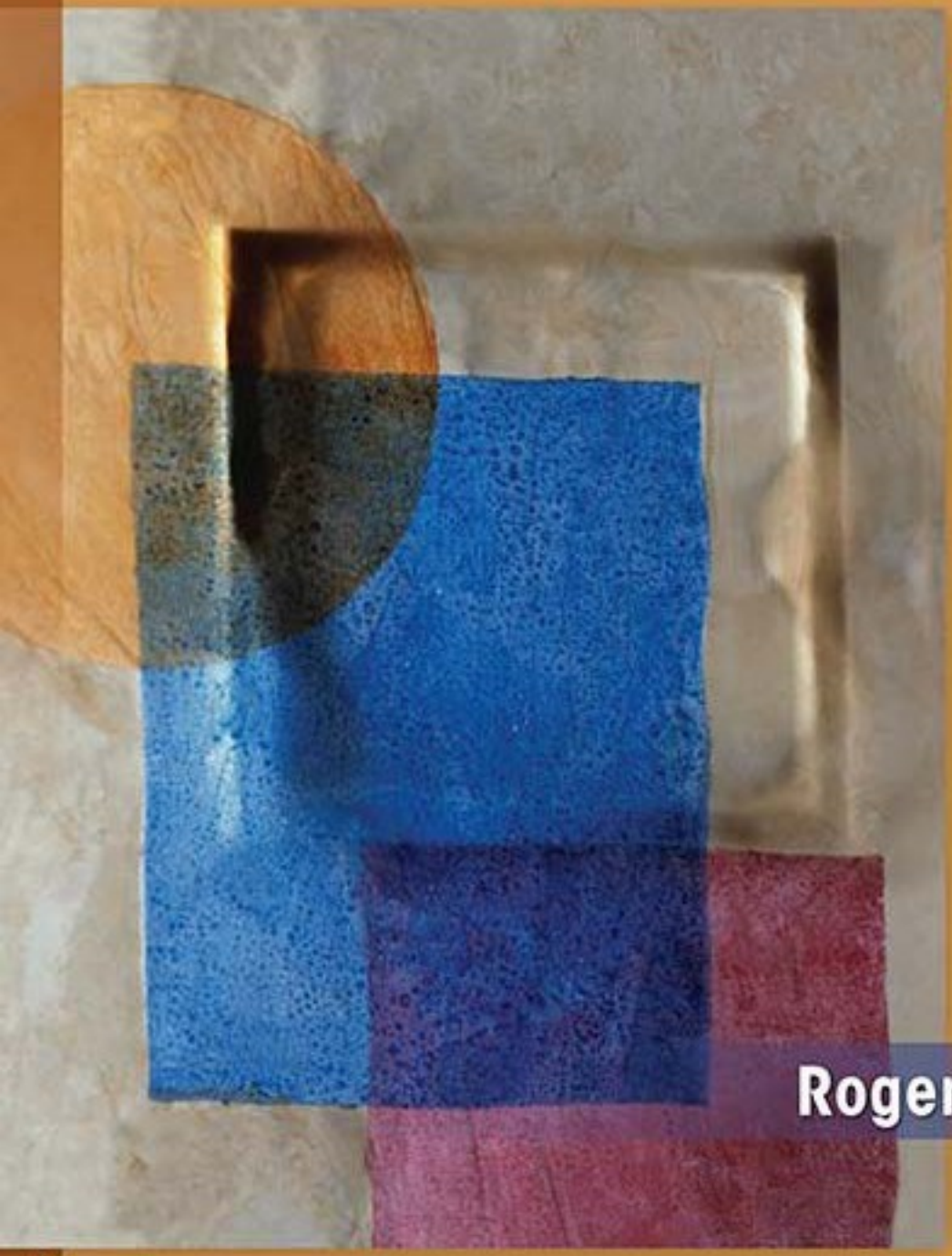
Ingeniería del software

Un enfoque práctico



Séptima edición

Roger S. Pressman

Mc Graw Hill

Ingeniería del software

UN ENFOQUE PRÁCTICO

SÉPTIMA EDICIÓN

Roger S. Pressman, Ph.D.
University of Connecticut

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID
NUEVA YORK • SAN JUAN • SANTIAGO • SÃO PAULO • AUCKLAND • LONDRES • MILÁN
MONTREAL • NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Editor sponsor: Pablo Roig Vázquez
Coordinadora editorial: Marcela I. Rocha Martínez
Editora de desarrollo: María Teresa Zapata Terrazas
Supervisor de producción: Zeferino García García

Traductores: Víctor Campos Olguín
Javier Enríquez Brito
Revisión técnica: Carlos Villegas Quezada
Bárbaro Jorge Ferro Castro

INGENIERÍA DEL SOFTWARE. UN ENFOQUE PRÁCTICO
Séptima edición

Educación

A Subsidiary of ***The McGraw-Hill*** *Companies, Inc.*
Prolongación Paseo de la Reforma 1015, Torre A
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-607-15-0314-5
(ISBN edición anterior: 970-10-5473-3)

Traducido de la séptima edición de SOFTWARE ENGINEERING. A PRACTITIONER'S APPROACH.
Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2010 by The McGraw-Hill Companies, Inc. All rights reserved.
978-0-07-337597-7

1234567890 109876543210

Impreso en México *Printed in Mexico*

The ***McGraw-Hill*** *Companies*

Acerca del autor

Roger S. Pressman es una autoridad internacionalmente reconocida en el mejoramiento del proceso del software y en las tecnologías de la ingeniería del mismo. Durante casi cuatro décadas ha trabajado como ingeniero de software, gestor, profesor, escritor y consultor, especializado en temas de ingeniería del software.

Como profesional y gestor industrial, el doctor Pressman trabajó en el desarrollo de sistemas CAD/CAM para aplicaciones de ingeniería y fabricación avanzadas. También ha tenido posiciones de responsabilidad en la programación científica y de sistemas.

Después de recibir su doctorado en ingeniería por parte de la Universidad de Connecticut, Pressman se dedicó a la academia, donde se convirtió en profesor asociado de la cátedra Bullard en ingeniería de cómputo de la Universidad de Bridgeport, y en director del Centro de Diseño y Fabricación Asistidos por Computadora de dicha universidad.

En la actualidad, el doctor Pressman es presidente de R. S. Pressman & Associates, Inc., una empresa de consultoría especializada en métodos y capacitación en ingeniería del software. Trabaja como consultor principal y diseñó y desarrolló *Ingeniería del software esencial*, un video curricular completo acerca de ingeniería del software, y *Consultor de procesos*, un sistema autodirigido para el mejoramiento del proceso de software. Ambos productos los utilizan miles de compañías en todo el mundo. Más recientemente, trabajó en colaboración con *EdistaLearning*, en India, para desarrollar capacitación abarcadora basada en internet acerca de ingeniería del software.

