuelto, mientras que todos los demás todavía están abiertos. Las dependencias entre problemas indican que la resolución de un problema puede restringir las alternativas para los problemas dependientes.

equipos iniciales?" puede clasificarse como un problema de planeación, y "¿qué arquitectura debemos usar?" puede clasificarse como un problema de diseño del sistema. El estado de los problemas puede usarse para el seguimiento del estado de cada actividad. Si todavía están abiertos algunos problemas del diseño del sistema, entonces la actividad *Diseño del sistema* no se ha terminado. Los modelos de ciclo de vida que describimos antes pueden verse entonces como casos especiales del modelo basado en problemas. En el modelo en cascada, por ejemplo, los desarrolladores resuelven por completo los problemas asociados con una actividad antes de pasar a la siguiente. La figura 12-16 muestra el estado de un proyecto durante el *Diseño del sistema*.

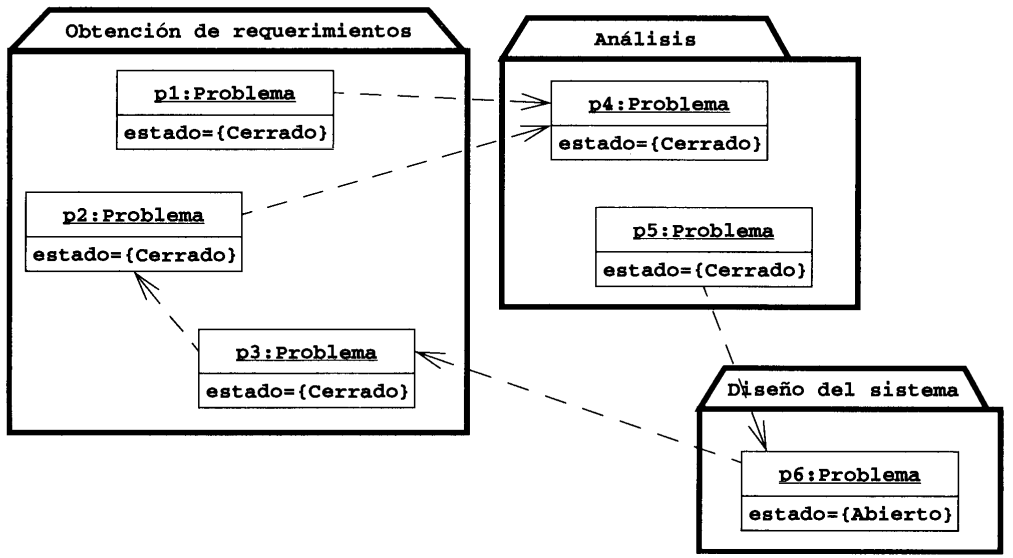


Figura 12-16 El modelo en cascada como un caso especial del modelo de ciclo de vida basado en problemas (diagrama de objetos UML). Todos los problemas que pertenecen a la misma categoría de problemas están contenidos en el mismo paquete UML. En el estado del proyecto que se muestra en la figura ya se han cerrado todos los problemas de la obtención de requerimientos y el análisis; esto es, se han terminado las actividades de obtención de requerimientos y de análisis.

En el modelo espiral de Boehm los riesgos corresponden a problemas que están evaluados y se vuelven a abrir al inicio de cada ronda. Los problemas se resuelven en orden de su prioridad, como se definen durante el análisis de riesgos. Sin embargo, tome en cuenta que "problema" es un concepto más general que "riesgo". Por ejemplo, el problema "¿cuál modelo de control de acceso debemos usar?" es un problema de diseño y no un riesgo.

En el caso general (figura 12-17), todas las actividades todavía pueden tener problemas abiertos asociados con ellas, lo que significa que todas las actividades necesitan administrarse en forma concurrente. El objetivo del gerente del proyecto es mantener la cantidad de problemas abiertos pequeña y manejable, sin imponer restricciones de tiempo o basadas en actividad sobre la resolución de los problemas. El uso de problemas y sus dependencias para administrar las actividades del ciclo de vida permite que todas las actividades del ciclo de vida se lleven a cabo en forma concurrente.

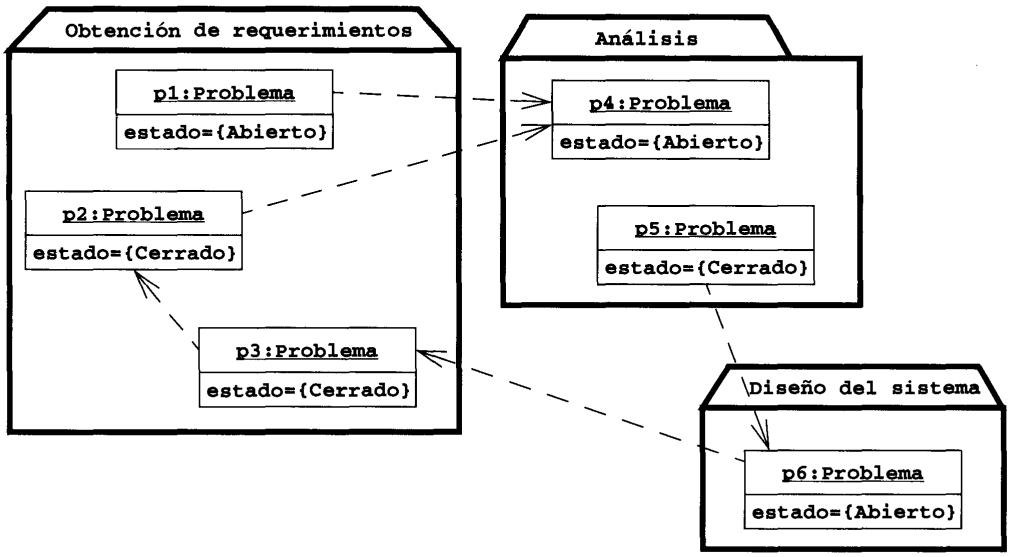


Figura 12-17 En un estado de proyecto complejo todas las actividades pueden tener todavía algunos asuntos abiertos, lo que significa que todas las actividades necesitan manejarse en forma concurrente.

12.5 Administración de las actividades y productos

En esta sección describimos los procesos del modelado del ciclo de vida para dos proyectos de ejemplo. El primer proyecto de ejemplo, un desarrollo de prototipo en un solo sitio durante cuatro meses, puede usarse para un curso de proyectos de un solo semestre. El segundo proyecto de ejemplo, un desarrollo de prototipo en dos sitios durante ocho meses, puede usarse para un curso de proyectos de dos semestres, en donde cada semestre sucede en una ubicación diferente.

12.5.1 Ejemplo 1: proyecto de un sitio durante cuatro meses

Objetivos y restricciones del proyecto

Un objetivo del proyecto es desarrollar un prototipo de demostración para un sistema de software. Un segundo objetivo del proyecto es exponer a sus participantes los métodos y herramientas de la ingeniería de software más recientes. Tiene cuarenta participantes y la mayoría de ellos nunca han trabajado juntos. Los desarrolladores y escritores técnicos involucrados en el proyecto tienen grandes conocimientos técnicos en la escritura de programas y documentación, pero ninguna experiencia de desarrollo a gran escala. El dominio de aplicación es nuevo para todos los participantes. La entrega del prototipo debe darse en cuatro meses. Tomando en cuenta el lapso tan pequeño, debe usarse contratación llana.

Actividades

Seleccionamos un modelo de ciclo de vida que consta de tres fases. Durante la primera fase, llamada predesarrollo, se formulan los requerimientos preliminares y se desarrolla una arquitectura

de software inicial con el propósito de asignar recursos para el resto del proyecto. La selección de infraestructura también se hace durante la fase preliminar. Sólo está involucrado un subconjunto de los participantes en el proyecto: el cliente, el gerente del proyecto y los instructores. Durante la segunda fase, llamada desarrollo, se construye un sistema que muestra los requerimientos medulares siguiendo un modelo de cascada. Todos los desarrolladores han sido asignados a un equipo y están trabajando en forma activa para la construcción del sistema. El objetivo de la fase de desarrollo es validar la arquitectura del sistema y exponer a los participantes a todos los aspectos del ciclo de vida. Durante la tercera fase, llamada posdesarrollo, se añade funcionalidad opcional al sistema. El objetivo de la tercera fase es extender la funcionalidad del sistema y, al mismo tiempo, controlar los riesgos. La funcionalidad se añade en todos los niveles de abstracción: los requerimientos, el diseño, la implementación y los casos de prueba se desarrollan en forma incremental. La fundamentación para este enfoque es proporcionar un sistema funcional a tiempo, en vez de entregar un sistema completo atrasado. La tabla 12-9 describe cada una de estas tres fases desde el punto de vista de objetivos y actividades y procesos IEEE 1074.

Tabla 12-9 Actividades para el ejemplo del proyecto 1.

Fase	Propósito	Actividades	Actividades IEEE 1074 correspondientes
Predesarrollo	Iniciar el proyecto	Inicio del proyecto Exploración de conceptos Diseño de alto nivel	3.1 Inicio del proyecto 4.1 Exploración de conceptos 4.2 Asignación del sistema
Desarrollo	Validar la arquitectura Entrenar participantes Demostrar requerimientos medulares Demostrar factibilidad	Administración del proyecto Obtención de requerimientos Análisis Revisión del análisis Diseño del sistema Revisión del diseño del sistema Diseño de objetos Revisión del diseño de objetos Implementación Prueba unitaria Prueba de integración del sistema Prueba del sistema (prueba alfa) Administración de la configuración Cursos prácticos	3.2 Supervisión y control del proyecto 5.1 Requerimientos 5.2 Diseño 5.2.7 Realizar diseño detallado 5.3 Implementación 7.1 Verificación y validación 7.2 Administración de la configuración del software 7.4 Entrenamiento
Posdesarrollo	Demostrar requerimientos opcionales Entregar prototipo Aceptación del cliente	Administración del proyecto Refinamiento Instalación del software Prueba de aceptación del cliente Prueba de campo (prueba beta) Administración de la configuración	3.2 Supervisión y control del proyecto 6.3 Mantenimiento 6.1.5 Instalación del software 6.1.6 Aceptación del software 7.1 Verificación y validación 7.2 Administración de la configuración del software

Observe que este ciclo de vida no está completo, y tampoco incluye actividades de operación y soporte, ya que el objetivo del proyecto de ejemplo es desarrollar un prototipo de demostración.

Productos de trabajo

El proyecto produce tres tipos de información: *información del sistema*, *información de administración* y *fundamentación*. Esta información está representada por tres tipos de modelos: los modelos de tarea representan información de administración, los modelos del sistema describen al sistema (requerimientos, análisis, diseño, implementación) y los modelos de problemas representan la fundamentación que hay detrás de las decisiones de diseño y de administración.

Los modelos de tareas son modelos de procesos detallados y específicos del proyecto. Los construye la administración durante el inicio del proyecto y se revisan a los largo del proyecto. Están documentados en el *Acuerdo del proyecto*, en el *Plan de administración de proyecto de software* y en el documento *Diseño de alto nivel* (tabla 12-10). En el capítulo 11, *Administración del proyecto*, describimos los modelos de tareas.

Tabla 12-10 Documentos de administración.

Documento	Propósito	Producido por
Enunciado del problema	Describe las necesidades, escenarios visionarios, sistema actual, sistema de destino, requerimientos y restricciones	Exploración de conceptos
Calendarización inicial	Describe los indicadores de avance principales	Inicio del proyecto
Diseño de alto nivel	Describe la arquitectura preliminar del sistema, equipos, restricciones, infraestructura de comunicación	Diseño de alto nivel
Plan de administración del proyecto de software	Control de los documentos para la administración del proyecto	Inicio del proyecto Supervisión y control del proyecto
Plan de administración de la configuración del software	Control de los documentos para las actividades de administración de la configuración del software	Administración de la configuración del software

Los modelos del sistema representan el sistema en diferentes niveles de abstracción y desde perspectivas diferentes. El modelo de casos de uso describe el sistema desde el punto de vista del usuario. El modelo de análisis describe los objetos de aplicación que manipulan el sistema. Los modelos de diseño de sistema y de objetos representan a los objetos de solución. Los modelos de casos de uso y análisis se documentan en el *Documento de análisis de requerimientos* (tabla 12-11). El modelo de diseño del sistema se documenta en el *Documento de diseño del sistema*. El modelo de diseño de objetos y los diagramas de secuencia se documentan en el *Documento de diseño de objetos*. En los capítulos 6 y 7 describimos el *Documento de diseño del sistema* y el *Documento de diseño de objetos*, respectivamente.

Tabla 12-11 Documentos del sistema.

Documento	Propósito	Producido por
Documento de análisis de requerimientos (RAD)	Describe los requerimientos funcionales y globales del sistema y también cuatro modelos: el modelo de casos de uso, el modelo de objetos, el modelo funcional y el modelo dinámico	Requerimientos
Manual de usuario	Describe el uso del sistema, con frecuencia en forma de curso práctico	Requerimientos
Documento de diseño del sistema (SDD)	Describe los objetivos de diseño, compromisos entre objetivos de diseño, la descomposición de alto nivel del sistema, identificación de concurrencia, plataformas de hardware/software, administración de datos, manejo de recursos globales, implementación del control del software y condiciones de frontera	Diseño del sistema
Documento de diseño de objetos (ODD)	Describe el sistema desde el punto de vista de modelos de objetos refinados, en particular las estructuras de datos y algoritmos seleccionados, así como las firmas completas de todos los métodos públicos. Este documento da como resultado la especificación detallada de cada clase usada por los programadores durante la fase de implementación	Diseño de objetos
Manual de pruebas	Describe la estrategia de las pruebas, las pruebas unitarias y de sistema realizadas en el sistema, junto con los resultados esperados y actuales	Pruebas
Manual del administrador	Describe los procedimientos administrativos para instalar, operar y desactivar el sistema. También contiene una lista de códigos de error, fallas y condiciones de terminación	Instalación del software

La fundamentación del sistema está representada con un modelo de problemas. Los desarrolladores se enfocan en el sistema, argumentan acerca de alternativas diferentes, tienen objetivos de diseño diferentes o basan muchas decisiones en su experiencia anterior. Los motivos que hay tras la captura de la fundamentación es facilitar la comunicación y hacer explícitos los objetivos de diseño y las restricciones a lo largo del proyecto. Los objetivos del sistema y las restricciones se describen en el documento de diseño del sistema. Los problemas se describen en el *Documento de asuntos* (tabla 12-12). En el capítulo 8, *Administración de la fundamentación*, describimos los modelos de problemas.

Tabla 12-12 Documentos de fundamentación.

Documento	Propósito	Producido por
Documento de problemas (ID)	Describe los problemas abiertos, opciones posibles, argumentos y sus resoluciones	Todos los procesos

12.5.2 Ejemplo 2: proyecto de ocho meses en dos sitios

Objetivos y restricciones del proyecto

En este ejemplo describimos un proyecto piloto con dos fases de desarrollo que suceden en dos sitios. El final de la primera fase de desarrollo se traslapa con el inicio de la segunda, permitiendo que suceda cierta transferencia de conocimiento entre ambos sitios. Nuevamente, la mayoría de los participantes no está involucrada con las fases de predesarrollo y posdesarrollo. Los participantes de un sitio todavía no han trabajado con los participantes del otro sitio antes de este proyecto. La entrega de un sistema prototipo debe darse en los ocho meses siguientes, con una demostración intermedia del prototipo al final de los primeros cuatro meses.

Actividades

Refinamos el ciclo de vida del primer proyecto de ejemplo para que incluya una segunda fase de desarrollo, y las actividades se enfocan en la transferencia de conocimiento de un sitio al otro (tabla 12-13). Ambas fases de desarrollo están estructuradas de la misma forma, de tal manera que pueden usarse los mismos documentos, métodos y procedimientos. La segunda fase de desarrollo es una iteración de mejoras que produce una versión refinada del *Documento de análisis de requerimientos*, del *Documento de diseño del sistema*, del *Documento de diseño de objetos* y del sistema prototipo. Un producto de trabajo adicional, llamado *Documento de análisis de inventario*, se produce al inicio de la segunda fase de desarrollo para describir el estado actual del sistema. Incluye una lista de componentes, documentos y modelos que se han construido en la primera fase y una lista de problemas que describen dificultades que necesitan resolverse en esos elementos.

Transferencia de conocimiento entre sitios

El desarrollo de software distribuido tiene varias ventajas. Por ejemplo, un proyecto puede aprovechar combinaciones específicas de habilidades sin reubicar a los participantes dentro de la organización. A menudo, el sistema también se organizará en sitios diferentes, y el gerente del proyecto tal vez querrá que los desarrolladores estén geográficamente cercanos a varias comunidades de usuarios. Sin embargo, la distribución introduce nuevos retos, como la comunicación asíncrona y la transferencia de conocimientos de un grupo de participantes a otro.

En el proyecto de ejemplo 2, la transferencia de conocimientos necesita darse al final de la primera fase de desarrollo y al inicio de la segunda. Dado que hay una superposición mínima entre ambas fases, los participantes de diferentes sitios no se conocen. En consecuencia, no existen oportunidades para el intercambio espontáneo. En vez de ello, formalizamos esta transferencia de conocimiento con la actividad Análisis de inventario. Los participantes del segundo sitio se familiarizan con el sistema asistiendo primero, mediante videoconferencias, a una o más revisiones, como la revisión del diseño de objetos o la prueba de aceptación del cliente. Luego los participantes del segundo sitio aplican ingeniería inversa al primer sistema prototipo para encontrar inconsistencias y omisiones con la documentación que le acompaña. Algunos de los participantes del primer sitio permanecen en el proyecto de manera temporal para cerrar la primera fase de desarrollo y responder preguntas de los participantes del segundo sitio. El resultado del Análisis de inventario es el *Documento de análisis de inventario* que describe los componentes del primer prototipo y su estado.

Tabla 12-13 Actividades para el proyecto de ejemplo 2. Las diferencias con respecto al ejemplo 1 se muestran en *cursivas.*

Fase	Propósito	Actividades	Actividades IEEE 1072
Predesarrollo	Iniciar el proyecto	Inicio del proyecto Exploración de conceptos Diseño de alto nivel	3.1 Inicio del proyecto 4.1 Exploración de conceptos 4.2 Asignación del sistema
Primera fase de desarrollo	Validar la arquitectura Entrenar participantes del primer sitio Demostrar requerimientos medulares Demostrar factibilidad	Administración del proyecto Obtención de requerimientos Análisis Diseño del sistema Diseño de objetos Implementación Prueba unitaria Prueba de integración del sistema Prueba del sistema (prueba alfa) Administración de la configuración Conferencias y cursos prácticos	3.2 Supervisión y control del proyecto 5.1 Requerimientos 5.2 Diseño 5.2.7 Realizar diseño detallado 5.3 Implementación 7.1 Verificación y validación 7.2 Administración de la configuración del software 7.4 Entrenamiento
Segunda fase de desarrollo	*Mejorar la arquitectura* *Entrenar participantes del segundo sitio* *Demostrar extensibilidad* *Demostrar requerimientos deseables*	*Análisis del inventario* *Revisión del inventario* *Administración del proyecto* *Obtención de requerimientos* *Análisis* *Revisión del análisis de requerimientos* *Diseño del sistema* *Revisión del diseño del sistema* *Diseño de objetos* *Revisión del diseño de objetos* *Implementación* *Prueba unitaria* *Prueba de integración del sistema* *Prueba del sistema (prueba alfa)* *Administración de la configuración* *Cursos prácticos*	*3.2 Supervisión y control del proyecto* *5.1 Requerimientos* *5.2 Diseño* *5.2.7 Realizar diseño detallado* *5.3 Implementación* *7.1 Verificación y validación* *7.2 Administración de la configuración del software* *7.4 Entrenamiento*
Fase de posdesarrollo	Demostrar la funcionalidad completa Entregar prototipo Aceptación del cliente	Administración del proyecto Refinamiento Instalación del software Prueba de aceptación del cliente Prueba de campo (prueba beta) Administración de la configuración	3.2 Supervisión y control del proyecto 6.3 Mantenimiento 6.1.5 Instalación del software 6.1.6 Aceptación del software 7.1 Verificación y validación 7.2 Administración de la configuración del software

La cantidad de superposición entre ambas fases presenta un compromiso crítico de administración del proyecto. Una superposición corta impide que se transfiera suficiente conocimiento. Los participantes del segundo sitio tendrán que realizar reingeniería en la mayor parte del conocimiento sólo a partir de los productos de trabajo. Entre más grande sea la superposición tendrán mayores oportunidades para que se transmita el conocimiento en forma de preguntas y respuestas dirigidas. Sin embargo, si la segunda fase de desarrollo se inicia demasiado pronto, los participantes del segundo sitio interferirán con los participantes del primero, quienes todavía están en el proceso de creación del conocimiento que se va a transmitir.

Documentos

Los documentos creados en este ejemplo son los mismos que en el proyecto de ejemplo 1, además del *Documento de análisis del inventario* producido durante el Análisis del inventario (tabla 12-14). La diferencia principal entre estos proyectos es que los documentos técnicos primordiales (es decir, *Documento de análisis de requerimientos*, *Documento de diseño del sistema*, *Documento de diseño de objetos*, *Documento de problemas*) se revisan en forma considerable durante la segunda fase de desarrollo para tomar en cuenta las lecciones aprendidas durante la primera fase de desarrollo.

Tabla 12-14 Documentos de inventario.

Documento	Propósito	Producido por
Documento de análisis del inventario	Describe los componentes y modelos que se han realizado, su estado y los problemas asociados con ellos	Análisis del inventario

12.6 Ejercicios

1. Suponga que ha construido una herramienta CASE para el modelo de ciclo de vida estándar del IEEE 1074. Defina a los actores y casos de uso para el Proceso de diseño.
2. Adapte las figuras 12-1, 12-3 y 12-4 para producir un diagrama de clase UML integrado que represente las actividades y productos de trabajo tratados en este capítulo.
3. Suponga que el modelo de cascada que se muestra en la figura 12-8 se ha derivado del modelo estándar IEEE de la figura 12-7 durante la actividad Modelado del ciclo de vida. ¿Cuáles procesos y actividades se han omitido en el modelo de cascada?
4. Haga la correspondencia entre los nombres de las actividades del estándar DOD 2167A de la figura 12-9 y los nombres de las actividades del estándar IEEE 1074.
5. Vuelva a trazar el modelo espiral de Boehm de la figura 12-11 como un diagrama de actividad UML. Compare la legibilidad de la figura original con el diagrama de actividad.
6. Trace un diagrama de actividad UML que describa la dependencia entre las actividades de un ciclo de vida en donde los requerimientos, diseño, implementación, prueba y mantenimiento sucedan de manera concurrente. (A éste se le llama ciclo de vida evolutivo.)
7. Describa cómo pueden iniciarse las actividades de prueba mucho antes que las actividades de implementación. Explique por qué es deseable.

8. Suponga que es parte del comité IEEE que revisará el estándar IEEE 1074. Se le ha asignado la tarea del modelado de la comunicación como un proceso integral explícito. Haga un caso para las actividades que pertenecerían a este proceso.

Referencias

[Boehm, 1987] B. Boehm, "A spiral model of software development and enhancement", *Software Engineering Project Management*. págs. 128–142, 1987.

[Humphrey, 1989] W. Humphrey, *Managing the Software Process*. Addison-Wesley, Reading, MA, 1989.

[IEEE, 1997] *IEEE Standards Collection Software Engineering*. IEEE, Piscataway, NJ, 1997.

[IEEE Std. 1074-1995] *IEEE Standard for Developing Software Life Cycle Processes*, IEEE Computer Society, Nueva York, 1995, en [IEEE 1997].

[Jacobson *et al.*, 1999] I. Jacobson, G. Booch y J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[Jensen y Tonies, 1979] R. W. Jensen y C. C. Tonies, *Software Engineering*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[Paulk *et al.*, 1995] M. C. Paulk, C. V. Weber, y B. Curtis (eds.), *The Capability Maturity Mode: Guidelines for Improving the Software Process*. Addison-Wesley, Reading, MA, 1995.

[Rowen, 1990] R. B. Rowen, "Software project management under incomplete and ambiguous specifications", *IEEE Transactions on Engineering Management*, vol. 37, núm. 1, 1990.

[Royse, 1970] W. W. Royse, "Managing the development of large software systems", in *Tutorial: Software Engineering Project Management*, IEEE Computer Society, Washington, DC, págs. 118–127, 1970.

PARTE V
Apéndices

Patrones de diseño

Los patrones de diseño son soluciones parciales a problemas comunes, como la separación de una interfaz con respecto a varias implementaciones alternas, la envoltura de un conjunto de clases heredadas, la protección de quien llama con respecto a cambios asociados con plataformas específicas. Un patrón de diseño está compuesto por una pequeña cantidad de clases que, mediante delegación y herencia, proporcionan una solución robusta y modificable. Estas clases pueden adaptarse y refinarse para el sistema específico que se está construyendo. Además, los patrones de diseño proporcionan ejemplos de herencia y delegación.

Desde la publicación del primer libro sobre patrones de diseño para el software [Gamma *et al.*, 1994], se han propuesto muchos patrones adicionales para una amplia variedad de problemas, incluyendo análisis [Fowler, 1997], [Larman, 1998], diseño del sistema [Buschmann *et al.*, 1996], middleware [Mowbray y Malveau, 1997], modelado de procesos [Ambler, 1998], administración de la dependencia [Feiler *et al.*, 1998] y administración de la configuración [Brown *et al.*, 1999]. El término mismo se ha convertido en una palabra rimbombante a la que con frecuencia se le atribuyen muchas definiciones diferentes. En este libro nos enfocamos sólo en el catálogo original de patrones de diseño, ya que proporcionan un conjunto conciso de soluciones elegantes para muchos problemas comunes. Este apéndice resume los patrones de diseño que usamos en este libro. Para cada patrón proporcionamos indicadores de los ejemplos de este libro que los utilizan. Nuestro objetivo es proporcionar una referencia rápida que también puede usarse como índice. Suponemos que el lector tiene un conocimiento básico de los patrones de diseño, los conceptos orientados a objetos y los diagramas de clase UML.

A.1 Fábrica abstracta: encapsulado de plataformas

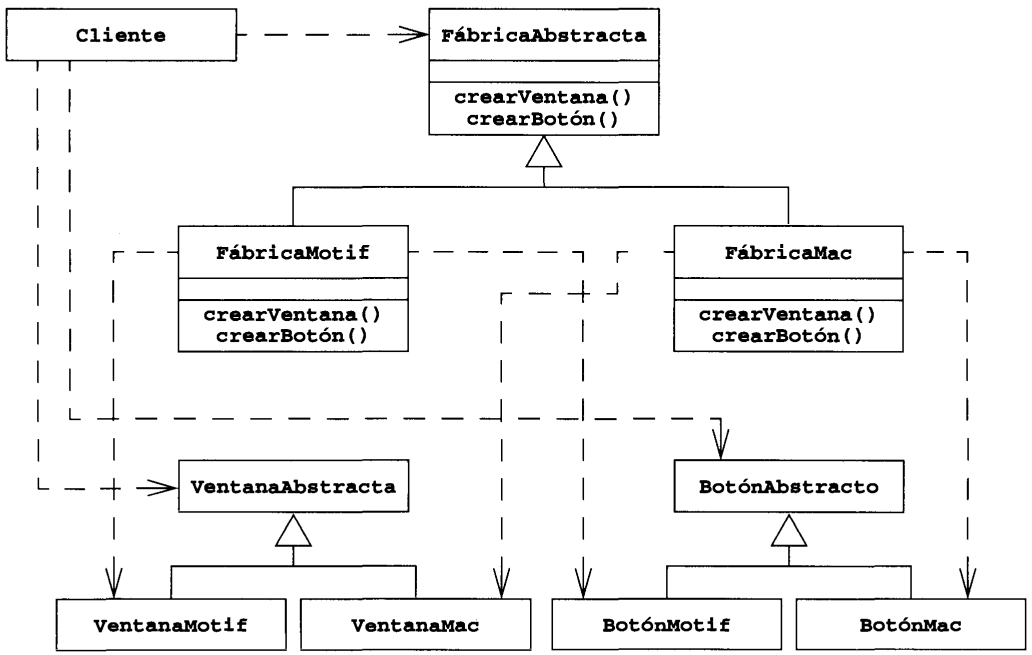


Figura A-1 Patrón de diseño `FábricaAbstracta` (diagrama de clase UML).

Propósito Este patrón se usa para aislar una aplicación con respecto a las clases concretas proporcionadas por una plataforma específica, como un estilo de ventanas o un sistema operativo. En consecuencia, usando este patrón se puede desarrollar una aplicación para que ejecute de manera uniforme en un rango de plataformas.

Descripción Cada plataforma (por ejemplo, un estilo de ventana) está representada por una clase `Fábrica` y varias `ClaseConcreta` para cada concepto de la plataforma (por ejemplo, ventana, botón, diálogo). La clase `Fábrica` proporciona métodos para la creación de instancias de las `ClaseConcreta`. El transporte de una aplicación a una nueva plataforma se reduce a la implementación de una `Fábrica` y una `ClaseConcreta` para cada concepto.

Ejemplos

- Encapsulamiento estático de estilos de ventanas (figura 7-28 en la página 267).
- Encapsulamiento dinámico de estilos de ventanas (Swing, [JFC, 1999]).

Conceptos relacionados Incrementar la reutilización (sección 7.4.9 en la página 265), eliminación de dependencias de implementación (sección 7.4.10 en la página 267).

A.2 Adaptador: envoltura alrededor de código heredado

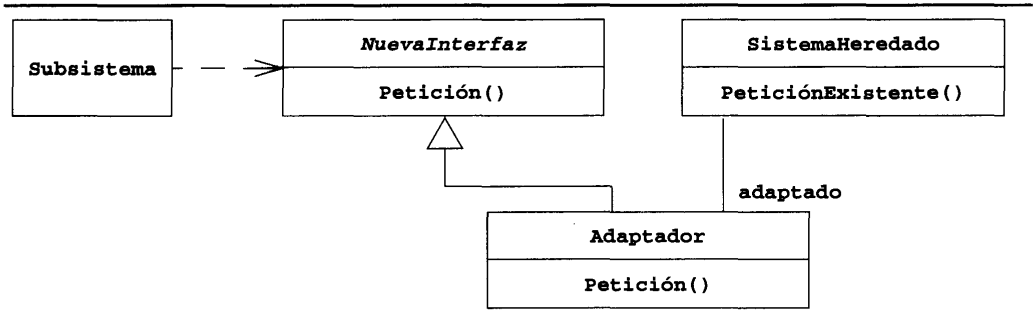


Figura A-2 Patrón `Adaptador` (diagrama de clase UML). El patrón `Adaptador` se usa para proporcionar una interfaz diferente (*`NuevaInterfaz`*) a un componente existente (`SistemaHeredado`).

Propósito Este patrón encapsula un fragmento de código heredado que no fue diseñado para trabajar con el sistema. También limita el impacto de la sustitución del fragmento de código heredado con un componente diferente.

Descripción Supongamos que un `Subsistema` que llama necesita acceso a la funcionalidad que proporciona un `SistemaHeredado` existente. Sin embargo, el `SistemaHeredado` no se apega a la *`NuevaInterfaz`* que usa el `Subsistema` que llama. Este hueco se llena creando una clase Adaptador que implementa la *`NuevaInterfaz`* usando métodos del `SistemaHeredado`. Ya que el que llama sólo accede a la *`NuevaInterfaz`*, el `SistemaHeredado` puede reemplazarse más adelante con un componente alterno.

Ejemplo

- Ordenamiento de instancias de una clase `String` existente con un método `sort()` existente (figura 6-34 en la página 202): `MyStringsComparator` es un `Adaptador` para cubrir el hueco entre la clase `String` y la interfaz `Comparator` usada por el método `Array.sort()`.

Conceptos relacionados El `Puente` (sección A.3) llena el hueco entre una interfaz y sus implementaciones.

A.3 Puente: permitir implementaciones alternas

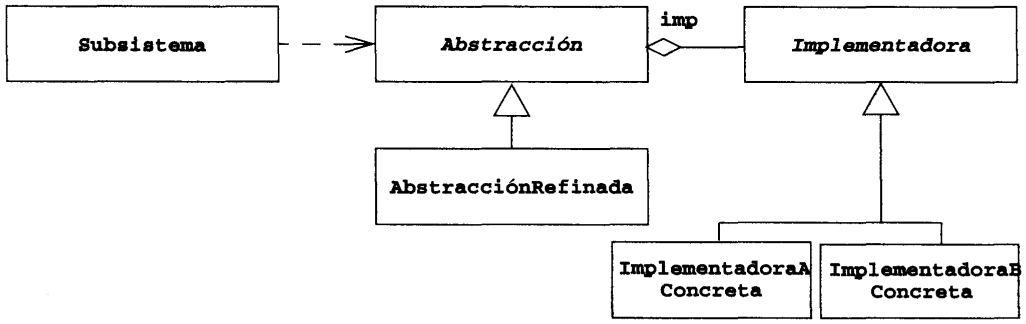


Figura A-3 Patrón `Puente` (diagrama de clase UML).

Propósito Este patrón desacopla la interfaz de una clase con respecto a su implementación. A diferencia del patrón `Adaptador`, el desarrollador no está restringido por un fragmento de código existente. Tanto la interfaz como la implementación pueden refinarse en forma independiente.

Descripción Supongamos que necesitamos desacoplar una *`Abstracción`* con respecto a una *`Implementadora`*, debido a que necesitamos sustituir diferentes *`Implementadora`* para una *`Abstracción`* dada sin causar ningún impacto en un *`Subsistema`* que llama. Esto se realiza proporcionando una clase *`Abstracción`* que implementa sus servicios desde el punto de vista de los métodos de una interfaz *`Implementadora`*. Las *`Implementadora Concreta`* que necesitan sustituirse refinan la interfaz *`Implementadora`*.

Ejemplos

- Independencia de vendedor (figura 6-37 en la página 206): La interfaz `ODBC` (la `Abstracción`) desacopla a quien llama con respecto a un sistema de administración de base de datos. Para cada sistema de administración de base de datos, un `Manejador ODBC` refina la `Implementación ODBC` (la *`Implementadora`*). Cuando la *`Abstracción`* no hace suposiciones acerca de las `Implementadora Concreta`, se les puede cambiar sin que se dé cuenta el `Subsistema` que llama, incluso en el tiempo de ejecución.
- Prueba unitaria (figura 9-11 en la página 342): la `Interfaz BaseDeDatos` (la `Abstracción`) desacopla a la `InterfazUsuario` (el `Subsistema`) con respecto a la `BaseDeDatos` (una `Implementadora Concreta`), permitiendo que la `InterfazUsuario` y la `BaseDeDatos` se prueben en forma independiente. Cuando se prueba la `InterfazUsuario`, un `Stub de prueba` (otra `Implementadora Concreta`) sustituye a la `BaseDeDatos`.

Conceptos relacionados El patrón `Adaptador` (sección A.2) llena el hueco entre dos interfaces.

A.4 Comando: encapsulamiento del control

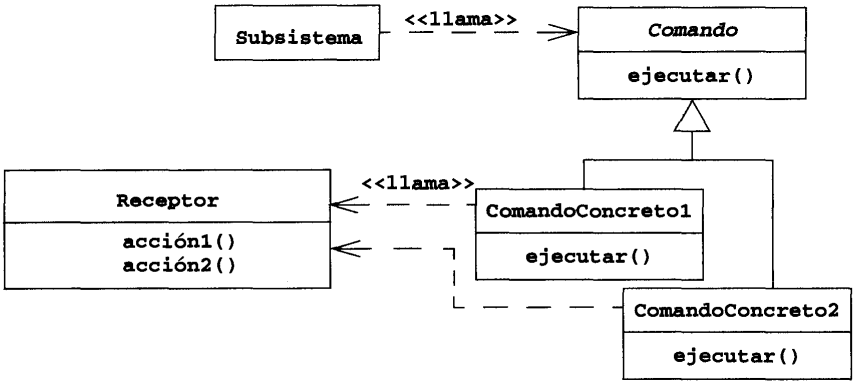


Figura A-4 Patrón `Comando` (diagrama de clase UML).

Propósito Este patrón permite el encapsulamiento del control de tal forma que las peticiones del usuario pueden tratarse de manera uniforme, independientemente de la petición específica. Este patrón protege esos objetos contra los cambios que resultan de nueva funcionalidad. Otra ventaja de este patrón es que se centraliza el flujo de control en el objeto de comando, en vez de que esté distribuido entre los objetos de interfaz.