El doctor Pressman ha escrito muchos artículos técnicos, es colaborador regular en revistas periódicas industriales y autor de siete libros técnicos. Además de *Ingeniería del software: un enfoque práctico*, es coautor de *Web Engineering* (McGraw-Hill), uno de los primeros libros en aplicar un conjunto personalizado de principios y prácticas de la ingeniería del software al desarrollo de sistemas y aplicaciones basados en web. También escribió el premiado *A Manager's Guide to Software Engineering* (McGraw-Hill); *Making Software Engineering Happen* (Prentice hall), el primer libro en abordar los problemas administrativos cruciales asociados con el mejoramiento del proceso de software; y *Software Shock* (Dorset House), un tratamiento que se enfoca en el software y su impacto en los negocios y la sociedad. Pressman ha formado parte de los consejos editoriales de varias publicaciones industriales y durante muchos años fue editor de la columna "Manager" en *IEEE Software*.

Además, es un orador bien conocido, y ha sido el orador principal en muchas conferencias industriales importantes. Es miembro de IEEE, y de Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu y Pi Tau Sigma.

En el lado personal, Pressman vive en el sur de Florida con su esposa, Bárbara. Atleta de toda la vida, sigue siendo un serio jugador de tenis (4.5 en el programa estadounidense de calificación de tenis, NTRP) y un golfista con un *handicap* de un solo dígito. En su tiempo libre escribió dos novelas, *Aymara Bridge* y *The Puppeteer*, y tiene planes para escribir una más.

Contenido breve

CONTENIDO

Prefacio

Cuando el software de computadora triunfa (al satisfacer las necesidades de las personas que lo usan, trabajar sin fallos durante largos periodos, será fácil de modificar e incluso más fácil de usar) puede y debe cambiar las cosas a fin de mejorar. Pero cuando el software fracasa (cuando sus usuarios no están satisfechos, es proclive al error, es difícil de cambiar e incluso más difícil de usar) pueden ocurrir, y ocurren, cosas malas. Todo mundo quiere construir software que haga mejor las cosas y que evite las malas que acechan en la sombra de los esfuerzos fallidos. Para triunfar, se necesita disciplina al momento de diseñar y construir el software. Es necesario un enfoque de ingeniería.

Han pasado casi tres décadas desde que se escribió la primera edición de este libro. Durante ese tiempo, la ingeniería del software evolucionó desde una oscura idea practicada por un número relativamente pequeño de fanáticos hasta una legítima disciplina de la ingeniería. En la actualidad, se le reconoce como una materia merecedora de investigación seria, estudio concienzudo y debate turbulento. A lo largo de toda la industria, el ingeniero de software sustituyó al programador como el título laboral de preferencia. Los modelos de proceso de software, los métodos de ingeniería de software y las herramientas del software se adoptaron exitosamente a través de un amplio espectro de segmentos industriales.

Aunque los gestores y profesionales reconocen por igual la necesidad de un enfoque del software más disciplinado, continúan debatiendo la forma en la que la disciplina debe aplicarse. Muchos individuos y compañías todavía desarrollan el software de manera fortuita, incluso cuando construyen sistemas para atender las tecnologías más avanzadas de la actualidad. Muchos profesionales y estudiantes no están conscientes de los métodos modernos. Como resultado, la calidad del software que producen es deficiente y ocurren cosas malas. Además, continúa el debate y la controversia en torno de la verdadera naturaleza del enfoque de la ingeniería del software. El estatus de la ingeniería del software es un estudio en contrastes. Las actitudes han cambiado, se ha progresado, pero todavía falta mucho por hacer antes de que la disciplina alcance madurez plena.

La séptima edición de *Ingeniería del software: un enfoque práctico* tiene la intención de funcionar como guía hacia una disciplina de ingeniería que madura. Como las seis ediciones que la precedieron, la séptima se dirige a estudiantes y profesionales, y conserva su atractivo como guía para el profesional industrial y como introducción abarcadora para el estudiante en los niveles superiores de pregrado o en el primer año de graduado.

La séptima edición es considerablemente más que una simple actualización. El libro se revisó y reestructuró para mejorar el flujo pedagógico y enfatizar nuevos e importantes procesos y prácticas de la ingeniería del software. Además, este texto cuenta con un paquete de complementos, los cuales están disponibles para los profesores que lo adopten. Consulte con el representante de McGraw-Hill local.

La séptima edición. Los 32 capítulos de la séptima edición se reorganizaron en cinco partes. Esta organización, que difiere considerablemente de la sexta edición, se realizó para dividir mejor los temas y ayudar a los profesores que tal vez no tengan tiempo para completar todo el libro en un semestre.

La parte 1, *El proceso*, presenta varias visiones diferentes del proceso de software, considera todos los modelos de proceso importantes y aborda el debate entre las filosofías de proceso

prescriptivo y ágil. La parte 2, *Modelado*, presenta los métodos de análisis y diseño con énfasis en las técnicas orientadas a objeto y al modelado UML. También se considera el diseño basado en patrón y el diseño para aplicaciones web. La parte 3, *Gestión de la calidad*, presenta los conceptos, procedimientos, técnicas y métodos que permiten a un equipo de software valorar la calidad del software, revisar los productos de trabajo de la ingeniería del software, realizar procedimientos SQA y aplicar una estrategia y tácticas de prueba efectivas. Además, también se considera el modelado formal y los métodos de verificación. La parte 4, *Gestión de proyectos de software*, presenta temas que son relevantes a quienes planean, gestionan y controlan un proyecto de desarrollo de software. La parte 5, *Temas avanzados*, considera el mejoramiento del proceso de software y las tendencias en la ingeniería del software. Al continuar con la tradición de las ediciones pasadas, a lo largo del libro se usa una serie de recuadros para presentar las experiencias y tribulaciones de un equipo de software (ficticio) y para proporcionar materiales complementarios acerca de los métodos y herramientas que son relevantes para los temas del capítulo. Dos nuevos apéndices proporcionan breves tutoriales acerca del UML y del pensamiento orientado a objeto para quienes no estén familiarizados con estos importantes temas.

La organización en cinco partes de la séptima edición permite al profesor "englobar" los temas con base en el tiempo disponible y las necesidades del estudiante. Un curso de todo un semestre podría construirse en torno de uno o más de las cinco partes. Uno de evaluación de ingeniería del software seleccionaría capítulos de las cinco. Uno de ingeniería del software que enfatice el análisis y el diseño elegiría temas de las partes 1 y 2. Un curso de ingeniería del software orientado a pruebas seleccionaría temas de las partes 1 y 3, con una breve incursión en la parte 2. Un "curso administrativo" subrayaría las partes 1 y 4.

Reconocimientos. Mi trabajo en las siete ediciones de *Ingeniería del software: un enfoque práctico* ha sido el proyecto técnico continuo más largo de mi vida. Aun cuando la escritura cesó, la información extraída de la literatura técnica continúa asimilándose y organizándose, y las críticas y sugerencias de los lectores en todo el mundo se evalúan y catalogan. Por esta razón, agradezco a los muchos autores de libros, ponencias y artículos (tanto en copia dura como en medios electrónicos) que me han proporcionado comprensión, ideas y comentarios adicionales durante casi 30 años.

Agradezco especialmente a Tim Lethbridge, de la Universidad de Ottawa, quien me auxilió en el desarrollo de los ejemplos UML y OCL, y quien desarrolló el estudio de caso que acompaña a este libro, y a Dale Skrien, de Colby College, quien desarrolló el tutorial UML en el apéndice 1. Su asistencia y sus comentarios fueron invaluables. Un agradecimiento especial también para Bruce Maxim, de la Universidad de Michigan-Dearborn, quien me auxilió en el desarrollo de gran parte del contenido pedagógico en el sitio web que acompaña a este libro. Finalmente, quiero agradecer a los revisores de la séptima edición: sus comentarios a profundidad y críticas bien pensadas han sido invaluables.

Osman Balci,
Virginia Tech University
Max Fomitchev,
Penn State University
Jerry (Zeyu) Gao,
San Jose State University
Guillermo Garcia,
Universidad Alfonso X Madrid
Pablo Gervas,
Universidad Complutense de Madrid
SK Jain,
National Institute of Technology Hamirpur
Saeed Monemi,
Cal Poly Pomona
Ahmed Salem,
California State University
Vasudeva Varma,
IIIT Hyderabad

El contenido de la séptima edición de *Ingeniería del software: un enfoque práctico* fue conformado por profesionales de la industria, profesores universitarios y estudiantes, quienes usaron ediciones anteriores del libro y tomaron tiempo para comunicar sus sugerencias, críticas e ideas.

Mi agradecimiento a cada uno de ustedes. Además, mi reconocimiento personal a nuestros muchos clientes industriales en todo el mundo, quienes, ciertamente, me enseñaron tanto o más de lo que yo podría haberles enseñado en algún momento.

Conforme las ediciones de este libro evolucionaban, mis hijos, Mathew y Michael, crecieron de niños a hombres. Su madurez, carácter y éxito en el mundo real han sido una inspiración para mí. Nada me ha llenado más de orgullo. Y finalmente, a Bárbara, mi amor y agradecimiento por tolerar las muchísimas horas en la oficina y por alentar todavía otra edición de "el libro".

Roger S. Pressman

CAPÍTULO

1

EL SOFTWARE Y LA INGENIERÍA DE SOFTWARE

Tenía la apariencia clásica de un alto ejecutivo de una compañía importante de software —a la mitad de los 40, con las sienes comenzando a encanecer, esbelto y atlético, con ojos que penetraban al observador mientras hablaba—. Pero lo que dijo me dejó anonadado. "El software ha *muerto*".

Pestañeé con sorpresa y sonreí. "Bromeas, ¿verdad? El mundo es dirigido con software y tu empresa se ha beneficiado mucho de ello. ¡No ha muerto! Está vivo y en desarrollo."

Movió su cabeza de manera enfática. "No, está muerto... al menos como lo conocimos."

Me apoyé en el escritorio. "Continúa."

Habló al tiempo que golpeaba en la mesa con énfasis. "El concepto antiguo del software —lo compras, lo posees y tu trabajo consiste en administrarlo— está llegando a su fin. Hoy día, con Web 2.0 y la computación ubicua cada vez más fuerte, vamos a ver una generación de software por completo diferente. Se distribuirá por internet y se verá exactamente como si estuviera instalado en el equipo de cómputo de cada usuario... pero se encontrará en un servidor remoto."

Tuve que estar de acuerdo. "Entonces, tu vida será mucho más sencilla. Tus muchachos no tendrán que preocuparse por las cinco diferentes versiones de la misma App que utilizan decenas de miles de usuarios."

Sonrió. "Absolutamente. Sólo la versión más reciente estará en nuestros servidores. Cuando hagamos un cambio o corrección, actualizaremos funcionalidad y contenido a cada usuario. Todos lo tendrán en forma instantánea..."

Hice una mueca. "Pero si cometes un error, todos lo tendrán también instantáneamente".

Él se rió entre dientes. "Es verdad, por eso estamos redoblando nuestros esfuerzos para hacer una ingeniería de software aún mejor. El problema es que tenemos que hacerlo 'rápido' porque el mercado se ha acelerado en cada área de aplicación."

UNA MIRADA RÁPIDA

¿Qué es? El software de computadora es el producto que construyen los programadores profesionales y al que después le dan mantenimiento durante un largo tiempo. Incluye programas que se ejecutan en una computadora de cualquier tamaño y arquitectura, contenido que se presenta a medida de que se ejecutan los programas de cómputo e información descriptiva tanto en una copia dura como en formatos virtuales que engloban virtualmente a cualesquiera medios electrónicos. La ingeniería de software está formada por un proceso, un conjunto de métodos (prácticas) y un arreglo de herramientas que permite a los profesionales elaborar software de cómputo de alta calidad.

¿Quién lo hace? Los ingenieros de software elaboran y dan mantenimiento al software, y virtualmente cada persona lo emplea en el mundo industrializado, ya sea en forma directa o indirecta.

¿Por qué es importante? El software es importante porque afecta a casi todos los aspectos de nuestras vidas y ha invadido nuestro comercio, cultura y actividades cotidianas. La ingeniería de software es importante porque nos permite construir sistemas complejos en un tiempo razonable y con alta calidad.

¿Cuáles son los pasos? El software de computadora se construye del mismo modo que cualquier producto exitoso, con la aplicación de un proceso ágil y adaptable para obtener un resultado de mucha calidad, que satisfaga las necesidades de las personas que usarán el producto. En estos pasos se aplica el enfoque de la ingeniería de software.

¿Cuál es el producto final? Desde el punto de vista de un ingeniero de software, el producto final es el conjunto de programas, contenido (datos) y otros productos terminados que constituyen el software de computadora. Pero desde la perspectiva del usuario, el producto final es la información resultante que de algún modo hace mejor al mundo en el que vive.

¿Cómo me aseguro de que lo hice bien? Lea el resto de este libro, seleccione aquellas ideas que sean aplicables al software que usted hace y aplíquelas a su trabajo.