Descripción Una interfaz de *Comando* abstracta define los servicios comunes que deberán implementar los `ComandoConcreto`. Los `ComandoConcreto` recopilan datos del `Subsistema` cliente y manipulan los objetos de entidad (`Receptores`). Los `Subsistema` interesados sólo en la abstracción del *Comando* general (por ejemplo, una pila deshacer) sólo acceden a la clase abstracta *Comando*. Los `Subsistema` cliente no acceden en forma directa a los objetos de entidad.

Ejemplos

- Proporcionar una pila deshacer para comandos de usuario: todos los comandos visibles para el usuario son refinamientos de la clase abstracta *Comando*. Se requiere que cada comando implemente los métodos `hacer()`, `deshacer()` y `rehacer()`. Una vez que se ejecuta un comando se le empuja en una pila deshacer. Si el usuario quiere deshacer el último comando, se envía el mensaje `deshacer()` al objeto *Comando* que está hasta arriba de la pila.
- Desacoplamiento de los objetos de interfaz con respecto a los objetos de control (figura 6-45 en la página 215, vea también `Acciones` Swing [JFC, 1999]): todos los comandos visibles para el usuario son refinamientos de la clase abstracta *Comando*. Los objetos de interfaz, como los conceptos de menú y los botones, crean y envían mensajes a los objetos *Comando*. Sólo los objetos *Comando* modifican a los objetos de entidad. Cuando se cambia la interfaz de usuario (por ejemplo, una barra de menú se reemplaza con una barra de herramientas) sólo se modifican los objetos de interfaz.

Conceptos relacionados Arquitectura MVC (figura 6-15 en la página 184).

A.5 Compuesto: representación de jerarquías recurrentes

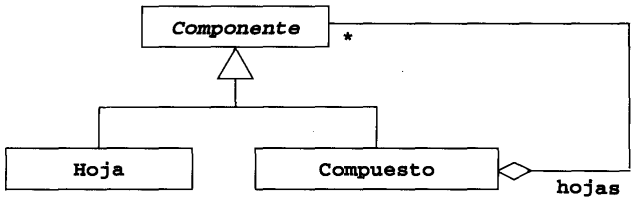


Figura A-5 Patrón `Compuesto` (diagrama de clase UML).

Propósito Este patrón representa una jerarquía recurrente. Los servicios relacionados con la jerarquía contenedora se factorizan usando herencia, permitiendo que un sistema trate a una hoja o a un compuesto de manera uniforme. El comportamiento específico de una hoja puede modificarse sin que haya ningún impacto en la jerarquía contenedora.

Descripción La interfaz `Componente` especifica los servicios que se comparten entre `Hoja` y `Compuesto` (por ejemplo, `mover(x,y)` para un elemento gráfico). Un `Compuesto` tiene una asociación de agregación con *`Componente`* e implementa cada servicio iterando sobre cada *`Componente`* contenido (por ejemplo, el método `Compuesto.mover(x,y)` llama iterativamente a `Componente.mover(x,y)`). Los servicios `Hoja` hacen el trabajo real (por ejemplo, `Hoja.mover (x,y)` modifica las coordenadas de `Hoja` y lo vuelve a trazar).

Ejemplos

- Grupos de acceso recurrente (Lotus Notes): un grupo de acceso de Lotus Notes puede contener cualquier cantidad de usuarios y grupos de acceso.
- Grupos de elementos trazables: los elementos trazables pueden organizarse en grupos que pueden moverse y escalarse de manera uniforme. Los grupos también pueden contener otros grupos.
- Jerarquía de archivos y directorios (figura 5-7 en la página 137): los directorios pueden contener archivos y otros directorios. Las mismas operaciones están disponibles para mover, renombrar y eliminar de manera uniforme archivos y directorios.
- Descripción de la descomposición en subsistemas (figura 6-3 en la página 173): usamos un patrón `Compuesto` para describir la descomposición en subsistemas. Un subsistema está compuesto por clases y otros subsistemas. Tome en cuenta que los subsistemas no se implementan, de hecho, como `Compuestos` a los que se les añaden clases en forma dinámica.
- Descripción de jerarquías de tareas (figura 6-8 en la página 177): usamos un patrón `Compuesto` para describir la organización de `Tareas` (`Compuestos`) en `Subtareas` (*`Componente`*) y `ConceptoAcción` (`Hojas`). Usamos un modelo similar para describir las `Fase`, `Actividad` y `Tarea` (figura 12-6 en la página 462).

Conceptos relacionados Patrón `Fachada` (sección A.6).

A.6 Fachada: encapsulamiento de subsistemas

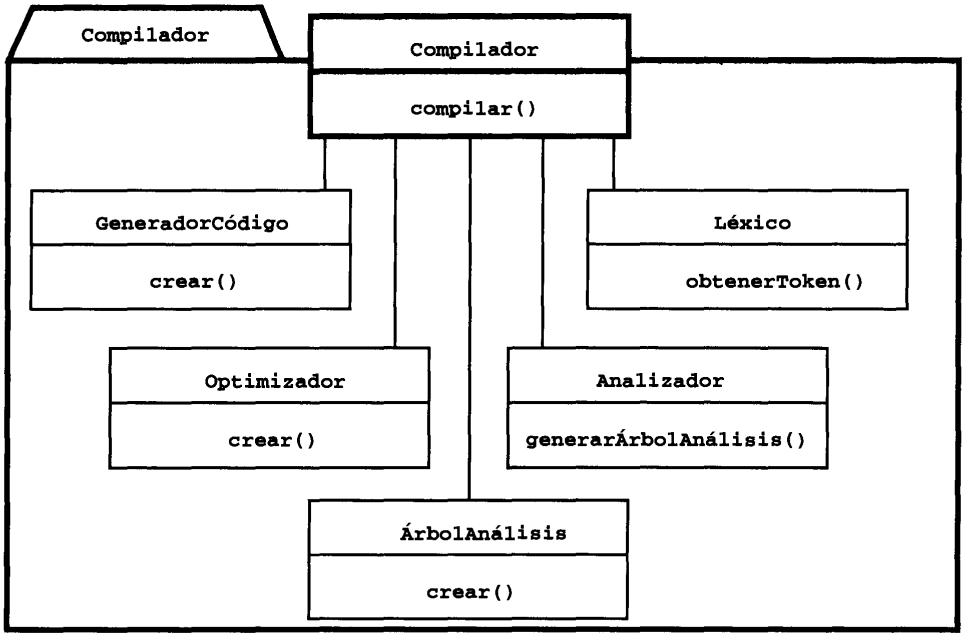


Figura A-6 Un ejemplo de un patrón `Fachada` (diagrama de clase UML).

Propósito El patrón `Fachada` reduce las dependencias entre clases encapsulando un subsistema con una interfaz unificada simple.

Descripción Una sola clase `Fachada` implementa una interfaz de alto nivel para un subsistema, llamando a los métodos de las clases de menor nivel. Una `Fachada` es opaca en el sentido de que quien llama no tiene acceso directo a las clases de menor nivel. El uso de un patrón `Fachada` en forma recurrente produce un sistema en capas.

Ejemplo

- Encapsulamiento de subsistema (figura 6-30 en la página 198): un `Compilador` consta de `Léxico`, `Analizador`, `ÁrbolAnálisis`, `GeneradorCódigo` y `Optimizador`. Sin embargo, cuando se compila una cadena hacia código ejecutable, quien llama sólo maneja la clase `Compilador`, la cual llama a los métodos adecuados de las clases contenidas.

Conceptos relacionados Acoplamiento y coherencia (sección 6.3.3 en la página 174), capas y particiones (sección 6.3.4 en la página 178), patrón `Compuesto` (sección A.5).

A.7 Observador: desacoplamiento de entidades con respecto a vistas

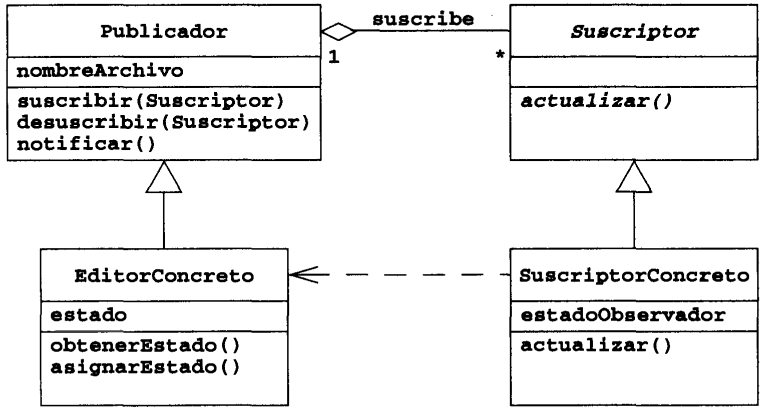


Figura A-7 El patrón `Observador` (diagrama de clase UML).

Propósito Este patrón nos permite mantener consistencia entre los estados de un `Publicador` y varios *`Suscriptor`*.

Descripción Un `Publicador` (llamado `Subject` en [Gamma *et al.*, 1994]) es un objeto cuya función principal es mantener cierto estado, por ejemplo, una matriz. Uno o más `Suscriptor` (llamados *`Observer`* en [Gamma *et al.*, 1994]) usan el estado mantenido por un `Publicador`, por ejemplo, para desplegar una matriz como una tabla o una gráfica. Esto introduce redundancias entre el estado del `Publicador` y los `Suscriptor`. Para resolver este problema, los `Suscriptor` llaman al método `suscribir()` para registrarse con un `Publicador`. Cada `SuscriptorConcreto` también define un método `actualizar()` para sincronizar el estado entre el `Publicador` y el `SuscriptorConcreto`. Cada vez que cambia el estado del `Publicador`, el `Publicador` llama a su método `notificar()` que llama en forma iterativa a cada método `Suscriptor.actualizar()`.

Ejemplos

- La interfaz `Observador` y la clase `Observable` se usan en Java para realizar un patrón `Observador` ([JFC, 1999]).
- El patrón Observador puede usarse para realizar la suscripción y notificación en una arquitectura Modelo/Vista/Controlador (figura 6-15 de la página 184).

Conceptos relacionados Entidad, interfaz, objetos de control (sección 5.3.1 en la página 134).

A.8 Apoderado: encapsulamiento de objetos caros

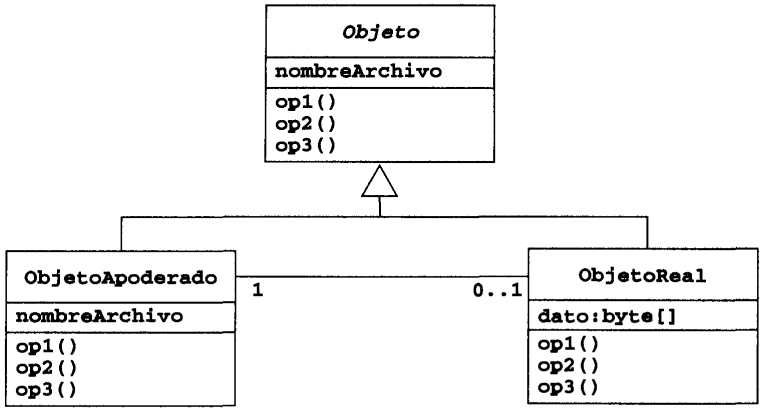


Figura A-8 El patrón `Apoderado` (Diagrama de clase UML).

Propósito Este patrón mejora el desempeño o la seguridad de un sistema retrasando cálculos caros, usando memoria sólo cuando se necesita o revisando el acceso antes de cargar un objeto en memoria.

Descripción La clase `ObjetoApoderado` actúa a nombre de la clase `ObjetoReal`. Ambas clases implementan la misma interfaz. El `ObjetoApoderado` guarda un subconjunto de los atributos de `ObjetoReal`. El `ObjetoApoderado` maneja completamente determinadas peticiones (por ejemplo, la determinación del tamaño de una imagen), mientras que las demás se delegan hacia el `ObjetoReal`. Después de la delegación se crea y carga en memoria el `ObjetoReal`.

Ejemplos

- Apoderado de protección (figura 6-38 en la página 210): una clase de asociación `Acceso` contiene un conjunto de operaciones que puede usar un `Corredor` para acceder a un `Portafolio`. Cada operación del `ApoderadoPortafolio` revisa primero con `esAccesible()` si el `Corredor` que llama tiene acceso legítimo. Una vez que se ha otorgado el acceso, `ApoderadoPorafolio` delega la operación al objeto `Portafolio` real. Si se niega el acceso no se carga en memoria el objeto `Portafolio` real.
- Apoderado de almacenamiento (figura 7-31 en la página 273): un objeto `ApoderadoImagen` actúa a nombre de una `Imagen` guardada en disco. El `ApoderadoImagen` contiene la misma información que la `Imagen` (por ejemplo, anchura, altura, posición, resolución) a excepción del contenido de la `Imagen`. El `ApoderadoImagen` da servicio a todas las peticiones que son independientes del contenido. Sólo cuando se necesita el acceso al contenido de Imagen (por ejemplo, cuando se traza en pantalla) el `ApoderadoImagen` crea al objeto `ImagenReal` y carga su contenido desde el disco.

Conceptos relacionados Cacheo de cálculos costosos (sección 7.4.13 en la página 272).

A.9 Estrategia: encapsulamiento de algoritmos

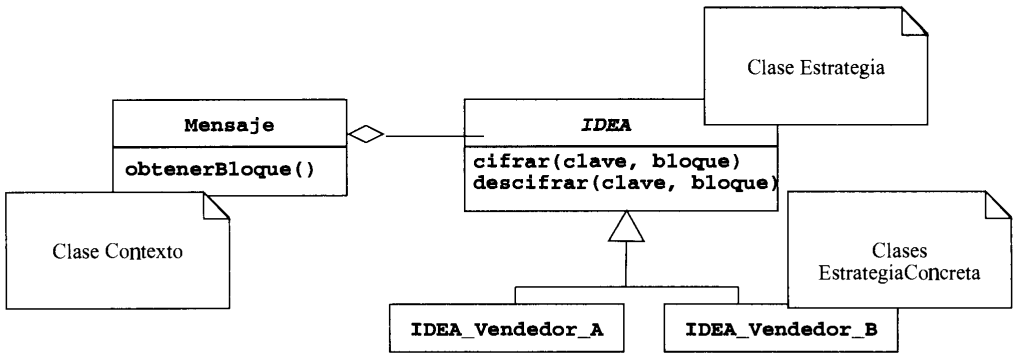


Figura A-9 Un ejemplo del patrón `Estrategia` que encapsula varias implementaciones del algoritmo de cifrado IDEA (diagrama de clase UML). Las clases `Mensaje` e *IDEA* cooperan para realizar el cifrado de texto llano. La selección de una implementación puede realizarse en forma dinámica.

Propósito Este patrón desacopla un algoritmo con respecto a sus implementaciones. Sirve al mismo propósito que los patrones adaptador y puente, excepto que lo que se encapsula es un comportamiento.

Descripción Una clase abstracta `Algoritmo` proporciona métodos para inicializar, ejecutar y obtener los resultados de un `Algoritmo`. Las clases `AlgoritmoConcreto` refinan a `Algoritmo` y proporcionan implementaciones alternas del mismo comportamiento. Los `AlgoritmoConcreto` pueden cambiarse sin que haya ningún impacto en quien llama.

Ejemplo

- Algoritmos de cifrado (figura 6-40 en la página 212): los algoritmos de cifrado proporcionados por vendedores plantean un problema interesante: ¿cómo podemos estar seguros que el software proporcionado no tiene una trampa? Además, una vez que se encuentra una vulnerabilidad en un paquete ampliamente usado, ¿cómo protegemos al sistema hasta que se disponga de un parche? Para resolver ambos problemas, podemos usar implementaciones redundantes del mismo algoritmo. Para reducir la dependencia de un vendedor específico encapsulamos estas implementaciones con un solo patrón `Estrategia`.

Conceptos relacionados Patrón `Adaptador` (sección A.2) y patrón `Puente` (sección A.3).

Referencias

[Ambler, 1998] S. W. Ambler, *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press, Nueva York, 1998.

[Brown *et al.*, 1999] W. J. Brown, H. W. McCormick y S. W. Thomas, *AntiPatterns and Patterns in Software Configuration Management*, Wiley, Nueva York, 1999.

[Buschmann *et al.*, 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, Chichester, Inglaterra, 1996.

[Feiler *et al.*, 1998] P. Feiler y W. Tichy, "Propagator: A family of patterns", *Proceedings of TOOLS-23'97*, 28 de julio–1 de agosto de 1997, Santa Bárbara, CA.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, MA, 1997.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1994.

[JFC, 1999] *Java Foundation Classes*, JDK Documentation, Javasoft, 1999.

[Larman, 1998] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Mowbray y Malveau, 1997] T. J. Mowbray y R. C. Malveau, *CORBA Design Patterns*, Wiley, Nueva York, 1997.

Glosario

Abstracción Clasificación de fenómenos en forma de conceptos. *Vea también* modelado.

Acoplamiento Fortaleza de las dependencias entre dos subsistemas o dos clases. Un acoplamiento bajo da como resultado sistemas que pueden modificarse con un impacto mínimo en los demás subsistemas.

Actividad Conjunto de tareas que se realizan para lograr un propósito específico. Las actividades pueden incluir pocas o muchas tareas, dependiendo del alcance de su objetivo. Algunos ejemplos de actividades incluyen la obtención de requerimientos, la identificación de objetos y las pruebas unitarias. *Vea también* proceso.

Actor Entidad externa que necesita intercambiar información con el sistema. Un actor puede representar un papel de usuario o a otro sistema.

Acuerdo del proyecto Documento que define de manera formal el alcance, la duración, el costo y los productos a entregar de un proyecto. Los requerimientos se clasifican a menudo en medulares (es decir, los que debe satisfacer el sistema cuando se entrega), deseables (es decir, los que deben satisfacerse a la larga) y opcionales (es decir, los que pueden requerirse o no en el futuro). El acuerdo del proyecto incluye el enunciado del problema.

Adaptabilidad Cualidad de un sistema que indica con cuánta facilidad se puede transportar un sistema a diferentes dominios de aplicación.