Me recargué en la espalda y coloqué mis manos en mi nuca. "Ya sabes lo que se dice... puedes tenerlo rápido o bien hecho o barato. Escoge dos de estas características..."

"Elijo rápido y bien hecho", dijo mientras comenzaba a levantarse.

También me incorporé. "Entonces realmente necesitas ingeniería de software."

"Ya lo sé", dijo mientras salía. "El problema es que tenemos que llegar a convencer a otra generación más de técnicos de que así es..."

¿Está muerto *realmente* el software? Si lo estuviera, usted no estaría leyendo este libro...

Cita:

"Las ideas y los descubrimientos tecnológicos son los motores que impulsan el crecimiento económico."

Wall Street Journal

El software de computadora sigue siendo la tecnología más importante en la escena mundial. Y también es un ejemplo magnífico de la ley de las consecuencias inesperadas. Hace 50 años, nadie hubiera podido predecir que el software se convertiría en una tecnología indispensable para los negocios, ciencias e ingeniería, ni que permitiría la creación de tecnologías nuevas (por ejemplo, ingeniería genética y nanotecnología), la ampliación de tecnologías ya existentes (telecomunicaciones) y el cambio radical de tecnologías antiguas (la industria de la impresión); tampoco que el software sería la fuerza que impulsaría la revolución de las computadoras personales, que productos de software empacados se comprarían en los supermercados, que el software evolucionaría poco a poco de un producto a un servicio cuando compañías de software "sobre pedido" proporcionaran funcionalidad justo a tiempo a través de un navegador web, que una compañía de software sería más grande y tendría más influencia que casi todas las empresas de la era industrial, que una vasta red llamada internet sería operada con software y evolucionaría y cambiaría todo, desde la investigación en bibliotecas y la compra de productos para el consumidor hasta el discurso político y los hábitos de encuentro de los adultos jóvenes (y no tan jóvenes).

Nadie pudo prever que habría software incrustado en sistemas de toda clase: de transporte, médicos, de telecomunicaciones, militares, industriales, de entretenimiento, en máquinas de oficina... la lista es casi infinita. Y si usted cree en la ley de las consecuencias inesperadas, hay muchos efectos que aún no podemos predecir.

Nadie podía anticipar que millones de programas de computadora tendrían que ser corregidos, adaptados y mejorados a medida que transcurriera el tiempo. Ni que la carga de ejecutar estas actividades de "mantenimiento" absorbería más personas y recursos que todo el trabajo aplicado a la creación de software nuevo.

Conforme ha aumentado la importancia del software, la comunidad de programadores ha tratado continuamente de desarrollar tecnologías que hagan más fácil, rápida y barata la elaboración de programas de cómputo de alta calidad. Algunas de estas tecnologías se dirigen a un dominio específico de aplicaciones (por ejemplo, diseño e implantación de un sitio web), otras se centran en un dominio tecnológico (sistemas orientados a objetos o programación orientada a aspectos), otros más tienen una base amplia (sistemas operativos, como Linux). Sin embargo, todavía falta por desarrollarse una tecnología de software que haga todo esto, y hay pocas probabilidades de que surja una en el futuro. A pesar de ello, las personas basan sus trabajos, confort, seguridad, diversiones, decisiones y sus propias vidas en software de computadora. Más vale que esté bien hecho.

Este libro presenta una estructura que puede ser utilizada por aquellos que hacen software de cómputo —personas que deben hacerlo bien—. La estructura incluye un proceso, un conjunto de métodos y unas herramientas que llamamos *ingeniería de software*.

1.1 La naturaleza del software

En la actualidad, el software tiene un papel dual. Es un producto y al mismo tiempo es el vehículo para entregar un producto. En su forma de producto, brinda el potencial de cómputo incorporado en el hardware de cómputo o, con más amplitud, en una red de computadoras a las

PUNTO CLAVE

El software es tanto un producto como un vehículo para entregar un producto.

que se accede por medio de un hardware local. Ya sea que resida en un teléfono móvil u opere en el interior de una computadora central, el software es un transformador de información —produce, administra, adquiere, modifica, despliega o transmite información que puede ser tan simple como un solo bit o tan compleja como una presentación con multimedios generada a partir de datos obtenidos de decenas de fuentes independientes—. Como vehículo utilizado para distribuir el producto, el software actúa como la base para el control de la computadora (sistemas operativos), para la comunicación de información (redes) y para la creación y control de otros programas (herramientas y ambientes de software).

El software distribuye el producto más importante de nuestro tiempo: *información*. Transforma los datos personales (por ejemplo, las transacciones financieras de un individuo) de modo que puedan ser más útiles en un contexto local, administra la información de negocios para mejorar la competitividad, provee una vía para las redes mundiales de información (la internet) y brinda los medios para obtener información en todas sus formas.

Cita:

"El software es un lugar donde se siembran sueños y se cosechan pesadillas, una ciénega abstracta y mística en la que terribles demonios luchan contra panaceas mágicas, un mundo de hombres lobo y balas de plata."

Brad J. Cox

En el último medio siglo, el papel del software de cómputo ha sufrido un cambio significativo. Las notables mejoras en el funcionamiento del hardware, los profundos cambios en las arquitecturas de computadora, el gran incremento en la memoria y capacidad de almacenamiento, y una amplia variedad de opciones de entradas y salidas exóticas han propiciado la existencia de sistemas basados en computadora más sofisticados y complejos. Cuando un sistema tiene éxito, la sofisticación y complejidad producen resultados deslumbrantes, pero también plantean problemas enormes para aquellos que deben construir sistemas complejos.

En la actualidad, la enorme industria del software se ha convertido en un factor dominante en las economías del mundo industrializado. Equipos de especialistas de software, cada uno centrado en una parte de la tecnología que se requiere para llegar a una aplicación compleja, han reemplazado al programador solitario de los primeros tiempos. A pesar de ello, las preguntas que se hacía aquel programador son las mismas que surgen cuando se construyen sistemas modernos basados en computadora:[1]

- ¿Por qué se requiere tanto tiempo para terminar el software?
- ¿Por qué son tan altos los costos de desarrollo?
- ¿Por qué no podemos detectar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué dedicamos tanto tiempo y esfuerzo a mantener los programas existentes?
- ¿Por qué seguimos con dificultades para medir el avance mientras se desarrolla y mantiene el software?

Éstas y muchas otras preguntas, denotan la preocupación sobre el software y la manera en que se desarrolla, preocupación que ha llevado a la adopción de la práctica de la ingeniería del software.

1.1.1 Definición de software

En la actualidad, la mayoría de profesionales y muchos usuarios tienen la fuerte sensación de que entienden el software. Pero, ¿es así?

La descripción que daría un libro de texto sobre software sería más o menos así:

? ¿Cómo se define *software*?