Administración de construcción Soporte de herramientas para la construcción automática de un sistema cuando se añaden nuevas versiones al sistema.

Administración de la configuración *Vea* administración de la configuración del software.

Administración de la configuración del software Actividad durante la cual los desarrolladores supervisan y controlan los cambios al sistema o a los modelos. Los objetivos de la adminis-

tración de la configuración incluyen guardar suficiente información para que sea posible restaurar el sistema como era en una versión anterior, la prevención de cambios que no son consistentes con los objetivos del proyecto y la administración de las rutas de desarrollo concurrentes orientadas a la evaluación de soluciones competitivas.

Administración de la fundamentación Actividad durante la cual los desarrolladores crean, capturan, actualizan y tienen acceso a la información sobre la fundamentación.

Administración del proyecto Actividad durante la cual los gerentes planean, presupuestan, supervisan y controlan el proceso de desarrollo. La administración del proyecto asegura que se satisfagan las restricciones y objetivos del proyecto.

Administración del riesgo Método de administración para la identificación y resolución de áreas de incertidumbre antes de que tengan un impacto negativo en la calendarización del proyecto o en la calidad del sistema.

Adquisición de conocimiento Actividad de recopilar datos, organizarlos en información y formalizarla en conocimiento.

Agregación Asociación que indica una relación de partes completas entre dos clases. Por ejemplo, un `Sistema` que está compuesto por `Subsistemas` es una asociación de agregación.

Agregado AC *Vea* agregado de la administración de la configuración.

Agregado de la administración de la configuración Agregado de artículos de la administración de la configuración relacionados. También se llama agregado AC.

Alternativa *Vea* propuesta.

Ambigüedad Propiedad de un modelo que indica si un concepto corresponde a dos o más fenómenos no relacionados.

Análisis Actividad durante la cual los desarrolladores aseguran que los requerimientos del sistema son correctos, completos, consistentes, no ambiguos y realistas. El análisis produce el modelo de análisis.

Análisis de conocimiento de tareas (KAT) Técnica de modelado del dominio de aplicación basado en la observación de usuarios en acción.

Análisis orientado a objetos Actividad que se refiere al modelado del dominio de aplicación con objetos.

Analista Papel que representa a los desarrolladores que obtienen información del dominio de aplicación a partir de los usuarios y clientes y que construyen un modelo de casos de uso del sistema a desarrollar.

Anomalía *Vea* defecto.

API *Vea* interfaz de programación de aplicaciones.

Área de proceso principal (KPA) Conjunto de actividades que logra un objetivo que se considera importante para alcanzar un nivel dado de madurez del proceso. Algunos ejemplos de KPA son la administración de la configuración, la administración del cambio de requerimientos y la administración del riesgo.

Argumentación Debate que sostienen los participantes para resolver un problema.

Argumento Valor pasado junto con un mensaje.

Arquitecto Papel que representa a los desarrolladores que toman decisiones estratégicas sobre el sistema y que construyen el modelo de diseño del sistema.

Arquitectura *Vea* modelo de diseño del sistema.

Arquitectura abierta Arquitectura en capas en la cual una capa utiliza los servicios de cualquier capa que se encuentre debajo de ella (y no sólo las capas que están inmediatamente debajo de ella). *Compare con:* arquitectura cerrada.

Arquitectura cerrada Sistema en capas en el cual una capa sólo puede depender de las capas que están inmediatamente debajo de ella. *Compare con:* arquitectura abierta.

Arquitectura cliente/servidor Arquitectura de software en la cual las interacciones del usuario están administradas por programas cliente simples, y un programa servidor central proporciona la funcionalidad.

Arquitectura de depósito Arquitectura de software en la cual un solo subsistema administra y guarda los datos persistentes. Los subsistemas periféricos son relativamente independientes y sólo interactúan mediante el subsistema central.

Arquitectura de software *Vea* modelo de diseño del sistema.

Arquitectura de tubo y filtro Arquitectura de software en la cual los subsistemas procesan datos en forma secuencial a partir de un conjunto de entradas y envían sus resultados a otros subsistemas mediante un conjunto de salidas. A las asociaciones entre subsistemas se les llama tubos. A los subsistemas se les llama filtros. Los filtros no tienen dependencias entre ellos y, por tanto, pueden reacomodarse en orden y configuración diferentes.

Arquitectura par a par Generalización de la arquitectura cliente/servidor en la cual los subsistemas pueden actuar como clientes o como servidores.

Artículo de configuración Producto de trabajo al que se trata como una sola entidad para efectos de la administración de la configuración y que necesita ser línea base. *Vea también* agregado de la administración de la configuración.

Asociación Relación entre dos o más clases que indica los vínculos posibles entre instancias de las clases. Una asociación tiene un nombre, y puede tener información de multiplicidad y papeles asociados a cada uno de sus extremos.

Asociación calificada Asociación con un extremo indexado por un atributo. Por ejemplo, la asociación entre `Directorio` y `Archivo` es una asociación calificada indexada por un nombre de archivo en el extremo de `Directorio`. La calificación es una técnica para reducir la multiplicidad de las asociaciones.

Asociación de muchos a muchos Asociación con multiplicidad `0..n` o `1..n` en ambos extremos.

Asociación de uno a uno Asociación con multiplicidad de 1 en cada extremo.

Asociación uno a muchos Asociación que tiene una multiplicidad de 1 en un extremo y de `0..n` o `1..n` en el otro extremo.

Atributo Propiedad de una clase que tiene nombre y que define un rango de valores que puede contener un objeto. Por ejemplo, `tiempo` es un atributo de la clase `Reloj`.

Auditor Papel responsable de la selección y evaluación de promociones para un lanzamiento.

Auditoría Validación de versiones antes de un lanzamiento que asegura una entrega consistente y completa.

Autentificación Proceso de asociar a una persona con derechos de acceso.

Calendarización Correspondencia entre el modelo de tareas y una línea de tiempo. Una calendarización representa al trabajo desde el punto de vista de tiempo calendario.

Cambio *Vea* delta.

Capa Subsistema en una descomposición jerárquica. Una capa sólo puede depender de capas de nivel inferior y no tiene conocimiento de las capas que están encima de ella.

Capacidad Una representación del control de acceso en la que los derechos legítimos están representados por pares `{clase, operación}` asociados a un `actor`. Algunos ejemplos de capacidades incluyen una llave de cerradura, una tarjeta inteligente y un boleto para el teatro.

Carril Concepto de agrupamiento UML que indica actividades que realiza el mismo objeto o un conjunto de objetos.

Caso de prueba Conjunto de entradas y resultados esperados que ejercitan a un componente con el propósito de causar fallas.

Caso de uso Secuencia de interacciones general entre uno o más actores y el sistema. *Vea también* escenario.

Ciclo de vida *Vea* ciclo de vida del software.

Ciclo de vida del software Todas las actividades y productos de trabajo necesarios para el desarrollo de un sistema de software.

Cifrado Conversión de un mensaje, llamado texto llano, en un mensaje cifrado, llamado texto cifrado, de tal forma que no pueda ser comprendido por personas no autorizadas.

Clase Abstracción de un conjunto de objetos que tienen los mismos atributos, operaciones, relaciones y semántica. Las clases son diferentes a los tipos de datos abstractos en que una clase puede definirse mediante la especialización de otra clase. Por ejemplo, los lenguajes de programación, como Modula y Ada, proporcionan mecanismos para la definición de tipos de datos abstractos, y los lenguajes orientados a objetos, como Java, C++ o Smalltalk, proporcionan mecanismos para la definición de clases.

Clase abstracta Superclase que sólo se utiliza para generalización y que nunca se instancia.

Clase de asociación Asociación que tiene atributos y operaciones.

Clase de evento Abstracción que representa a un conjunto de eventos para los cuales tiene una respuesta común el sistema.

Cliente Papel que representa a la persona o compañía que paga por el desarrollo del sistema.

CMM *Vea* modelo de madurez de capacidades.

Coherencia Fortaleza de las dependencias dentro de un subsistema o una clase. Es deseable una coherencia alta, ya que mantiene juntas a las clases relacionadas a fin de que puedan modificarse de manera consistente.

Comité de control del cambio Equipo que aprueba las peticiones de cambio en el contexto de un proceso de cambio formal.

Componente Parte física y reemplazable del sistema que se apega a una interfaz. Algunos ejemplos de componentes son las bibliotecas de clase, los marcos y los programas binarios.

Comunicación Actividad durante la cual los desarrolladores intercambian información, ya sea en forma síncrona o asíncrona, y de manera espontánea o de acuerdo con una calendarización.

Concepto Abstracción de un conjunto de fenómenos que tienen propiedades comunes. Por ejemplo, este libro, mi reloj negro y el club de pescadores del valle son fenómenos. Los libros de texto sobre ingeniería del software orientada a objetos, los relojes negros y los clubes de pescadores son conceptos. *Compare con:* fenómeno.

Concepto de acción Tarea asignada a un participante que tiene una fecha de terminación, por lo general a consecuencia de la resolución de un problema.

Condición de frontera Condición especial que debe manejar el sistema. Las condiciones de frontera incluyen el arranque, el apagado y las excepciones.

Confiabilidad Propiedad de un sistema que indica la probabilidad de que su comportamiento observado se apegue a la especificación de su comportamiento.

Confiabilidad del software Propiedad de un sistema de software que indica la probabilidad de que el software no causará una falla del sistema durante un lapso especificado.

Configuración Versión de un agregado AC.

Conjunto de cambio Conjunto de deltas que indica las diferencias entre dos configuraciones.

Consistencia Propiedad de un modelo que indica si se contradice a sí mismo o no. Un modelo es inconsistente si proporciona varias vistas incompatibles del sistema.

Consultor Cualquier papel que se refiera a proporcionar apoyo temporal y especializado en donde les falte experiencia a los participantes en el proyecto. Los papeles de consultor incluyen al usuario, al cliente, al consultor técnico y al especialista en el dominio de aplicación.

Consultor técnico Papel de consultor interesado en proporcionar al proyecto experiencia en el dominio de solución.

Contabilidad de estado En la administración de la configuración, seguimiento de las peticiones de cambio, de la aprobación del cambio y de la fundamentación del cambio.

Contrato Conjunto de restricciones sobre una clase o componente que permite que quien llama y a quien se llama compartan las mismas suposiciones acerca de la clase o componente. *Vea también* invariante, precondición y poscondición.

Control de acceso dinámico Política de control de acceso que puede especificarse al tiempo de ejecución. *Compare con:* control de acceso estático.

Control de acceso estático Política de control de acceso que sólo puede especificarse en el tiempo de compilación. *Compare con:* control de acceso dinámico.

Control de tráfico centralizado (CTC) Procedimientos y sistemas que permiten a los despachadores supervisar y controlar remotamente el tráfico de trenes desde una ubicación central.

Control del cambio Proceso de aprobación o rechazo de las solicitudes de cambios que asegura que el sistema esté evolucionando de manera consistente con los objetivos del proyecto.

Control manejado por eventos Paradigma de flujo de control en el que un ciclo principal espera un evento y lo despacha hacia el objeto adecuado.

Control manejado por procedimientos Paradigma de flujo de control en el que las operaciones esperan entrada. La secuencia de operaciones se logra de manera explícita mediante el envío de mensajes.

Conversaciones en los pasillos Mecanismo de comunicación síncrono durante el cual los participantes se reúnen frente a frente en forma accidental.

Corrección Cambio a un componente con el propósito de reparar un defecto.

Criterio Medida de bondad que se usa cuando se evalúan alternativas para un problema.

Criterio de costo Objetivo de diseño relacionado con el costo de desarrollo, operación o instalación del sistema.

Criterio de dependencia Objetivo de diseño relacionado con la minimización de la cantidad de fallas del sistema y sus consecuencias. Algunos ejemplos de criterios de dependencia son fortaleza, confiabilidad, disponibilidad, tolerancia a fallas, seguridad e inocuidad.

Criterio de desempeño Objetivo de diseño relacionado con atributos de velocidad o espacio de un sistema. Algunos ejemplos de criterio de desempeño son tiempo de respuesta, producción y espacio de memoria.

Criterio de mantenimiento Objetivo de diseño relacionado con la dificultad de cambiar o mejorar el sistema después de que se ha entregado.

Criterio del usuario final Objetivo de diseño relacionado con los atributos del sistema visibles para el usuario que todavía no se ha tratado bajo un criterio de dependencia o desempeño.

CTC *Vea* control de tráfico centralizado.

Cuestionario Mecanismo de comunicación síncrono durante el cual se obtiene información de un participante mediante un conjunto de preguntas estructurado.

Decisión En el contexto de los diagramas de actividad, una rama en el flujo de control de un diagrama de actividad. Una decisión indica transiciones alternas basadas en una condición. En el contexto del modelado de problemas: la resolución de un problema.

Defecto Causa mecánica o algorítmica de un error. También se le llama anomalía.

Definición del problema Modo de comunicación calendarizada durante el cual el cliente y el gerente definen el alcance del sistema.

Delegación Mecanismo para la reutilización del código en el que una operación simplemente vuelve a enviar (es decir, delega) un mensaje a otra clase para lograr el comportamiento deseado.

Delta Conjunto de diferencias entre dos versiones sucesivas. También se le llama cambio.

Depósito En una arquitectura de depósito, el subsistema central que administra los datos persistentes. En la administración de la configuración, una biblioteca de lanzamientos.

Depuración Técnica de detección de defectos que trata de encontrar un defecto a partir de una falla no planeada avanzando en forma gradual por los estados sucesivos del sistema.

Desarrollador *Vea* papel de desarrollador.

Desarrollo cruzado Grupo de procesos que incluye a los procesos que suceden a todo lo largo del proyecto y que asegura la terminación y calidad de las funciones del proyecto. Algunos ejemplos de desarrollo cruzado son la validación y verificación, la administración de la configuración, la documentación y el entrenamiento. También se llaman procesos integrales.

Descomposición en subsistemas División del sistema en subsistemas. Cada subsistema se describe en función de sus servicios durante el diseño del sistema y de su API durante el diseño de objetos. La descomposición en subsistemas es parte del modelo de diseño del sistema.

Detección de defectos Método que trata de descubrir defectos en un sistema en ejecución. Algunos ejemplos de técnicas de detección de defectos son la depuración y las pruebas.

Diagrama de actividad Notación UML que representa el comportamiento de un sistema desde el punto de vista de actividades. Una actividad es un estado que representa la ejecución de un conjunto de operaciones. La terminación de estas operaciones activa una transición hacia otra actividad. Los diagramas de acción se usan para ilustrar combinaciones de flujo de control y de datos.

Diagrama de caso de uso Notación UML que se usa durante la obtención de requerimientos y el análisis para representar la funcionalidad del sistema. Un caso de uso describe una función del sistema desde el punto de vista de una secuencia de interacciones entre un actor y el sistema. Un caso de uso también incluye condiciones iniciales que necesitan ser ciertas antes de la ejecución del caso de uso y las condiciones finales que son ciertas al final del caso de uso.

Diagrama de clase Notación UML que representa la estructura del sistema desde el punto de vista de objetos, clases, atributos, operaciones y asociaciones. Los diagramas de clase se usan para representar modelos de objetos durante el desarrollo.

Diagrama de gráfica de estado Notación UML que representa el comportamiento de un objeto individual como varios estados y transiciones entre esos estados. Un estado representa un conjunto de valores particular para un objeto. Encontrándose en un estado dado, una transición representa un estado futuro hacia el que puede moverse un objeto y las condiciones asociadas con el cambio de estado. Los diagramas de gráfica de estado se usan durante el análisis para describir objetos que tienen comportamiento no trivial.

Diagrama de objetos Diagrama de clase que sólo incluye instancias.

Diagrama de organización Diagrama UML que representa a los componentes del tiempo de ejecución y su asignación a nodos de hardware.

Diagrama de secuencia Notación UML que representa el comportamiento del sistema como una serie de interacciones entre un grupo de objetos. Cada objeto se muestra como una columna en el diagrama. Cada interacción se muestra como una flecha entre dos columnas. Los diagra-

mas de secuencia se usan durante el análisis para identificar objetos, atributos, relaciones o atributos faltantes. Los diagramas de secuencia se usan durante el diseño de objetos para refinar la especificación de las clases.

Diseñador de objetos Papel que representa a los desarrolladores que refinan y especifican las interfaces de clases durante el diseño de objetos.

Diseño de alto nivel Descomposición inicial en subsistemas que se usa para la organización de los equipos y para la planeación inicial.

Diseño de aplicación conjunto (JAD) Técnica de obtención de requerimientos que involucra la colaboración de clientes, usuarios y desarrolladores en la construcción de una especificación del sistema mediante una reunión de trabajo de una semana.

Diseño de objetos Actividad durante la cual los desarrolladores definen objetos personalizados para llenar el hueco entre el modelo de análisis y la plataforma de hardware/software. Esto incluye la especificación de interfaces de objetos y subsistemas, la selección de componentes hechos, la reestructuración del modelo de objetos para lograr los objetivos de diseño y la optimización del modelo de objetos para mejorar el desempeño. El diseño de objetos da como resultado el modelo de diseño de objetos.

Diseño del sistema Actividad mediante la cual los desarrolladores definen el modelo de diseño del sistema, incluyendo los objetivos de diseño del proyecto y descomponiéndolo en subsistemas más pequeños que pueden ser realizados por equipos individuales. El diseño del sistema también conduce a la selección de la plataforma de hardware/software, la estrategia de administración de datos persistentes, el control de flujo global, la política de control de acceso y el manejo de las condiciones de frontera.

Diseño orientado a objetos Actividad que se refiere al modelado del dominio de solución con objetos.

Disponibilidad Fracción de tiempo en que puede usarse el sistema para realizar las tareas normales.

Documento de análisis de requerimientos (RAD) Documento que describe el modelo de análisis.

Documento de diseño de objetos (ODD) Documento que describe el modelo de diseño de objetos. El modelo de diseño de objetos se genera, con frecuencia, a partir de comentarios incrustados en el código fuente.

Documento de diseño del sistema (SDD) Documento que describe el modelo de diseño del sistema.

Dominio de aplicación Representa todos los aspectos del problema del usuario. Esto incluye el ambiente físico en donde ejecutará el sistema, los usuarios y sus procesos de trabajo. El dominio de aplicación está representado por el modelo de análisis durante las actividades de requerimientos y análisis. Al dominio de aplicación también se le llama dominio del problema. *Compare con:* dominio de solución.