> El software es: 1) instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados; 2) estructuras de datos que permiten que los progra-

1 En un excelente libro de ensayos sobre el negocio del software, Tom DeMarco [DeM95] defiende el punto de vista contrario. Dice: "En lugar de preguntar por qué el software cuesta tanto, necesitamos comenzar a preguntar: ¿Qué hemos hecho para hacer posible que el software actual cueste tan poco? La respuesta a esa pregunta nos ayudará a continuar el extraordinario nivel de logro que siempre ha distinguido a la industria del software."

mas manipulen en forma adecuada la información, y 3) información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas.

No hay duda de que podrían darse definiciones más completas.

Pero es probable que una definición más formal no mejore de manera apreciable nuestra comprensión. Para asimilar lo anterior, es importante examinar las características del software que lo hacen diferente de otros objetos que construyen los seres humanos. El software es elemento de un sistema lógico y no de uno físico. Por tanto, tiene características que difieren considerablemente de las del hardware:

1. *El software se desarrolla o modifica con intelecto; no se manufactura en el sentido clásico.*

PUNTO CLAVE
El software se modifica con intelecto, no se manufactura.

Aunque hay algunas similitudes entre el desarrollo de software y la fabricación de hardware, las dos actividades son diferentes en lo fundamental. En ambas, la alta calidad se logra a través de un buen diseño, pero la fase de manufactura del hardware introduce problemas de calidad que no existen (o que se corrigen con facilidad) en el software. Ambas actividades dependen de personas, pero la relación entre los individuos dedicados y el trabajo logrado es diferente por completo (véase el capítulo 24). Las dos actividades requieren la construcción de un "producto", pero los enfoques son distintos. Los costos del software se concentran en la ingeniería. Esto significa que los proyectos de software no pueden administrarse como si fueran proyectos de manufactura.

2. *El software no se "desgasta".*

PUNTO CLAVE
El software no se desgasta, pero sí se deteriora.

La figura 1.1 ilustra la tasa de falla del hardware como función del tiempo. La relación, que es frecuente llamar "curva de tina", indica que el hardware presenta una tasa de fallas relativamente elevada en una etapa temprana de su vida (fallas que con frecuencia son atribuibles a defectos de diseño o manufactura); los defectos se corrigen y la tasa de fallas se abate a un nivel estable (muy bajo, por fortuna) durante cierto tiempo. No obstante, conforme pasa el tiempo, la tasa de fallas aumenta de nuevo a medida que los componentes del hardware resienten los efectos acumulativos de suciedad, vibración, abuso, temperaturas extremas y muchos otros inconvenientes ambientales. En pocas palabras, el hardware comienza a *desgastarse*.

El software no es susceptible a los problemas ambientales que hacen que el hardware se desgaste. Por tanto, en teoría, la curva de la tasa de fallas adopta la forma de la "curva idealizada" que se aprecia en la figura 1.2. Los defectos ocultos ocasionarán ta-

FIGURA 1.1
Curva de fallas del hardware

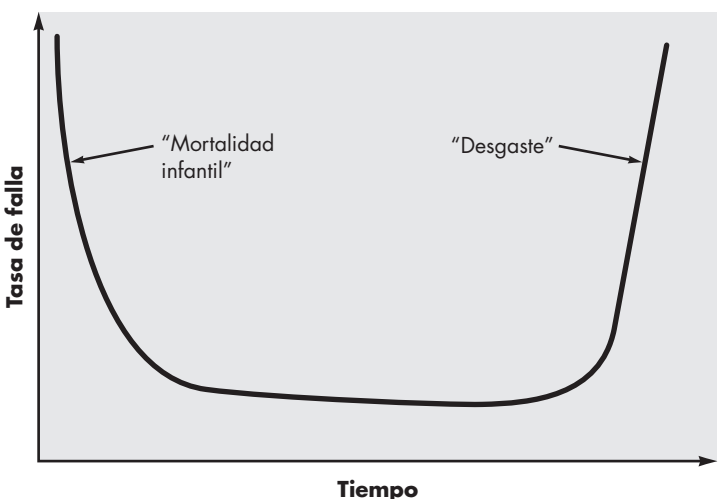
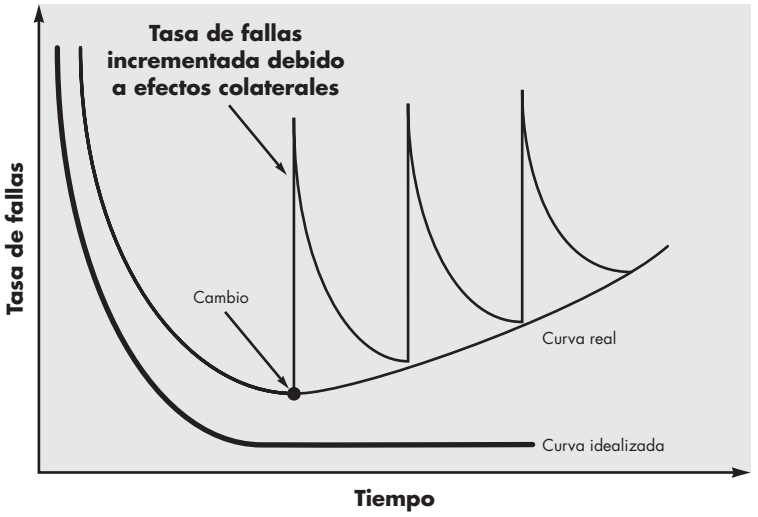


FIGURA 1.2
Curvas de falla del software



CONSEJO

Si quiere reducir el deterioro del software, tendrá que mejorar su diseño (capítulos 8 a 13).

PUNTO CLAVE

Los métodos de la ingeniería de software llevan a reducir la magnitud de los picos y de la pendiente de la curva real en la figura 1.2.

sas elevadas de fallas al comienzo de la vida de un programa. Sin embargo, éstas se corrigen y la curva se aplana, como se indica. La curva idealizada es una gran simplificación de los modelos reales de las fallas del software. Aun así, la implicación está clara: el software no se desgasta, ¡pero sí se *deteriora!*

Esta contradicción aparente se entiende mejor si se considera la curva real en la figura 1.2. Durante su vida,[2] el software sufrirá cambios. Es probable que cuando éstos se realicen, se introduzcan errores que ocasionen que la curva de tasa de fallas tenga aumentos súbitos, como se ilustra en la "curva real" (véase la figura 1.2). Antes de que la curva vuelva a su tasa de fallas original de estado estable, surge la solicitud de otro cambio que hace que la curva se dispare otra vez. Poco a poco, el nivel mínimo de la tasa de fallas comienza a aumentar: el software se está deteriorando como consecuencia del cambio.