Dominio de implementación *Vea* dominio de solución.

Dominio de solución El espacio de todos los sistemas posibles. El dominio de solución es el foco de las actividades de diseño del sistema, de diseño de objetos y de implementación. *Compare con:* dominio de aplicación.

Dominio del problema *Vea* dominio de aplicación.

DRL *Vea* lenguaje de representación de decisiones.

Editor de documentos Papel que representa a la persona que realiza la integración de los documentos.

Editor de fundamentación Es un papel responsable de la recopilación y organización de la información sobre la fundamentación.

Elaboración de prototipos Proceso de diseño y realización de una versión simplificada del sistema para que la evalúen los usuarios o el gerente. Por ejemplo, un prototipo de usabilidad evalúa la usabilidad de diferentes interfaces. Un prototipo de desempeño valora el desempeño de alternativas diferentes.

Enlace Papel de comunicación que es responsable del flujo de información entre dos equipos. Por ejemplo, un enlace de arquitectura representa a un equipo de subsistema en el equipo de arquitectura.

Enlace de arquitectura Papel que representa a los desarrolladores que forman un equipo de subsistema dentro del equipo de arquitectura. Comunican información de su equipo, negocian cambios a las interfaces y aseguran la consistencia de las API en todo el sistema.

Entrega Producto de trabajo destinado al cliente.

Enunciado del problema Documento escrito en colaboración por el cliente y el gerente del proyecto que describe en forma breve el alcance del sistema, incluyendo sus requerimientos de alto nivel, su ambiente de destino, los productos a entregar y los criterios de aceptación.

Equipo Conjunto de participantes que trabaja en un problema común dentro de un proyecto.

Equipo de funcionalidad cruzada Equipo responsable del soporte a equipos de subsistema durante una actividad de funcionalidad cruzada, como la administración de la configuración, la integración y las pruebas.

Equipo de programador en jefe Organización de proyecto jerárquica en la cual el líder del equipo toma todas las decisiones técnicas críticas.

Equipo de subsistema Equipo responsable del desarrollo de un subsistema. *Compare con:* equipo de funcionalidad cruzada.

Error Estado del sistema en que el procesamiento adicional conducirá a una falla.

Escenario Instancia de un caso de uso. Un escenario representa una secuencia concreta de interacciones entre uno o más actores y el sistema.

Escritor técnico *Vea* editor de documentos.

Espacio de trabajo En la administración de la configuración, una biblioteca de promociones.

Especialista del dominio de aplicación Papel de consultor responsable de proporcionar a un proyecto la experiencia sobre el dominio de aplicación.

Especificación del sistema Descripción completa y precisa del sistema desde el punto de vista del usuario. Una especificación del sistema, a diferencia del modelo de análisis, es comprensible para el usuario. En UML, la especificación del sistema se representa con casos de uso.

Estado Condición que se satisface con los valores de los atributos de un objeto o un subsistema.

Estado de acción Estado cuyas transiciones salientes se activan por la terminación de una acción que está asociada con el estado. Las actividades de los diagramas de actividad UML son estados de acción.

Estado estable del proyecto Actividad administrativa durante la cual la administración supervisa y controla el avance del proyecto.

Estereotipo Texto encerrado entre paréntesis angulares asociado a un elemento UML que permite que los modeladores creen nuevos bloques de construcción. Por ejemplo, el estereotipo `<<control>>` asociado a un objeto indica que es un objeto de control.

Estilo arquitectónico Modelo de diseño de sistema general que puede usarse como punto inicial para el diseño del sistema. Algunos ejemplos de estilos arquitectónicos incluyen cliente/servidor, par a par, de tubo y filtro, y modelo/vista/controlador.

Estímulo Mensaje enviado a un objeto o al sistema por un actor u otro objeto que da lugar, por lo general, al llamado de una operación. Algunos ejemplos de estímulos incluyen hacer clic en un botón de interfaz de usuario, la selección de un concepto de menú, el tecleo de un comando o el envío de un paquete de red.

Estructura de reporte Estructura que representa la cadena de control y de reporte de estado.

Evento Suceso relevante en el sistema. Un evento es una instancia de una clase de evento. Algunos ejemplos de eventos son un estímulo de un autor, una temporización o el envío de un mensaje entre dos objetos.

Evitación de defectos Método que trata de prevenir la inserción de defectos cuando se construye el sistema. Algunos ejemplos de técnicas para evitar defectos son las metodologías de desarrollo, la administración de la configuración, la verificación y las revisiones.

Excepción Evento inesperado que sucede durante la ejecución del sistema.

Extensibilidad Cualidad de un sistema que indica con cuánta facilidad puede cambiarse al sistema para que acomode nueva funcionalidad.

Falsificación Proceso por el que se trata de demostrar en forma explícita que un modelo tiene defectos (por ejemplo, el modelo omite detalles relevantes o representa en forma incorrecta los detalles irrelevantes). La elaboración de prototipos y las pruebas son ejemplos de falsificación.

Falla Desviación del comportamiento observado con respecto a lo especificado.

Fase A menudo se usa como sinónimo de actividad o proceso.

Fenómeno Objeto de una realidad tal como es percibido por el modelador. El modelado consiste en la selección de los fenómenos de una realidad que son interesantes, la identificación de sus propiedades comunes y su abstracción en conceptos. *Compare con:* concepto.

Firma En una operación dada, el tuplo formado por los tipos de sus parámetros y el tipo del valor de retorno. Las firmas de las operaciones se especifican durante el diseño del sistema.

Flujo de control Secuencia de ejecución de las operaciones en el sistema. *Compare con:* flujo de datos.

Flujo de datos La secuencia en que los datos se transfieren, usan y transforman en el sistema. *Compare con:* flujo de control.

FRIEND Sistema de administración de incidentes distribuido desarrollado en la Universidad Carnegie Mellon.

Fundamentación Justificación de las decisiones. Por ejemplo, la selección de MiDBMS como sistema de administración de base de datos es una decisión de diseño del sistema. Establecer que MiDBMS es lo bastante confiable y sensible para lograr los objetivos del proyecto es parte de la fundamentación para esta decisión de diseño. También se le llama fundamentación de diseño.

Generalización Tipo de relación entre una clase general y otra más especializada. La clase especializada añade semántica y funcionalidad a la clase general. La clase especializada se llama subclase y la clase general se llama superclase.

Gerente de configuración Papel que representa a los desarrolladores responsables del seguimiento, elaboración de línea base y archivado de productos de trabajo.

Groupware Herramienta de software que soporta el intercambio de información entre un conjunto de participantes. El groupware en diferentes lugares y al mismo tiempo soporta los intercambios síncronos. El groupware en diferentes lugares en diferentes momentos soporta los intercambios asíncronos.

Groupware en diferentes lugares y al mismo tiempo *Vea* groupware.

Grupo de proceso integral *Vea* desarrollo cruzado.

Grupo de procesos Agrupamiento de procesos relacionados. Algunos ejemplos de grupos de procesos son la administración, el predesarrollo, el desarrollo y el posdesarrollo.

Herencia Técnica de reutilización en la cual una clase hija hereda todos los atributos y operaciones de una clase madre. La herencia es un mecanismo que puede usarse para realizar una relación de generalización.

Herencia de establecimiento Es la herencia usada sólo como un mecanismo para la reutilización.

Herencia de interfaz Herencia usada como un medio de subtipeado.

Hilo Paradigma de flujo de control en el cual el sistema crea una cantidad arbitraria de hilos para manejar una cantidad arbitraria de canales de entrada.

IBIS *Vea* sistema de información basado en problemas.

Identificador de versión Número o nombre que identifica en forma única a una versión.

Implementación Actividad durante la cual los desarrolladores traducen el modelo de objetos hacia código.

Ingeniería de interfaz Proyecto de desarrollo en el que se rediseña y reimplementa la interfaz de un sistema. La funcionalidad medular se deja intacta. *Vea también* ingeniería *greenfield* y reingeniería.

Ingeniería de requerimientos Actividad que incluye la obtención de requerimientos y el análisis.

Ingeniería de viaje redondo Actividad de mantenimiento de modelos que combina la ingeniería hacia delante e inversa. Los cambios al modelo de implementación se propagan a los modelos de análisis y diseño mediante ingeniería inversa. Los cambios a los modelos de análisis y diseño se propagan al modelo de implementación mediante ingeniería hacia delante.

Ingeniería *greenfield* Proyecto de desarrollo que comienza a partir de cero. *Vea también* reingeniería e ingeniería de interfaz.

Ingeniería hacia delante Actividad de mantenimiento de modelo durante la cual se genera o actualiza el modelo de implementación a partir del modelo de análisis y diseño. *Compare con:* ingeniería inversa.

Ingeniería inversa Actividad de mantenimiento de modelo durante la cual se generan o actualizan los modelos de análisis y diseño a partir del modelo de implementación. *Compare con:* ingeniería hacia delante.

Ingeniero API Papel cuyo interés es la definición de la API de un subsistema. Este papel se combina a menudo con el de enlace de arquitectura.

Inicio del proyecto Actividad de administración del proyecto durante la cual se definen el alcance y los recursos del proyecto.

Inocuidad Propiedad de un sistema que indica su capacidad para no poner en peligro las vidas humanas, aun en presencia de errores.

Inspección Modo de comunicación calendarizado durante el cual los desarrolladores revisan un producto de trabajo de manera formal entre iguales.

Instalación Actividad durante la cual se instala y prueba el sistema en su ambiente de operación. La instalación también puede incluir el entrenamiento de usuarios.

Instancia Miembro de un tipo de dato específico. Por ejemplo, `1291` es una instancia del tipo de dato `int`, `3.14` es una instancia del tipo de dato `float`.

Interfaz de programación de aplicaciones (API) Conjunto de operaciones especificadas por completo proporcionadas por un subsistema.

Invariante Predicado que siempre es cierto para todas las instancias de una clase.

JAD *Vea* diseño de aplicación conjunto.

JEWEL Ambiente de modelado de emisiones realizado en la Universidad Carnegie Mellon que permite que los usuarios ejecuten, administren, organicen y visualicen simulaciones de emisiones.

KAT *Vea* análisis de conocimiento de tareas.

KPA *Vea* área de proceso principal.

Lanzamiento En comunicaciones, un modo de comunicación calendarizado durante el cual un desarrollador pone a disposición del resto del proyecto una nueva versión de un producto de

trabajo. En la administración de la configuración, una versión que se ha puesto a disposición de manera externa, esto es, del cliente o los usuarios.

Legibilidad Cualidad de un sistema que indica con cuánta facilidad puede comprenderse un sistema a partir de su código fuente.

Lenguaje de modelado unificado (UML) Conjunto de notaciones estándar para la representación de modelos.

Lenguaje de representación de decisiones (DRL) Un modelo de problemas que extiende al IBIS para representar los elementos cualitativos de la toma de decisiones.

Lenguaje de restricción de objetos (OCL) Lenguaje formal, definido como parte del UML, que se usa para expresar restricciones.

Líder de equipo Papel administrativo responsable de la planeación, supervisión y control de un solo equipo.

Línea base Versión del artículo de configuración que se ha revisado de manera formal y con el que se está de acuerdo.

Lista de control de acceso Representación del control de acceso en la cual los derechos legítimos están representados como una lista de pares `{actor, operación}` asociada a cada `clase` que se controla.

Lluvia de ideas Modo de comunicación calendarizado durante el cual los participantes generan gran cantidad de alternativas.

Manejador de prueba Implementación parcial de un componente que ejercita al componente que se prueba. Los manejadores de prueba se usan durante las pruebas unitaria y de integración.

Manejo de excepciones Mecanismo por el cual un sistema trata a una excepción.

Manual de pruebas Documento que describe los casos de prueba que se usan para probar al sistema junto con sus resultados.

Manual de usuario Documento que describe la interfaz de usuario del sistema de tal forma que pueda usarlo un usuario que no esté familiarizado con él.

Marco Conjunto de clases que proporciona una solución general que puede refinarse para proporcionar una aplicación o un subsistema.

Marco de aplicación empresarial Marco específico de la aplicación que se usa para las actividades de negocios empresariales.

Marco de caja blanca Marco que se apoya en la herencia y el enlace dinámico para la extensibilidad. *Compare con:* marco de caja negra.

Marco de caja negra Marco que se apoya en interfaces bien definidas para la extensibilidad. *Compare con:* marco de caja blanca.

Marco de infraestructura Marco que se usa para realizar un subsistema de infraestructura, como una interfaz de usuario o un subsistema de almacenamiento.

Marco middleware Marco utilizado para integrar aplicaciones y componentes distribuidos.

Mecanismo de comunicación Herramienta o procedimiento que puede usarse para transmitir y recibir información. Los mecanismos de comunicación soportan uno o más modos de comunicación. Algunos ejemplos de mecanismos de comunicación son el teléfono, el fax, el correo electrónico y el groupware. Los mecanismos de comunicación pueden ser síncronos o asíncronos, dependiendo de si el emisor y el receptor necesitan estar disponibles al mismo tiempo o no.

Mensaje Mecanismo por el cual un objeto que envía solicita la ejecución de una operación en el objeto que recibe. Un mensaje está compuesto por un nombre y varios argumentos. El objeto que recibe establece la correspondencia entre el nombre del mensaje y una o más operaciones, y pasa los argumentos a la operación. El envío de mensaje termina cuando el resultado de la operación se envía de regreso al objeto que lo envió.

Método En el contexto del desarrollo, una técnica repetible para la resolución de un problema específico. Por ejemplo, una receta es un método para cocinar un plato específico. En el contexto de una clase o un objeto, la implementación de una operación. Por ejemplo, `ajustarHora(t)` es un método de la clase `Reloj`.

Método gancho Método proporcionado por una clase de marco que se pretende se sobrescriba en una subclase para especializar al marco.

Metodología Colección de métodos para la resolución de una clase de problemas. Un libro de cocina de mariscos es una metodología para la preparación de mariscos. Este libro describe una metodología para el manejo de sistemas complejos y cambiantes.

Modelado Actividad durante la cual los participantes construyen una abstracción de un sistema enfocándose en los aspectos interesantes y omitiendo los detalles irrelevantes. Lo que es interesante o irrelevante depende de la tarea en la que se usa el modelo. *Vea también* abstracción.

Modelo Abstracción de un sistema orientada a la simplificación del razonamiento acerca del sistema omitiendo los detalles irrelevantes. Por ejemplo, si el sistema que nos interesa es un distribuidor de boletos para tren, los planos para el distribuidor de boletos, los esquemas de su alambrado eléctrico y los modelos de objetos de su software son modelos del distribuidor de boletos.

Modelo bazar Organización de proyecto que incluye un conjunto de grupos colaboradores dinámico y distribuido. *Compare con:* modelo catedral.

Modelo catedral Organización de proyecto que enfatiza la planeación, la arquitectura del sistema y el control jerárquico. El equipo de programador en jefe es un ejemplo de modelo catedral. *Compare con:* modelo bazar.

Modelo de análisis Modelo del sistema cuyo propósito es ser correcto, completo, consistente, no ambiguo, realista y verificable. El modelo de análisis consta de un modelo funcional, un modelo de objetos y un modelo dinámico.

Modelo de cascada Modelo de ciclo de vida del software en el que todos los procesos de desarrollo suceden de manera secuencial.

Modelo de ciclo de vida basado en problemas Modelo de ciclo de vida del software basado en entidad en el cual se usa un modelo de problemas para supervisar y controlar el avance del proyecto.

Modelo de ciclo de vida del software Abstracción que representa un ciclo de vida del software para comprender, supervisar o controlar un ciclo de vida del software. Algunos ejemplos de modelos de ciclo de vida del software son el modelo de cascada, el modelo V, el modelo espiral de Boehm, el proceso unificado y el modelo de ciclo de vida basado en problemas.

Modelo de ciclo de vida del software centrado en actividad Modelo de ciclo de vida que representa, sobre todo, las actividades de desarrollo. *Compare con:* modelo de ciclo de vida del software centrado en entidad.

Modelo de ciclo de vida del software centrado en entidad Modelo de ciclo de vida del software que representa principalmente a los productos de trabajo elaborados durante el desarrollo. *Compare con:* modelo de ciclo de vida del software centrado en actividad.

Modelo de diente de sierra Modelo de ciclo de vida del software en el cual los desarrolladores le muestran al cliente el progreso con la demostración de prototipos.

Modelo de diente de tiburón Variación del modelo de diente de sierra en la cual los desarrolladores muestran el avance, tanto al cliente como a la administración, con la demostración de prototipos.

Modelo de diseño de objetos Modelo de objetos detallado que representa los objetos de solución que forman el sistema. El modelo de diseño de objetos incluye especificaciones detalladas de clases, incluyendo contratos, tipos, firmas y visibilidades para todas las operaciones públicas.

Modelo de diseño del sistema Descripción de alto nivel del sistema que incluye los objetivos de diseño, la descomposición en subsistemas, la plataforma de hardware/software, la estrategia de almacenamiento persistente, el flujo de control global, la política de control de acceso y el manejo de las condiciones de frontera. El modelo de diseño del sistema representa las decisiones estratégicas tomadas por el equipo de arquitectura que permiten que los equipos de subsistemas trabajen en forma concurrente y cooperen de manera efectiva.

Modelo de madurez de capacidades (CMM) Marco para valorar la madurez de las organizaciones, caracterizado por cinco niveles de madurez.

Modelo de objetos Describe la estructura de un sistema desde el punto de vista de objetos, atributos, asociaciones y operaciones. El modelo de objetos de análisis representa el dominio de aplicación, esto es, los conceptos que son visibles para el usuario. El modelo de diseño de objetos representa el dominio de solución, esto es, los objetos personalizados y hechos que llenan el hueco entre el modelo de análisis y la plataforma de hardware/software.

Modelo de objetos de análisis Modelo de objetos producido durante el análisis. El modelo de objetos de análisis describe los conceptos del dominio de aplicación que manipula el sistema y las interfaces del sistema visibles para el usuario.

Modelo de problemas Modelo que representa la fundamentación de una o más decisiones mediante una gráfica. Los nodos de un modelo de problemas incluyen, por lo general, problemas, propuestas, argumentos, criterios y resoluciones.

Modelo de tarea Modelo de trabajo para un proyecto representado como tareas y sus interdependencias.

Modelo del sistema Modelo que describe el sistema. Algunos ejemplos de modelos del sistema incluyen el modelo de análisis, el modelo de diseño del sistema, el modelo de diseño de objetos y el código fuente.

Modelo dinámico El modelo dinámico describe los componentes del sistema que tienen comportamiento interesante. En este libro representamos el modelo dinámico con diagramas de gráfica de estado, diagramas de secuencia y diagramas de actividad.

Modelo espiral de Boehm Modelo de ciclo de vida del software iterativo e incremental centrado alrededor de la administración del riesgo.

Modelo funcional Describe la funcionalidad del sistema desde el punto de vista del usuario. En este libro representamos el modelo funcional con casos de uso.

Modelo V Variación del modelo de cascada que hace explícitas las dependencias entre los procesos de desarrollo y los de verificación.

Modelo/Vista/Controlador Arquitectura de software de tres patas en la que el conocimiento del dominio se mantiene con modelos de objetos, se despliega mediante objetos de vista y se manipula mediante objetos de control. En este libro, a los modelos de objetos se les llama objetos de entidad y a los objetos de vista se les llama objetos de frontera.

Moderador Papel de administración de reuniones que es responsable de la organización y ejecución de una reunión. El moderador principal escribe la agenda de la reunión, notifica a los participantes en la reunión y se asegura durante la reunión que se siga la agenda. Un moderador secundario apoya el papel del moderador principal haciendo que no se desvíe la reunión.

Modificabilidad Cualidad de un sistema que indica con cuánta facilidad pueden modificarse los modelos existentes.

Modo de comunicación Tipo de intercambio de información que tiene un objetivo y alcance definidos. Algunos ejemplos de modos de comunicación incluyen revisiones del cliente, revisiones de estado y reportes de problemas. Un modo de comunicación puede ser calendarizado o manejado por eventos.

Multiplicidad Conjunto de enteros asignados a un extremo de una asociación que indica la cantidad de vínculos que pueden originarse legítimamente a partir de una instancia de la clase en el extremo de la asociación. Por ejemplo, una asociación que indica que un `Carro` tiene cuatro `Rueda` tiene una multiplicidad de 1 en el extremo `Carro` y de 4 en el extremo `Rueda`.

Nota en UML Comentario añadido a un diagrama.

Notación Conjunto de reglas textuales o gráficas para la representación de un modelo. Por ejemplo, el alfabeto romano es una notación para la representación de palabras. UML es una notación gráfica para la representación de modelos de sistema.

Objetivo Principio de alto nivel que se usa para guiar al proyecto. Los objetivos definen los atributos del sistema que son importantes. Para un vehículo de transporte, la seguridad es un objetivo. Para un software que se vende, el bajo costo es un objetivo.

Objetivo de diseño Cualidad que debe optimizar el sistema. Los objetivos de diseño se infieren a menudo a partir de requerimientos no funcionales y se usan para guiar las decisiones

de diseño. Algunos ejemplos de objetivos de diseño son usabilidad, confiabilidad, seguridad e inocuidad.

Objeto Instancia de una clase. Un objeto tiene una identidad y guarda valores de atributos.

Objeto de aplicación Objeto del modelo de análisis que representa a un concepto del dominio de aplicación. Al objeto de aplicación también se le llama objeto del dominio de aplicación.

Objeto de control Objeto que representa una tarea realizada por el usuario y soportada por el sistema.

Objeto de entidad Objeto que representa la información persistente o de larga vida a la que da seguimiento el sistema.

Objeto de frontera Objeto que representa las interacciones entre el usuario y el sistema.

Objeto de solución Objeto en el modelo de diseño del sistema o de diseño de objetos que representa un concepto del dominio de solución.

Objetos participantes Objetos de análisis que están involucrados en un caso de uso dado.

Obtención de requerimientos Actividad durante la cual los participantes en el proyecto definen el propósito del sistema. La obtención de requerimientos produce el modelo funcional.

OCL *Vea* lenguaje de restricción de objetos.

ODD *Vea* documento de diseño de objetos.

Operación Pieza atómica de comportamiento que proporciona una clase. Un objeto que llama activa la ejecución de una operación enviando un mensaje al objeto sobre el cual deberá ejecutarse la operación.

Organización Conjunto de equipos, papeles, rutas de comunicación y estructuras de reporte orientadas a un proyecto específico (organización de proyecto) o a una clase de proyectos (organización de división o corporativa).

Organización basada en proyecto Organización corporativa en la cual el trabajo se divide de acuerdo con proyectos, es decir, cada participante trabaja sólo en un proyecto en cualquier momento.

Organización de matriz Organización de proyecto que combina a las organizaciones funcional y basada en proyecto. Cada participante reporta a dos gerentes, uno de división (o funcional) y otro de proyecto.

Organización funcional Organización de proyecto en la cual el trabajo se divide de acuerdo con las actividades del proyecto (por ejemplo, obtención de requerimientos, análisis, diseño del sistema, diseño de objetos, implementación, pruebas, administración de la configuración).

OWL Sistema de administración de instalaciones realizado en la Universidad Carnegie Mellon que permite que los usuarios controlen parámetros ambientales (por ejemplo, temperatura, flujo de aire, iluminación) mediante un navegador Web. OWL también permite que el administrador de la instalación localice componentes defectuosos.

Papel En el contexto de una organización, un conjunto de responsabilidades del proyecto asignadas a una persona o a un equipo. Una persona puede ocupar uno o más papeles. Algunos ejemplos de papeles incluyen al analista, al arquitecto del sistema, al probador, al desarrollador,

al gerente y al revisor. En el contexto de un extremo de asociación, un texto que indica el papel de la clase que está en el extremo de la asociación con respecto a la asociación.

Papel de administración Cualquier papel que se refiere a la planeación, supervisión y control del proyecto. Los ejemplos de papeles de administración incluyen al gerente del proyecto y al líder de equipo.

Papel de desarrollador Cualquier papel que esté relacionado con la especificación, diseño y construcción de subsistemas. Algunos ejemplos de papeles de desarrollador son el analista, el arquitecto del sistema, el diseñador de objetos y el implementador.

Papel de funcionalidad cruzada Cualquier papel que se refiera a la coordinación del trabajo de más de un equipo. Algunos ejemplos de papeles de funcionalidad cruzada son el gerente de configuración, el enlace de arquitectura, el probador y el editor de documentos.

Papel de promotor Cualquier papel que se relacione con la promoción del cambio a lo largo de una organización. Algunos ejemplos de papeles de promotor son el promotor de conocimientos, el promotor de procesos y el promotor poderoso.

Paquete en UML Concepto de agrupamiento UML que indica que un conjunto de objetos o clases están relacionados. Los paquetes se usan en los diagramas de caso de uso y de clase para manejar la complejidad asociada con gran cantidad de casos de uso o clases.

Partición Subsistema en una arquitectura par a par.

Participante Persona involucrada en un proyecto de desarrollo de software.

Patrón Adaptador Patrón de diseño que encapsula un componente existente que no fue diseñado para trabajar con el sistema.

Patrón Apoderado Patrón de diseño que encapsula a un cálculo caro. Por ejemplo, puede usarse un patrón Apoderado para diferir la carga de una imagen en memoria hasta que necesite desplegarse.

Patrón arquitectónico *Vea* estilo arquitectónico.

Patrón Comando Patrón que encapsula objetos de control de tal forma que pueden ser tratados de manera uniforme por el sistema.

Patrón Compuesto Patrón de diseño para la representación de jerarquías recurrentes.

Patrón Estrategia Patrón de diseño que encapsula varias implementaciones del mismo algoritmo.

Patrón Fábrica Abstracta Patrón de diseño que encapsula clases concretas proporcionadas por plataformas específicas, como un estilo por ventanas o un sistema operativo.

Patrón Observador Patrón de diseño que desacopla los objetos de entidad con respecto a sus vistas. Las vistas se suscriben a los objetos de entidad en los que están interesados. Los objetos de entidad distribuyen notificaciones a los suscriptores cuando cambia su estado.

Patrón Puente Patrón de diseño que encapsula implementaciones existentes y futuras de una interfaz. Un patrón Puente permite que las implementaciones se sustituyan al tiempo de ejecución.

Petición de aclaraciones Modo de comunicación manejado por eventos durante el cual un participante solicita más información.

Petición de cambio Modo de comunicación manejado por eventos durante el cual los participantes solicitan una característica nueva o modificada en un producto de trabajo. En la administración de la configuración, un reporte formal que solicita la modificación de un artículo de configuración.

Plan de administración de la configuración del software (SCMP) Documento que define los procedimientos y convenciones asociados con la administración de la configuración de un proyecto. Incluye la identificación de los conceptos de configuración, la contabilización de su estado, el proceso para la aprobación de las peticiones de cambio y las actividades de auditoría.

Plan de administración de proyectos de software (SPMP) Documento que controla un proyecto de software. El SPMP define las actividades, productos de trabajo, indicadores de avance y recursos asignados al proyecto. En el SPMP también están definidos los procedimientos administrativos y convenciones aplicables al proyecto, como el reporte de estado, la administración del riesgo y la administración de contingencias.

Portabilidad Cualidad de un sistema que indica con cuánta facilidad puede transportarse el sistema hacia plataformas de hardware/software diferentes.

Poscondición Predicado que debe ser cierto después de que se llama a una operación.

Precondición Predicado que debe ser cierto antes de que se llame a una condición.

Preguntas, opciones y criterios (QOC) Modelo de problemas propuesto por McLean *et al.* que extiende a IBIS para representar información de criterios y valoración.

Probador Papel que se refiere a la planeación, diseño, ejecución y análisis de las pruebas.

Problema Dificultad crítica que no tiene una solución clara.

Problema abierto Dificultad que todavía no se ha resuelto.

Problema cerrado Un problema que se ha resuelto.

Proceso Conjunto de actividades que se realizan para lograr un propósito específico. Algunos ejemplos de procesos incluyen la obtención de requerimientos, el análisis, la administración del proyecto y las pruebas. Un proceso es un sinónimo para una actividad de alto nivel. *Vea también* actividad.

Proceso de desarrollo de software unificado Modelo de ciclo de vida del software iterativo que se caracteriza por ciclos de cuatro fases, llamadas *Comienzo, Elaboración, Construcción* y *Transición.*

Proceso unificado *Vea* proceso de desarrollo de software unificado.

Producto de trabajo Artefacto que se produce durante el desarrollo. Algunos ejemplos de productos de trabajo incluyen el documento de análisis de requerimientos, el documento de diseño del sistema, los prototipos de interfaz de usuario, los estudios de mercado y el sistema entregado.

Producto de trabajo interno Producto de trabajo diseñado sólo para consumo interno del proyecto.

Programación sin ego Organización de proyecto en donde las responsabilidades se asignan a un equipo en vez de a individuos. Los papeles dentro del equipo son intercambiables.

Promoción Versión que se ha puesto a disposición de los demás desarrolladores del proyecto en forma interna. *Vea también* lanzamiento.

Promotor de conocimientos Papel de promotor que se refiere a presionar el cambio por medio de una organización usando conocimiento especializado acerca de los beneficios y limitaciones de las tecnologías o métodos.

Promotor de procesos Papel de promotor interesado en impulsar el cambio a lo largo de una organización usando su conocimiento de los procesos internos de la organización.

Promotor poderoso Papel de promotor interesado en impulsar el cambio a lo largo de una organización usando el conocimiento de la cadena de control.

Propuesta Resolución posible de un problema.

Prueba Actividad durante la cual los desarrolladores encuentran diferencias entre el sistema y sus modelos ejecutando el sistema (o partes de él) con conjuntos de datos de entrada de prueba. Las pruebas incluyen la prueba unitaria, la prueba de integración, la prueba del sistema y la prueba de usabilidad.

Prueba de abajo hacia arriba Estrategia de prueba de integración en la que los componentes se integran en forma incremental, comenzando con los componentes de más bajo nivel. Las pruebas de abajo hacia arriba no requieren ningún *stub* de prueba.

Prueba de aceptación Actividad de prueba del sistema durante la cual el cliente decide si el sistema satisface los criterios de aceptación.

Prueba de arriba hacia abajo Estrategia de pruebas de integración en la cual los componentes se integran incrementalmente comenzando con los componentes de más alto nivel. La prueba de arriba hacia abajo no requiere ningún manejador de prueba.

Prueba de caja blanca Prueba que se enfoca en la estructura interna de un componente. *Compare con:* prueba de caja negra.

Prueba de caja negra Prueba que se enfoca en el comportamiento de entrada/salida de un componente sin considerar su implementación. *Compare con:* prueba de caja blanca.

Prueba de desempeño Actividad de prueba del sistema durante la cual los desarrolladores encuentran diferencias entre los requerimientos no funcionales y el desempeño del sistema.

Prueba de emparedado Estrategia de pruebas de integración que combina las pruebas de arriba hacia abajo y de abajo hacia arriba.

Prueba de gran explosión Estrategia de prueba de integración en la que todos los componentes se prueban juntos inmediatamente después de la prueba unitaria.

Prueba de instalación Actividad de prueba del sistema en la cual los desarrolladores y el cliente prueban el sistema en el ambiente de operación.

Prueba de integración Actividad de prueba durante la cual los desarrolladores encuentran defectos combinando una pequeña cantidad de subsistemas u objetos.

Prueba de usabilidad Validación de un sistema o un modelo mediante el uso de prototipos y simulaciones de un usuario.

Prueba del sistema Actividad de prueba durante la cual los desarrolladores prueban todos los subsistemas como un solo sistema. La prueba del sistema incluye la prueba funcional, la prueba de desempeño, la prueba de aceptación y la prueba de instalación.

Prueba unitaria Prueba de componentes individuales.

Pruebas funcionales Actividad de pruebas del sistema durante la cual los desarrolladores encuentran diferencias entre la funcionalidad observada y el modelo de casos de uso.

RAD *Vea* documento de análisis de requerimientos.

Rama Ruta de desarrollo concurrente bajo la administración de la configuración.

Rastreabilidad Propiedad de un modelo que indica si se puede rastrear un elemento del modelo hasta los requerimientos o fundamentación originales que motivaron su existencia.

Realizabilidad Propiedad de un modelo que indica si puede realizarse lo que representa.

Recursos Bienes que se usan para realizar un trabajo. Los recursos incluyen tiempo, equipo y mano de obra.

Reingeniería Proyecto de desarrollo en el cual se rediseñan y reimplementan un sistema y los procesos de negocios que le acompañan. *Vea también* ingeniería *greenfield* e ingeniería de interfaz.

Relación de comunicación Tipo de relación en un diagrama de caso de uso que indica el flujo de información entre un actor y un caso de uso.

Relación de extensión Tipo de relación en un diagrama de caso de uso que indica que un caso de uso extiende el flujo de eventos hacia otro. Las relaciones de extensión se usan, por lo general, para el modelado del comportamiento excepcional, como el manejo de excepciones y la funcionalidad de ayuda.

Relación de inclusión Tipo de relación en un diagrama de caso de uso que indica que un caso de uso llama a otro caso de uso. Una inclusión de caso de uso es similar a una invocación de método.

Requerimiento Función que debe tener el sistema (un requerimiento funcional) o una restricción sobre el sistema visible ante el usuario (requerimiento no funcional).

Requerimiento funcional Área de funcionalidad que debe soportar el sistema. Los requerimientos funcionales describen las interacciones entre los actores y el sistema, independientemente de la realización del sistema.

Requerimiento no funcional Restricción del sistema visible para el usuario. Los requerimientos no funcionales describen los aspectos del sistema visibles para el usuario que no están relacionados en forma directa con la funcionalidad del sistema. *Vea también* objetivo de diseño.

Resolución Alternativa seleccionada por los participantes para cerrar un problema.

Resolución de problemas Modo de comunicación manejado por eventos durante el cual los participantes logran un consenso o una decisión sobre un problema.

Restricción Regla asociada a un elemento UML que restringe su semántica. Las restricciones pueden mostrarse con una nota que contenga texto en lenguaje natural o una expresión en un lenguaje formal (por ejemplo, OCL).

Reunión Mecanismo síncrono de comunicación durante el cual se presentan, discuten, negocian y resuelven problemas, ya sea frente a frente o mediante teléfono o videoconferencia.

Reunión de arranque Junta que incluye a todos los participantes en el proyecto y que marca el final de la fase de inicio del proyecto y el comienzo del estado estable.

Revisión de estado Modo de comunicación calendarizado durante el cual los líderes de equipo supervisan el estado de su equipo.

Revisión del cliente Modo de comunicación calendarizado durante el cual un cliente supervisa el estado de un proyecto.

Revisión del proyecto Modo de comunicación calendarizado durante el cual un gerente de proyecto supervisa el estado del proyecto.

Revisión post mortem Modo de comunicación calendarizado durante el cual los participantes capturan las lecciones que aprendieron durante el proyecto.

Revisor Papel que representa a las personas que validan los productos de trabajo contra criterios de calidad, como suficiencia, corrección, consistencia y claridad.

Riesgo Área de incertidumbre que puede dar lugar a una desviación en el plan del proyecto (por ejemplo, entrega atrasada, requerimientos no satisfechos, costos mayores a lo presupuestado), incluyendo la falla del proyecto.

Robustez Capacidad para resistir entrada inesperada. Por ejemplo, un componente robusto detecta y maneja argumentos inválidos que se le pasan a sus operaciones. Una interfaz de usuario robusta revisa y maneja entrada inválida del usuario.

SCMP *Vea* plan de administración de la configuración del software.

SDD *Vea* documento de diseño del sistema.

Secretario de actas Papel de reunión responsable del registro de la reunión, en particular, las resoluciones sobre las que se pusieron de acuerdo los participantes y su implementación desde el punto de vista de conceptos de acción.

Seguridad Propiedad de un sistema que indica su capacidad para proteger los recursos en contra del uso no autorizado, ya sea malintencionado o accidental.

Servicio Conjunto de operaciones relacionadas que proporciona una clase.

Seudorrequerimientos Restricción sobre la implementación del sistema impuesta por el cliente.

Sistema Conjunto organizado de partes que se comunican diseñado para un propósito específico. Por ejemplo, un automóvil, compuesto por cuatro ruedas, un chasis, una carrocería y un motor, está diseñado para transportar personas. Un reloj, compuesto por una pila, un circuito, ruedas y manecillas, está diseñado para medir el tiempo.