Otro aspecto del desgaste ilustra la diferencia entre el hardware y el software. Cuando un componente del hardware se desgasta es sustituido por una refacción. En cambio, no hay refacciones para el software. Cada falla de éste indica un error en el diseño o en el proceso que tradujo el diseño a código ejecutable por la máquina. Entonces, las tareas de mantenimiento del software, que incluyen la satisfacción de peticiones de cambios, involucran una complejidad considerablemente mayor que el mantenimiento del hardware.

Cita:

"Las ideas son los ladrillos con los que se construyen las ideas."

Jason Zebehazy

3. *Aunque la industria se mueve hacia la construcción basada en componentes, la mayor parte del software se construye para un uso individualizado*.

A medida que evoluciona una disciplina de ingeniería, se crea un conjunto de componentes estandarizados para el diseño. Los tornillos estándar y los circuitos integrados preconstruidos son sólo dos de los miles de componentes estándar que utilizan los ingenieros mecánicos y eléctricos conforme diseñan nuevos sistemas. Los componentes reutilizables han sido creados para que el ingeniero pueda concentrarse en los elementos verdaderamente innovadores de un diseño; es decir, en las partes de éste que representan algo nuevo. En el mundo del hardware, volver a usar componentes es una parte

2 En realidad, los distintos participantes solicitan cambios desde el momento en que comienza el desarrollo y mucho antes de que se disponga de la primera versión.

natural del proceso de ingeniería. En el del software, es algo que apenas ha empezado a hacerse a gran escala.

Un componente de software debe diseñarse e implementarse de modo que pueda volverse a usar en muchos programas diferentes. Los modernos componentes reutilizables incorporan tanto los datos como el procesamiento que se les aplica, lo que permite que el ingeniero de software cree nuevas aplicaciones a partir de partes susceptibles de volverse a usar.[3] Por ejemplo, las actuales interfaces interactivas de usuario se construyen con componentes reutilizables que permiten la creación de ventanas gráficas, menús desplegables y una amplia variedad de mecanismos de interacción. Las estructuras de datos y el detalle de procesamiento que se requieren para construir la interfaz están contenidos en una librería de componentes reusables para tal fin.

1.1.2 Dominios de aplicación del software

Actualmente, hay siete grandes categorías de software de computadora que plantean retos continuos a los ingenieros de software:

Software de sistemas: conjunto de programas escritos para dar servicio a otros programas. Determinado software de sistemas (por ejemplo, compiladores, editores y herramientas para administrar archivos) procesa estructuras de información complejas pero deterministas.[4] Otras aplicaciones de sistemas (por ejemplo, componentes de sistemas operativos, manejadores, software de redes, procesadores de telecomunicaciones) procesan sobre todo datos indeterminados. En cualquier caso, el área de software de sistemas se caracteriza por: gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la secuenciación, recursos compartidos y administración de un proceso sofisticado, estructuras complejas de datos e interfaces externas múltiples.

WebRef

En la dirección **shareware.cnet.com** se encuentra una de las librerías más completas de software compartido y libre.

Software de aplicación: programas aislados que resuelven una necesidad específica de negocios. Las aplicaciones en esta área procesan datos comerciales o técnicos en una forma que facilita las operaciones de negocios o la toma de decisiones administrativas o técnicas. Además de las aplicaciones convencionales de procesamiento de datos, el software de aplicación se usa para controlar funciones de negocios en tiempo real (por ejemplo, procesamiento de transacciones en punto de venta, control de procesos de manufactura en tiempo real).

Software de ingeniería y ciencias: se ha caracterizado por algoritmos "devoradores de números". Las aplicaciones van de la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada. Sin embargo, las aplicaciones modernas dentro del área de la ingeniería y las ciencias están abandonando los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas, han comenzado a hacerse en tiempo real e incluso han tomado características del software de sistemas.

Software incrustado: reside dentro de un producto o sistema y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. El software incrustado ejecuta funciones limitadas y particulares (por ejemplo, control del tablero de un horno de microondas) o provee una capacidad significativa de funcionamiento y control

3 El desarrollo basado en componentes se estudia en el capítulo 10.

4 El software es *determinista* si es posible predecir el orden y momento de las entradas, el procesamiento y las salidas. El software es *no determinista* si no pueden predecirse el orden y momento en que ocurren éstos.

(funciones digitales en un automóvil, como el control del combustible, del tablero de control y de los sistemas de frenado).

Software de línea de productos: es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular (por ejemplo, control del inventario de productos) o se dirige a mercados masivos de consumidores (procesamiento de textos, hojas de cálculo, gráficas por computadora, multimedios, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).

Aplicaciones web: llamadas "*webapps*", esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones. En su forma más sencilla, las *webapps* son poco más que un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto y gráficas limitadas. Sin embargo, desde que surgió Web 2.0, las *webapps* están evolucionando hacia ambientes de cómputo sofisticados que no sólo proveen características aisladas, funciones de cómputo y contenido para el usuario final, sino que también están integradas con bases de datos corporativas y aplicaciones de negocios.

Cita:

"No hay computadora que tenga sentido común."

Marvin Minsky

Software de inteligencia artificial: hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área incluyen robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

Son millones de ingenieros de software en todo el mundo los que trabajan duro en proyectos de software en una o más de estas categorías. En ciertos casos se elaboran sistemas nuevos, pero en muchos otros se corrigen, adaptan y mejoran aplicaciones ya existentes. No es raro que una joven ingeniera de software trabaje en un programa de mayor edad que la de ella… Las generaciones pasadas de los trabajadores del software dejaron un legado en cada una de las categorías mencionadas. Por fortuna, la herencia que dejará la actual generación aligerará la carga de los futuros ingenieros de software. Aun así, nuevos desafíos (capítulo 31) han aparecido en el horizonte.

Computación en un mundo abierto: el rápido crecimiento de las redes inalámbricas quizá lleve pronto a la computación verdaderamente ubicua y distribuida. El reto para los ingenieros de software será desarrollar software de sistemas y aplicación que permita a dispositivos móviles, computadoras personales y sistemas empresariales comunicarse a través de redes enormes.

Construcción de redes: la red mundial (World Wide Web) se está convirtiendo con rapidez tanto en un motor de computación como en un proveedor de contenido. El desafío para los ingenieros de software es hacer arquitecturas sencillas (por ejemplo, planeación financiera personal y aplicaciones sofisticadas que proporcionen un beneficio a mercados objetivo de usuarios finales en todo el mundo).