Sistema de información basado en problemas (IBIS) Modelo de problemas propuesto por Kunz Rittel compuesto por tres tipos de nodos: Problema, Posición y Argumento.

Solución de problemas Actividad de búsqueda que incluye la generación y evaluación de alternativas que atacan a un problema dado, a menudo por ensayo y error.

SPMP *Vea* plan de administración de proyectos de software.

***Stub* de prueba** Implementación parcial de un componente del que depende el componente a probar. Los *stubs* de prueba se usan para aislar componentes durante las pruebas unitaria y de integración, y permiten que los componentes se prueben aunque todavía no se hayan implementado sus componentes dependientes.

Subclase Clase especializada en una relación de generalización. *Vea también* generalización.

Subsistema En general, una parte más pequeña y simple de un sistema más grande. En el diseño del sistema, un componente de software bien definido que proporciona varios servicios a los demás componentes. Algunos ejemplos de subsistemas incluyen los subsistemas de almacenamiento (administración de datos persistentes), los subsistemas de interfaz de usuario (administración de la interacción con el usuario), los subsistemas de red (administración de la comunicación con los demás subsistemas por medio de una red).

Suficiencia o completitud Propiedad de un modelo que indica si están modelados todos los fenómenos relevantes o no. Un modelo está incompleto si uno o más de los fenómenos relevantes no tiene un concepto correspondiente en el modelo.

Superclase Clase general en una relación de generalización. *Vea también* generalización.

Tabla de acceso global Representación de los derechos de acceso en la cual cada derecho legítimo está representado por un tuplo `{actor, clase, operación}`.

Tarea Unidad atómica de trabajo que puede administrarse. Las tareas consumen recursos y producen uno o más productos de trabajo.

Terminación del proyecto Actividad administrativa durante la cual se concluye el proyecto: el sistema se entrega y lo acepta el cliente.

Tipo de dato Abstracción de un conjunto de valores en el contexto de un lenguaje de programación. Por ejemplo, `int` es el tipo de dato de Java que abstrae a todos los valores enteros.

Tipo de dato abstracto Tipo de dato cuya estructura está encapsulada ante quien llama.

Tolerancia a fallas Método dirigido a la construcción de sistemas que no fallen en presencia de defectos. La capacidad para soportar defectos sin fallar.

Tomador de tiempo Papel responsable del seguimiento del tiempo en una reunión para que el moderador principal pueda acelerar la resolución (o presentación) de un problema si es necesario.

Transición Cambio de estado posible asociado con un evento.

Transición compleja Transición con diversos estados de origen o diversos estados de destino. Una transición representa la unión o división de varios hilos de control. Las transiciones complejas se usan en los diagramas de actividad para indicar la sincronización de actividades.

UML *Vea* lenguaje de modelado unificado.

Usabilidad Cualidad de un sistema que indica con cuánta facilidad pueden interactuar los usuarios con el sistema.

Usuario Papel que representa a las personas que interactúan en forma directa con el sistema cuando realizan su trabajo.

Variante Versiones que se pretende que coexistan. Por ejemplo, si un sistema puede ejecutar en plataformas diferentes, el sistema tiene una variante para cada plataforma (por ejemplo, una variante Windows, una variante Macintosh, una variante Linux).

Verificabilidad Propiedad de un modelo que indica si puede falsificársele o no.

Verificación Conjunto de métodos formales que tratan de detectar defectos sin ejecutar el sistema.

Versión Estado de un artículo de configuración o un agregado AC en un momento bien definido del tiempo. A la versión de un agregado AC se le llama configuración.

Vínculo Instancia de una asociación. Un vínculo conecta a dos objetos.

Visibilidad de atributo Especifica si otras clases pueden tener acceso al atributo o no.

Vista Subconjunto de un modelo. Las vistas sólo se enfocan en elementos del modelo seleccionados para que sean más comprensibles.

Bibliografía

[Abbott, 1983] R. Abbott, "Program design by informal English descriptions", *Communications of the ACM*, vol. 26, núm. 11, 1983.

[Allen, 1985] T. J. Allen, *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization*, 2a. ed., MIT Press, Cambridge, MA, 1995.

[Ambler, 1998] S. W. Ambler, *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press, Nueva York, 1998.

[Apple, 1989] Apple Computers, Inc., *Macintosh Programmers Workshop Pascal 3.0 Reference*, Apple Computers, Cupertino, CA, 1989.

[Babich, 1986] W. A. Babich, *Software Configuration Management*, Addison-Wesley, Reading, MA, 1986.

[Bass *et al.*, 1999] L. Bass, P. Clements y R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1999.

[Berliner, 1990] B. Berliner, "CVS II: parallelizing software development", *Proceedings of the 1990 USENIX Conference*, Washington, DC, enero de 1990, págs. 22-26.

[Bersoff *et al.*, 1980] E. H. Bersoff, V. D. Henderson y S. G. Siegel, *Software Configuration Management: An Investment in Product Integrity*, Prentice Hall, Englewood Cliffs, NJ, 1980.

[Bezier, 1990] B. Bezier, *Software Testing Techniques*, 2a. ed., Van Nostrand, Nueva York, 1990.

[Binder, 1994] R. V. Binder, "Testing object-oriented systems: A status report", *American Programmer*, abril de 1994.

[Birrer, 1993] E. T. Birrer, "Frameworks in the financial engineering domain: An experience report", *ECOOP'93 Proceedings*, Lecture Notes in Computer Science, núm. 707, 1993.

[Boehm, 1987] B. Boehm, "A spiral model of software development and enhancement", *Software Engineering Project Management*, 1987, págs. 128-142.

[Boehm, 1991] B. Boehm, "Software risk management: Principles and practices", *IEEE Software*, vol. 1, 1991, págs. 32-41.

[Boehm *et al.*, 1995] B. Boehm, P. Bose, E. Horowitz y M. J. Lee, "Software requirements negotiation and renegotiation aids: A theory-W based spiral approach", *Proceedings of the ICSE-17*, Seattle, WA, 1995.

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications, 2a. ed., Benjamin/Cummings*, Redwood City, CA, 1994.

[Booch *et al.*, 1998] G. Booch, J. Rumbaugh e I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

[Brooks, 1975] F. P. Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, MA, 1975.

[Brown *et al.*, 1999] W. J. Brown, H. W. McCormick y S. W. Thomas, *AntiPatterns and Patterns in Software Configuration Management*, Wiley, Nueva York, 1999.

[Bruegge, 1992] B. Bruegge, "Teaching an Industry-oriented Software Engineering Course", *Software Enginering Education, SEI Conference*, Lecture Notes in Computer Sciences, Springer Verlag, San Diego, CA, vol. 640, octubre de 1992, págs. 65-87.

[Bruegge, 1994] B. Bruegge, "From toy systems to real system development", *Improvements in Software Engineering Education*, taller de la sección alemana de la ACM, B. G. Teubner Verlag, Stuttgart, febrero de 1994, págs. 62-72.

[Bruegge *et al.*, 1992] B. Bruegge, J. Blythe, J. Jackson y J. Shufelt, "Object-oriented system modeling with OMT", *Conference Proceedings OOPSLA '92 (Object-Oriented Programming Systems, Languages, and Applications)*, octubre de 1992, págs. 359-376.

[Bruegge *et al.*, 1993] B. Bruegge, T. Gottschalk y B. Luo, "A framework for dynamic program analyzers", *OOPSLA'93 (Object-Oriented Programming Systems, Languages, and Applications)*, Washington, DC, septiembre de 1993, págs. 65-82.

[Bruegge *et al.*, 1994] B. Bruegge, K. O'Toole y D. Rothenberger, "Design considerations for an accident management system", en M. Brodie, M. Jarke, M. Papazoglou (eds.), *Proceedings of the Second International Conference on Cooperative Information Systems*, University of Toronto Press, Toronto, mayo de 1994, págs. 90-100.

[Bruegge *et al.*, 1995] B. Bruegge, E. Riedel, G. McRae y T. Russel, "GEMS: An Environmental Modeling System", *IEEE Journal for Computational Science and Engineering*, septiembre de 1995, págs. 55-68.

[Bruegge *et al.*, 1997] B. Bruegge, S. Chang, T. Fenton, B. Fernandes, V. Hartkopf, T. Kim, R. Pravia y A. Sharma, "Turning lightbulbs into objects", ponencia de experiencia presentada en la *OOPSLA'97*, Atlanta, 1997.

[Bruegge y Bennington, 1995] B. Bruegge y B. Bennington, "Applications of mobile computing and communication", *IEEE Journal on Personal Communications, Special Issue on Mobile Computing*, febrero de 1996, págs. 64-71.

[Bruegge y Coyne, 1993] B. Bruegge y R. Coyne, "Model-based software engineering in larger scale project courses", *IFIP Transactions on Computer Science and Technology*, vol. A-40, 1993, págs. 273-287.

[Bruegge y Coyne, 1994] B. Bruegge y R. Coyne, "Teaching iterative object-oriented development: Lessons and directions", en Jorge L. Díaz-Herrera (ed.), *7th Conference on Software Engineering Education*, Lecture Notes in Computer Science, Springer Verlag, vol. 750, enero de 1994, págs. 413–427.

[Bruegge y Riedel, 1994] B. Bruegge y E. Riedel, "A geographic environmental modeling system: Towards an object-oriented framework", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP-94)*, Lecture Notes in Computer Science, Springer Verlag, Bolonia, Italia, julio de 1994.

[Buckingham Shum y Hammond, 1994] S. Buckingham Shum y N. Hammond, "Argumentation-based design rationale: What use at what cost?", *International Journal of Human-Computer Studies*, vol. 40, 1994, págs. 603-652.

[Buschmann *et al.*, 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, Chichester, Inglaterra, 1996.

[Carr *et al.*, 1993] M. J. Carr, S. L. Konda, I. Monarch, F. C. Ulrich y C. F. Walker, *Taxonomy-Based Risk Identification*, Technical Report CMU/SEI-93-TR-6, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[Carroll, 1995] J. M. Carroll (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development*, Wiley, Nueva York, 1995.

[Charette, 1989] R. N. Charette, *Software Engineering Risk Analysis and Management,* McGraw-Hill, Nueva York, 1989.

[Christel y Kang, 1992] M. G. Christel y K. C. Kang, *Issues in requirements elicitation*, Technical Report CMU/SEI-92-TR-12, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Coad *et al.*, 1995] P. Coad, D. North y M. Mayfield, *Object Models: Strategies, Patterns, & Applications*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Conklin y Burgess-Yakemovic, 1991] J. Conklin y K. C. Burgess-Yakemovic, "A process-oriented approach to design rationale", *Human-Computer Interaction*, vol. 6, 1991, págs. 357-391.

[Conradi y Westfechtel, 1998] R. Conradi y B. Westfechtel, "Version models for software configuration management", *ACM Computing Surveys*, vol. 30, núm. 2, junio de 1998.

[Constantine y Lockwood, 1999] L. L. Constantine y L. A. D. Lockwood, *Software for Use*, Addison-Wesley, Reading, MA, 1999.

[Coyne *et al.*, 1995] R. Coyne, B. Bruegge, A. Dutoit y D. Rothenberger, "Teaching more comprehensive model-based software engineering: Experience with Objectory's use case approach", en Linda Ibraham (ed), *8th Conference on Software Engineering Education*, Lecture Notes in Computer Science, Springer Verlag, Berlín, abril de 1995, págs. 339-374.

[Dart, 1991] S. Dart, "Concepts in configuration management systems", *Third International Software Configuration Management Workshop*, ACM, junio de 1991.

[Day y Zimmermann, 1983] J. D. Day y H. Zimmermann, "The OSI Reference Model", *Proceedings of the IEEE*, vol. 71, diciembre de 1983, págs. 1334-1340.

[De Marco, 1978] T. De Marco, *Structured Analysis and System Specification*, Yourdon, Nueva York, 1978.

[DIAMOND, 1995] *DIAMOND Project Documentation*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1995-1996.

[D'Souza y Wills, 1999] D. F. D'Souza y A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, The Addison-Wesley Object Technology Series, Addison-Wesley, Reading, MA, 1999.

[Dumas y Redish, 1998] Dumas y Redish, *A Practical Guide to Usability Testing*, Ablex, NJ, 1993.

[Dutoit, 1996] A. H. Dutoit, "The rôle of communication in team-based software engineering projects", tesis de doctorado, Carnegie Mellon University, Pittsburgh, PA, diciembre de 1996.

[Dutoit *et al.*, 1996] A. H. Dutoit, B. Bruegge y R. F. Coyne, "The use of an issue-based model in a team-based software engineering course", *Conference Proceedings of Software Engineering: Education and Practice* (SEEP'96), Dunedin, NZ, enero de 1996.

[Dutoit y Bruegge, 1998] A. H. Dutoit y B. Bruegge, "Communication metrics for software development", *IEEE Transactions on Software Engineering*, agosto de 1998.

[Erman *et al.*, 1980] L. D. Erman, F. Hayes-Roth *et al.*, "The Hearsay-II Speech-Understanding System: Integrating knowledge to resolve uncertainty", *ACM Computing Surveys*, vol. 12, núm. 2, 1980, págs. 213-253.

[Fagan, 1976] M. E. Fagan, "Design and code inspections to reduce errors in program development", *IBM System Journal*, vol. 15, núm. 3, 1976, págs. 182-211.

[Fayad y Hamu, 1997] M. E. Fayad y D. S. Hamu, "Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection", *The Communications of ACM*, 1997.

[Feiler y Tichy, 1998] P. Feiler y W. Tichy, "Propagator: A family of patterns", en *Proceedings of TOOLS-23'97*, Santa Bárbara, CA, 28 de julio–1 de agosto de 1997.

[Feynman, 1988] R. P. Feynman, "Personal observations on the reliability of the Shuttle", en [Rogers *et al.*, 1986]. Este artículo también se encuentra disponible en varios lugares de la Web, por ejemplo, http://www.virtualschool.edu/mon/Social-Construction/ FeynmanChallengerRpt.html.

[Fischer *et al.*, 1991] R. Fischer, W. Ury y B. Patton, *Getting to Yes: Negotiating Agreement Without Giving In*, 2a., ed., Penguin Books, 1991.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, MA, 1997.

[FRIEND, 1994] *FRIEND Project Documentation*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 1994.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

[Grudin, 1988] J. Grudin, "Why CSCW applications fail: Problems in design and evaluation of organization interfaces", *Proceedings of CSCW'88, Portland*, OR, 1988.

[Grudin, 1990] J. Grudin, "Obstacles to user involvement in interface design in large product development organizations", *Proceeding IFIP INTERACT'90 Third International Conference on Human-Computer Interaction*, Cambridge, Inglaterra, agosto de 1990.

[Hammer y Champy, 1993] M. Hammer y J. Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*, Harper Business, Nueva York, 1993.

[Harel, 1987] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 1987, págs. 231-274.

[Hartkopf *et al.*, 1997] V. Hartkopf, V. Loftness, A. Mahdavi, S. Lee y J. Shankavarm, "An integrated approach to design and engineering of intelligent buildings—The Intelligent Workplace at Carnegie Mellon University", *Automation in Construction*, vol. 6, 1997, págs. 401-415.

[Hauschildt y Gemuenden, 1998] J. Hauschildt y H. G. Gemuenden, *Promotoren*, Gabler, Wiesbaden, Alemania, 1998.

[Hillier y Lieberman, 1967] F. S. Hillier y G. J. Lieberman, *Introduction to Operation Research*, Holden-Day, San Francisco, 1967.

[Horn, 1992] B. Horn, "Constraint patterns as a basis for object-oriented programming", en *Proceedings of the OOPSLA'92*, Vancouver, Canadá, 1992.

[Hueni *et al.*, 1995] H. Hueni, R. Johnson y R. Engel, "A framework for network protocol software", *Proceedings of OOPSLA*, Austin, TX, octubre de 1995.

[Humphrey, 1989] W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.

[iContract] Java Design by Contract Tool, http://www.reliable-systems.com/tools/iContract/iContract.htm.

[IEEE, 1997] *IEEE Standards Collection Software Engineering*, IEEE, Piscataway, NJ, 1997.

[IEEE Std. 1042-1987] *IEEE Guide to Software Configuration Management*, IEEE Standards Board, septiembre de 1987 (confirmado en diciembre de 1993), en [IEEE, 1997].

[IEEE Std. 982-1989] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Standards Board, julio de 1989, en [IEEE, 1997].

[IEEE Std. 828-1990] *IEEE Standard for Software Configuration Management Plans*, IEEE Standards Board, septiembre de 1990, en [IEEE, 1997].

[IEEE Std. 829-1991] *IEEE Standard for Software Test Documentation*, IEEE Standards Board, marzo de 1991, en [IEEE, 1997].

[IEEE Std. 1058.1-1993] *IEEE Standard for Software Project Management Plans*, IEEE Computer Society, Nueva York, 1993, en [IEEE, 1997].

[IEEE Std. 1074-1995] *IEEE Standard for Developing Software Life Cycle Processes*, IEEE Computer Society, Nueva York, 1995, en [IEEE, 1997].

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading, MA, 1995.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson y G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.

[Jacobson *et al.*, 1999] I. Jacobson, G. Booch y J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Javadoc, 1999a] Sun Microsystems, Javadoc homepage, http://java.sun.com/products/jdk/javadoc/.

[Javadoc, 1999b] Sun Microsystems, "How to write doc comments for Javadoc", http:// java.sun.com/products/jdk/javadoc/writingdoccomments.html.

[JCA, 1998] *Java Cryptography Architecture*, JDK Documentation, Javasoft, 1998.

[JDBC, 1998] *JDBC™—Connecting Java and Databases*, JDK Documentation, Javasoft, 1998.

[Jensen y Tonies, 1979] R. W. Jensen y C. C. Tonies, *Software Engineering*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[JFC, 1999] *Java Foundation Classes*, JDK Documentation, Javasoft, 1999.

[Johnson, 1992] P. Johnson, *Human Computer Interaction: Psychology, Task Analysis and Software Engineering*, McGraw-Hill Int., Londres, 1992.

[Johnson y Foote, 1988] R. Johnson y B. Foote, "Designing reusable classes", *Journal of Object-Oriented Programming*, vol. 1, núm. 5, 1988, págs. 22-35.

[Jones, 1977] T. C. Jones, "Programmer quality and programmer productivity", IBM Technical Report TR–02.764, 1977.

[Kayser, 1990] T. A. Kayser, *Mining Group Gold*, Serif, El Segundo, CA, 1990.

[Kemerer, 1997] C. F. Kemerer, *Software Project Management: Readings and Cases*, Irwin/McGraw-Hill, Boston, MA, 1997.

[Knuth, 1986] D. E. Knuth, *The TeXbook*, Addison-Wesley, Reading, MA, 1986.

[Kompanek *et al.*, 1996] A. Kompanek, A. Houghton, H. Karatassos, A. Wetmore y B. Bruegge, "JEWEL: A distributed system for emissions modeling", *Proceedings of the Conference for Air and Waste Management*, Nashville, TN, junio de 1996.

[Kunz y Rittel, 1970] W. Kunz y H. Rittel, "Issues as elements of information systems", Working Paper No. 131, Institut für Grundlagen der Plannung, Universität Stuttgart, Alemania, 1970.

[Larman, 1998] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Leblang, 1994] D. Leblang, "The CM challenge: Configuration management that works", en W. F. Tichy (ed.), *Configuration Management*, vol. 2, *Trends in Software*. Wiley, Nueva York.

[Lee, 1990] J. Lee, "A qualitative decision management system", en P. H. Winston & S. Shellard (eds.), *Artificial Intelligence at MIT: Expanding Frontiers*. MIT Press, Cambridge, MA, vol. 1, 1990, págs. 104-133.

[Lee, 1997] J. Lee, "Design rationale systems: Understanding the issues", *IEEE Expert*, mayo/junio de 1997.

[Lions, 1996] J.-L. Lions, *ARIANE 5 Flight 501 Failure: Report by the Inquiry Board*, http:// www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 1996.

[Liskov y Guttag, 1986] B. Liskov y J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, McGraw-Hill, Nueva York, 1986.

[Macaulay, 1996] L. Macaulay, *Requirements Engineering*, Springer Verlag, Londres, 1996.

[MacLean *et al.*, 1991] A. MacLean, R. M. Young, V. Bellotti y T. Moran, "Questions, options, and criteria: Elements of design space analysis", *Human-Computer Interaction*, vol. 6, 1996, págs. 201-250.

[Martin y Odell, 1992] J. Martin y J. J. Odell, *Object-Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[McCabe, 1976] T. McCabe, "A software complexity measure", *IEEE Transactions on Software Engineering*, vol. 2, núm. 12, diciembre de 1976.

[Mellor y Shlaer, 1998] S. Mellor y S. Shlaer, *Recursive Design Approach*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Meyer, 1997] B. Meyer, *Object-Oriented Software Construction*, 2a., ed., Prentice Hall, Upper Saddle River, NJ, 1997.

[Microsoft, 1995] Microsoft, "Chapter 9: Open Database Connectivity (ODBC) 2.0 fundamentals", *Microsoft Windows Operating Systems and Services Architecture,* Microsoft Corp., 1995.

[MIL Std. 480] MIL Std. 480, U.S. Department of Defense, Washington, DC.

[Miller, 1956] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information", *Psychological Review*, vol. 63, 1956, págs. 81-97.

[Moran y Carroll, 1996] T. P. Moran y J. M. Carroll (eds.), *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Mowbray y Malveau, 1997] T. J. Mowbray y R. C. Malveau, *CORBA Design Patterns*, Wiley, Nueva York, 1997.

[Myers, 1979] G. J. Myers, *The Art of Software Testing*, Wiley, Nueva York, 1979.

[Neumann, 1995] P. G. Neumann, *Computer-Related Risks*, Addison-Wesley, Reading, MA, 1995.

[Nielsen, 1993] J. Nielsen, *Usability Engineering*, Academic, Nueva York, 1993.

[Nielsen y Mack, 1994] J. Nielsen y R. L. Mack (eds.), *Usability Inspection Methods*, Wiley, Nueva York, 1994.

[Nye y O'Reilly, 1992] A. Nye y T. O'Reilly, *X Toolkit Intrinsics Programming Manual: OSF/Motif 1.1 Edition for X11 Release 5—The Definitive Guides to the X Windows Systems*, vol. 4. O'Reilly & Associates, Sebastopol, CA, 1992.

[OMG, 1995] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Wiley, Nueva York, 1995.

[OMG, 1998] Object Management Group, *OMG Unified Modeling Language Specification,* Framingham, MA, 1998, http://www.omg.org.

[Orlikowski, 1992] W. J. Orlikowski, "Learning from Notes: Organizational issues in groupware implementation", en *Conference on Computer-Supported Cooperative Work Proceedings*, Toronto, Canadá, 1992.

[OWL, 1996] *OWL Project Documentation*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Paech, 1998] B. Paech, "The four levels of use case description", *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations for Software Quality*, Pisa, Italia, junio de 1998.

[Parnas y Weiss, 1985] D. L. Parnas y D. M. Weiss, "Active design reviews: principles and practice", *Proceedings of the Eight International Conference on Software Engineering*, Londres, Inglaterra, agosto de 1985, págs. 132-136.

[Paulk *et al.*, 1995] M. C. Paulk, C. V. Weber y B. Curtis (eds.), *The Capability Maturity Mode: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA, 1995.

[Perforce] Perforce, Inc., 2420 Santa Clara Ave., Alameda, CA.

[Pfleeger, 1991] S. L. Pfleeger, *Software Engineering: The Production of Quality Software*, 2a. ed., Macmillan, 1991.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach*, Clarendon, Oxford, 1992.

[POSIX, 1990] *Portable Operating System Interface for Computing Environments*, IEEE Std. 1003.1, IEEE, 1990.

[Potts, 1996] C. Potts, "Supporting software design: Integrating design methods and design rationale", en T. P. Moran y J. M. Carroll (eds.), *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Potts *et al.*, 1994] C. Potts, K. Tkahashi y A. I. Anton, "Inquiry-based requirements analysis", *IEEE Software*, vol. 11, núm. 2, 1994, págs. 21-32.

[Potts y Bruns, 1988] C. Potts y G. Bruns, "Recording the Reasons for Design Decisions", en *Proceedings of the 10th International Conference on Software Engineering*, 1988, págs. 418-427.

[Purvis *et al*, 1996] M. Purvis, M. Purvis y P. Jones, "A group collaboration tool for software engineering projects", *Conference Proceedings of Software Engineering: Education and Practice (SEEP'96)*, Dunedin, NZ, enero de 1996.

[Rational, 1998] *Rationale Rose 98: Using Rose*, Rational Software Corp., Cupertino, CA, 1998.

[Raymond, 1998] E. Raymond, "The cathedral and the bazaar", disponible en http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html, 1998.

[Reeves *et al.*, 1992] B. Reeves y F. Shipman, "Supporting communication between designers with artifact-centered eVolving information spaces", en *Conference on Computer-Supported Cooperative Work Proceedings*, Toronto, Canadá, 1992.

[RMI, 1998] *Java Remote Method Invocation*, JDK Documentation, Javasoft, 1998.

[Rogers *et al.*, 1986] *The Presidential Commission on the Space Shuttle Challenger Accident Report*, Washington, DC, junio de 1986.

[Rowen, 1990] R. B. Rowen, "Software project management under incomplete and ambiguous specifications", *IEEE Transactions on Engineering Management*, vol. 37, núm. 1, 1990.

[Royse, 1970] W. W. Royse, "Managing the development of large software systems", en *Tutorial: Software Engineering Project Management*, IEEE Computer Society, Washington, DC, 1970, págs. 118-127.

[Rubin, 1994] J. Rubin, *Handbook of Usability Testing*, Wiley, Nueva York, 1994.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Saeki, 1995] M. Saeki, "Communication, collaboration, and cooperation in software development—How should we support group work in software development?", en *Asia-Pacific Software Engineering Conference Proceedings*, Brisbane, Australia, 1995.

[Schmidt, 1997] D. C. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software", en Peter Salus (ed.), *Handbook of Programming Languages*, vol. 1, MacMillan Computer, 1997.

[Seaman y Basili, 1997] C. B. Seaman y V. R. Basili, "An empirical study of communication in code inspections", en *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, 1997.

[Shaw y Garlan, 1996] M. Shaw y D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ, 1996.

[Shipman y McCall, 1997] F. M. Shipman III y R. J. McCall, "Integrating Different Perspectives on Design Rationale: Supporting the Emergence of Design Rationale from Design Communication", *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, vol. 11, núm. 2, 1997.

[Siewiorek y Swarz, 1992] D. P. Siewiorek y R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2a. ed., Digital, Burlington, MA, 1992.

[Silberschatz *et al*, 1991] A. Silberschatz, J. Peterson y P. Galvin, *Operating System Concepts*, 3a. ed., Addison-Wesley, Reading, MA, 1991.

[Simon, 1970] H. A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1970.

[Spivey, 1989] J. M. Spivey, *The Z Notation, A Reference Manual*, Prentice Hall Int., Hertfordshire, Inglaterra, 1989.

[Steelman, 1978] *Requirements for High Order Computer Programming Languages: Steelman*, U.S. Department of Defense, Washington, DC, 1978.

[Subrahmanian *et al.*, 1997] E. Subrahmanian, Y. Reich, S. L. Konda, A. Dutoit, D. Cunningham, R. Patrick, M. Thomas y A. W. Westerberg, "The n-dim approach to building design support systems", *Proceedings of ASME Design Theory and Methodology DTM '97*, ASME, Nueva York, 1997.

[Szyperski, 1998] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press, Addison-Wesley, Nueva York, 1998.

[Tanenbaum, 1996] A. S. Tanenbaum, *Computer Networks*, 3a. ed., Prentice Hall, Upper Saddle River, NJ, 1996.

[Teamwave, 1997] Teamwave Inc., http://www.teamwave.com, 1997.

[Tichy, 1985] W. Tichy, "RCS—a system for version control", *Software Practice and Experience*, vol. 15, núm. 7, 1985.

[Turner y Robson, 1993] C. D. Turner y D. J. Robson, "The state-based testing of object-oriented programs", *Conference on Software Maintenance*, septiembre de 1993, págs. 302-310.

[Vaughan, 1996] D. Vaughan, *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA*, The University of Chicago Press, Chicago, IL, 1996.

[Walker *et al.*, 1980] B. J. Walker, R. A. Kemmerer y G. J. Popek, "Specification and verification of the UCLA Unix security kernel", *Communications of the ACM*, vol. 23, núm. 2, 1980, págs. 118-131.

[Weinand *et al.*, 1988] A. Weinand, E. Gamma y R. Marty, "ET++—An object-oriented application framework in C++", en *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, San Diego, CA, septiembre de 1988.

[Weinberg, 1971] G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand, Nueva York, 1971.

[Wilson y Ostrem, 1999] G. Wilson y J. Ostrem, *WebObjects Developer's Guide*, Apple Computers, Cupertino, CA, 1998.

[Wirfs-Brock, 1995] R. Wirfs-Brock, "Design objects and their interactions: A brief look at responsibility-driven design", en J. M. Carroll (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development*, Wiley, Nueva York, 1995.

[Wirfs-Brock *et al.*, 1990] R. Wirfs-Brock, B. Wilkerson y L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Wood y Silver, 1989] J. Wood y D. Silver, *Joint Application Design*®, Wiley, Nueva York, 1989.

[Wordsworth, 1992] J. B. Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, Reading, MA, 1992.

[Zultner, 1993] R. E. Zultner, "TQM for technical teams", *Communications of the ACM*, vol. 36, núm. 10, 1993, págs. 79-91.

Índice

A

B

C

CH

D

E

F

LL

M

N

O

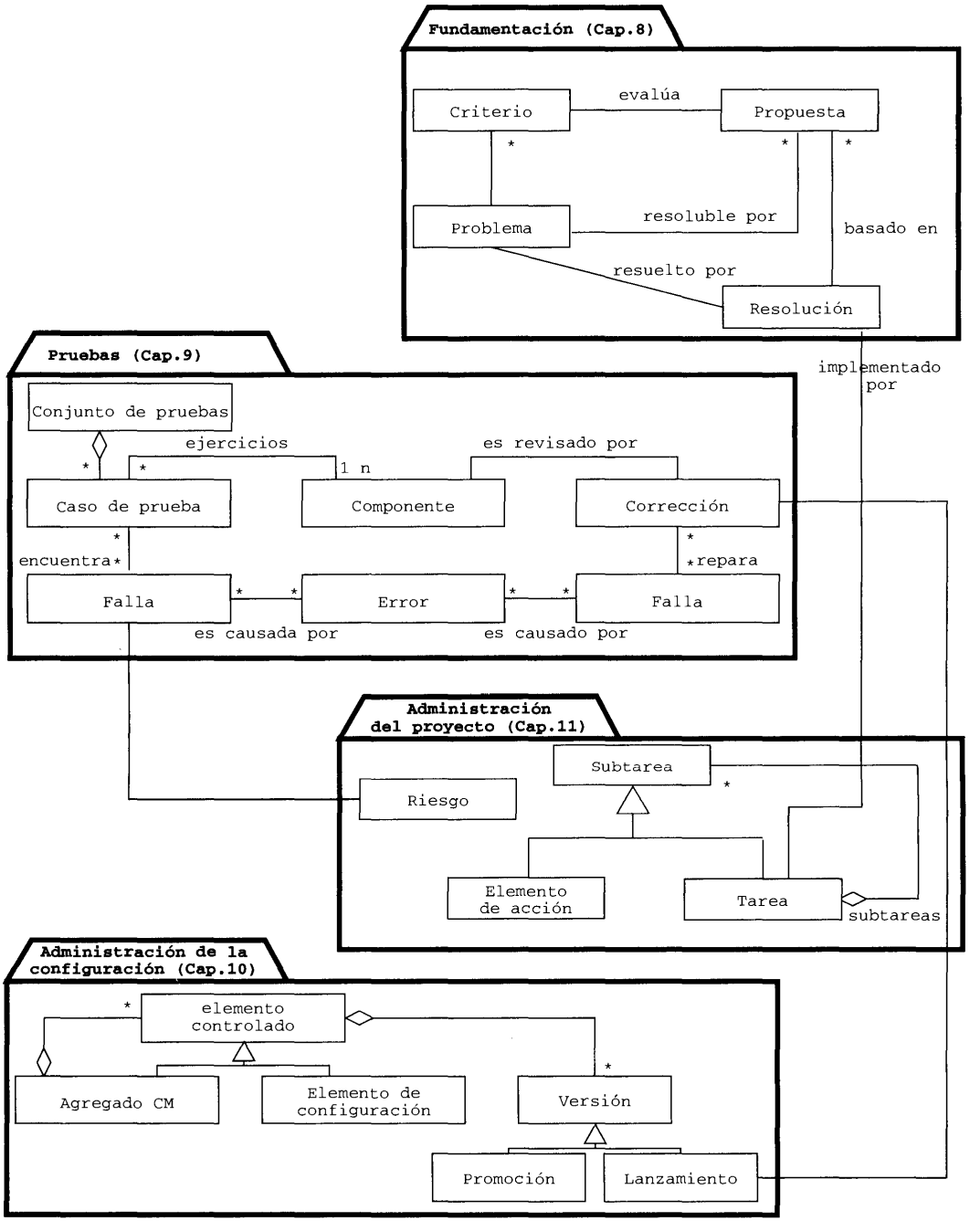
P

Q

R

S

Manejo del cambio



Patrones de diseño

Compuesto (137, 173, 177, 502) Este patrón representa una jerarquía reiterativa. Los servicios relacionados con la jerarquía contenedora se factorizan usando herencia, permitiendo que un sistema trate en forma uniforme a una hoja o a un compuesto.

Observador (184, 504) Este patrón permite que se mantenga la consistencia entre los estados de un `Publicador` y varios `Suscriptor`.

Fachada (198, 503) El patrón `Fachada` reduce las dependencias entre clases mediante el encapsulamiento de un subsistema con una interfaz unificada simple.

Adaptador (202, 499) Este patrón encapsula un fragmento de código heredado que no fue diseñado para trabajar con el sistema. También limita el impacto de la sustitución del fragmento de código heredado por un componente diferente.

Puente (206, 342, 500) Este patrón desacopla la interfaz de una clase y su implementación. A diferencia del patrón Adaptador, el desarrollador no está limitado por un fragmento de código existente.

Proxy (Apoderado) (210, 273, 505) Este patrón mejora el desempeño o la seguridad de un sistema mediante el retraso de cálculos costosos, usando memoria sólo cuando se necesita, o revisando el acceso antes de cargar un objeto en memoria.

Estrategia (212, 506) Este patrón desacopla un algoritmo con respecto a su implementación. Sirve al mismo propósito que los patrones de adaptador y puente, a excepción de que la unidad encapsulada es un comportamiento.

Comando (215, 501) Este patrón permite el encapsulamiento del control para que las peticiones del usuario puedan tratarse en forma uniforme, independientemente de la petición específica. Este patrón protege a estos objetos contra cambios a causa de nueva funcionalidad.

Fábrica abstracta (267, 498) Este patrón se usa para aislar una aplicación de las clases concretas proporcionadas por una plataforma específica, como un estilo de ventanas o un sistema operativo.

Notaciones

Diagramas de caso de uso UML (25, 39) Los diagramas de caso de uso se emplean para representar la funcionalidad del sistema, como es vista por un actor.

Diagramas de clase UML (25, 45, 135, 137) Los diagramas de clase se usan para representar la estructura del sistema, desde el punto de vista de subsistemas, objetos, sus atributos y sus operaciones.

Diagramas de secuencia UML (26, 50) Los diagramas de secuencia representan el comportamiento en función de una serie de interacciones entre un conjunto de objetos.

Diagramas de gráfica de estado UML (26, 52) Los diagramas de gráfica de estado representan el comportamiento como una máquina de estado en función de eventos y transiciones.

Diagramas de actividad (28, 54) Los diagramas de actividad son diagramas de flujo que representan el comportamiento como un conjunto de actividades y transiciones.

Diagramas de despliegue UML (188) Los diagramas de despliegue representan la correspondencia entre los componentes de software y los nodos de hardware.

Modelos de problema (290) Los modelos de problema representan la justificación de las decisiones en función de problemas, propuestas, argumentos, criterios y resoluciones.

Gráficas PERT (425) Las gráficas PERT representan la división del trabajo en tareas y restricciones de ordenamiento.