Fuente abierta: tendencia creciente que da como resultado la distribución de código fuente para aplicaciones de sistemas (por ejemplo, sistemas operativos, bases de datos y ambientes de desarrollo) de modo que mucha gente pueda contribuir a su desarrollo. El desafío para los ingenieros de software es elaborar código fuente que sea autodescriptivo, y también, lo que es más importante, desarrollar técnicas que permitirán tanto a los consumidores como a los desarrolladores saber cuáles son los cambios hechos y cómo se manifiestan dentro del software.

Cita:

"No siempre puedes predecir, pero siempre puedes prepararte."

Anónimo

Es indudable que cada uno de estos nuevos retos obedecerá a la ley de las consecuencias imprevistas y tendrá efectos (para hombres de negocios, ingenieros de software y usuarios finales) que hoy no pueden predecirse. Sin embargo, los ingenieros de software pueden prepararse de-

sarrollando un proceso que sea ágil y suficientemente adaptable para que acepte los cambios profundos en la tecnología y las reglas de los negocios que seguramente surgirán en la década siguiente.

1.1.3 Software heredado

Cientos de miles de programas de cómputo caen en uno de los siete dominios amplios de aplicación que se estudiaron en la subsección anterior. Algunos de ellos son software muy nuevo, disponible para ciertos individuos, industria y gobierno. Pero otros programas son más viejos, en ciertos casos *muy* viejos.

Estos programas antiguos —que es frecuente denominar *software heredado*— han sido centro de atención y preocupación continuas desde la década de 1960. Dayani-Fard y sus colegas [Day99] describen el software heredado de la manera siguiente:

> Los sistemas de software heredado [...] fueron desarrollados hace varias décadas y han sido modificados de manera continua para que satisfagan los cambios en los requerimientos de los negocios y plataformas de computación. La proliferación de tales sistemas es causa de dolores de cabeza para las organizaciones grandes, a las que resulta costoso mantenerlos y riesgoso hacerlos evolucionar.

Liu y sus colegas [Liu98] amplían esta descripción al hacer notar que "muchos sistemas heredados continúan siendo un apoyo para las funciones básicas del negocio y son 'indispensables' para éste". Además, el software heredado se caracteriza por su longevidad e importancia crítica para el negocio.

¿Qué hago si encuentro un sistema heredado de mala calidad?

Desafortunadamente, en ocasiones hay otra característica presente en el software heredado: *mala calidad*.[5] Hay veces en las que los sistemas heredados tienen diseños que no son susceptibles de extenderse, código confuso, documentación mala o inexistente, casos y resultados de pruebas que nunca se archivaron, una historia de los cambios mal administrada... la lista es muy larga. A pesar de esto, dichos sistemas dan apoyo a las "funciones básicas del negocio y son indispensables para éste". ¿Qué hacer?

La única respuesta razonable es: *hacer nada*, al menos hasta que el sistema heredado tenga un cambio significativo. Si el software heredado satisface las necesidades de sus usuarios y corre de manera confiable, entonces no falla ni necesita repararse. Sin embargo, conforme pase el tiempo será frecuente que los sistemas de software evolucionen por una o varias de las siguientes razones:

¿Qué tipos de cambios se hacen a los sistemas heredados?

- El software debe adaptarse para que cumpla las necesidades de los nuevos ambientes del cómputo y de la tecnología.
- El software debe ser mejorado para implementar nuevos requerimientos del negocio.
- El software debe ampliarse para que sea operable con otros sistemas o bases de datos modernos.
- La arquitectura del software debe rediseñarse para hacerla viable dentro de un ambiente de redes.

Todo ingeniero de software debe reconocer que el cambio es natural. No trate de evitarlo.

Cuando ocurren estos modos de evolución, debe hacerse la reingeniería del sistema heredado (capítulo 29) para que sea viable en el futuro. La meta de la ingeniería de software moderna es "desarrollar metodologías que se basen en el concepto de evolución; es decir, el concepto de que los sistemas de software cambian continuamente, que los nuevos sistemas de software se

5 En este caso, la calidad se juzga con base en el pensamiento moderno de la ingeniería de software, criterio algo injusto, ya que algunos conceptos y principios de la ingeniería de software moderna tal vez no hayan sido bien entendidos en la época en que se desarrolló el software heredado.

desarrollan a partir de los antiguos y [...] que todo debe operar entre sí y cooperar con cada uno de los demás" [Day99].

1.2 LA NATURALEZA ÚNICA DE LAS *WEBAPPS*

Cita:

"Cuando veamos cualquier tipo de estabilización, la web se habrá convertido en algo completamente diferente."

Louis Monier

En los primeros días de la Red Mundial (entre 1990 y 1995), los *sitios web* consistían en poco más que un conjunto de archivos de hipertexto vinculados que presentaban la información con el empleo de texto y gráficas limitadas. Al pasar el tiempo, el aumento de HTML por medio de herramientas de desarrollo (XML, Java) permitió a los ingenieros de la web brindar capacidad de cómputo junto con contenido de información. Habían nacido los *sistemas y aplicaciones basados en la web*[6] (denominó a éstas en forma colectiva como *webapps*). En la actualidad, las *webapps* se han convertido en herramientas sofisticadas de cómputo que no sólo proporcionan funciones aisladas al usuario final, sino que también se han integrado con bases de datos corporativas y aplicaciones de negocios.

Como se dijo en la sección 1.1.2, las *webapps* son una de varias categorías distintas de software. No obstante, podría argumentarse que las *webapps* son diferentes. Powell [Pow98] sugiere que los sistemas y aplicaciones basados en web "involucran una mezcla entre las publicaciones impresas y el desarrollo de software, entre la mercadotecnia y la computación, entre las comunicaciones internas y las relaciones exteriores, y entre el arte y la tecnología". La gran mayoría de *webapps* presenta los siguientes atributos:

¿Qué característica diferencia las *webapps* de otro software?

Uso intensivo de redes. Una *webapp* reside en una red y debe atender las necesidades de una comunidad diversa de clientes. La red permite acceso y comunicación mundiales (por ejemplo, internet) o tiene acceso y comunicación limitados (por ejemplo, una intranet corporativa).

Concurrencia. A la *webapp* puede acceder un gran número de usuarios a la vez. En muchos casos, los patrones de uso entre los usuarios finales varían mucho.

Carga impredecible. El número de usuarios de la *webapp* cambia en varios órdenes de magnitud de un día a otro. El lunes tal vez la utilicen cien personas, el jueves quizá 10 000 usen el sistema.

Rendimiento. Si un usuario de la *webapp* debe esperar demasiado (para entrar, para el procesamiento por parte del servidor, para el formado y despliegue del lado del cliente), él o ella quizá decidan irse a otra parte.

Disponibilidad. Aunque no es razonable esperar una disponibilidad de 100%, es frecuente que los usuarios de *webapps* populares demanden acceso las 24 horas de los 365 días del año. Los usuarios en Australia o Asia quizá demanden acceso en horas en las que las aplicaciones internas de software tradicionales en Norteamérica no estén en línea por razones de mantenimiento.

Orientadas a los datos. La función principal de muchas *webapp* es el uso de hipermedios para presentar al usuario final contenido en forma de texto, gráficas, audio y video. Además, las *webapps* se utilizan en forma común para acceder a información que existe en bases de datos que no son parte integral del ambiente basado en web (por ejemplo, comercio electrónico o aplicaciones financieras).

---

6 En el contexto de este libro, el término *aplicación web* (*webapp*) agrupa todo, desde una simple página web que ayude al consumidor a calcular el pago del arrendamiento de un automóvil hasta un sitio web integral que proporcione servicios completos de viaje para gente de negocios y vacacionistas. En esta categoría se incluyen sitios web completos, funcionalidad especializada dentro de sitios web y aplicaciones de procesamiento de información que residen en internet o en una intranet o extranet.

Contenido sensible. La calidad y naturaleza estética del contenido constituye un rasgo importante de la calidad de una *webapp*.

Evolución continua. A diferencia del software de aplicación convencional que evoluciona a lo largo de una serie de etapas planeadas y separadas cronológicamente, las aplicaciones web evolucionan en forma continua. No es raro que ciertas *webapp* (específicamente su contenido) se actualicen minuto a minuto o que su contenido se calcule en cada solicitud.

Inmediatez. Aunque la *inmediatez* —necesidad apremiante de que el software llegue con rapidez al mercado— es una característica en muchos dominios de aplicación, es frecuente que las *webapps* tengan plazos de algunos días o semanas para llegar al mercado.[7]

Seguridad. Debido a que las *webapps* se encuentran disponibles con el acceso a una red, es difícil o imposible limitar la población de usuarios finales que pueden acceder a la aplicación. Con el fin de proteger el contenido sensible y brindar modos seguros de transmisión de los datos, deben implementarse medidas estrictas de seguridad a través de la infraestructura de apoyo de una *webapp* y dentro de la aplicación misma.

Estética. Parte innegable del atractivo de una *webapp* es su apariencia y percepción. Cuando se ha diseñado una aplicación para comercializar o vender productos o ideas, la estética tiene tanto que ver con el éxito como el diseño técnico.

Podría argumentarse que otras categorías de aplicaciones estudiadas en la sección 1.1.2 muestran algunos de los atributos mencionados. Sin embargo, las *webapps* casi siempre poseen todos ellos.

1.3 INGENIERÍA DE SOFTWARE

Con objeto de elaborar software listo para enfrentar los retos del siglo XXI, el lector debe aceptar algunas realidades sencillas:

PUNTO CLAVE
Entender el problema antes de dar una solución.

- El software se ha incrustado profundamente en casi todos los aspectos de nuestras vidas y, como consecuencia, el número de personas que tienen interés en las características y funciones que brinda una aplicación específica[8] ha crecido en forma notable. Cuando ha de construirse una aplicación nueva o sistema incrustado, deben escucharse muchas opiniones. Y en ocasiones parece que cada una de ellas tiene una idea un poco distinta de cuáles características y funciones debiera tener el software. *Se concluye que debe hacerse un esfuerzo concertado para entender el problema antes de desarrollar una aplicación de software.*

PUNTO CLAVE
El diseño es una actividad crucial de la ingeniería de software.

- Los requerimientos de la tecnología de la información que demandan los individuos, negocios y gobiernos se hacen más complejos con cada año que pasa. En la actualidad, grandes equipos de personas crean programas de cómputo que antes eran elaborados por un solo individuo. El software sofisticado, que alguna vez se implementó en un ambiente de cómputo predecible y autocontenido, hoy en día se halla incrustado en el interior de todo, desde la electrónica de consumo hasta dispositivos médicos o sistemas de armamento. La complejidad de estos nuevos sistemas y productos basados en computadora demanda atención cuidadosa a las interacciones de todos los elementos del sistema. *Se concluye que el diseño se ha vuelto una actividad crucial.*

7 Con las herramientas modernas es posible producir páginas web sofisticadas en unas cuantas horas.
8 En una parte posterior de este libro, llamaré a estas personas "participantes".

PUNTO CLAVE

Tanto la calidad como la facilidad de recibir mantenimiento son resultado de un buen diseño.

- Los individuos, negocios y gobiernos dependen cada vez más del software para tomar decisiones estratégicas y tácticas, así como para sus operaciones y control cotidianos. Si el software falla, las personas y empresas grandes pueden experimentar desde un inconveniente menor hasta fallas catastróficas. *Se concluye que el software debe tener alta calidad.*
- A medida que aumenta el valor percibido de una aplicación específica se incrementa la probabilidad de que su base de usuarios y longevidad también crezcan. Conforme se extienda su base de usuarios y el tiempo de uso, las demandas para adaptarla y mejorarla también crecerán. *Se concluye que el software debe tener facilidad para recibir mantenimiento.*

Estas realidades simples llevan a una conclusión: *debe hacerse ingeniería con el software en todas sus formas y a través de todos sus dominios de aplicación.* Y esto conduce al tema de este libro: *la ingeniería de software*.

Cita:

"Más que una disciplina o cuerpo de conocimientos, ingeniería es un verbo, una palabra de acción, una forma de abordar un problema."

Scott Whitmir

Aunque cientos de autores han desarrollado definiciones personales de la ingeniería de software, la propuesta por Fritz Bauer [Nau69] en la conferencia fundamental sobre el tema todavía sirve como base para el análisis:

> [La ingeniería de software es] el establecimiento y uso de principios fundamentales de la ingeniería con objeto de desarrollar en forma económica software que sea confiable y que trabaje con eficiencia en máquinas reales.

El lector se sentirá tentado de ampliar esta definición.[9] Dice poco sobre los aspectos técnicos de la calidad del software; no habla directamente de la necesidad de satisfacer a los consumidores ni de entregar el producto a tiempo; omite mencionar la importancia de la medición y la metrología; no establece la importancia de un proceso eficaz. No obstante, la definición de Bauer proporciona una base. ¿Cuáles son los "principios fundamentales de la ingeniería" que pueden aplicarse al desarrollo del software de computadora? ¿Cómo se desarrolla software "en forma económica" y que sea "confiable"? ¿Qué se requiere para crear programas de cómputo que trabajen con "eficiencia", no en una sino en muchas "máquinas reales" diferentes? Éstas son las preguntas que siguen siendo un reto para los ingenieros de software.

? ¿Cómo se define la ingeniería de software?

El IEEE [IEEE93a] ha desarrollado una definición más completa, como sigue:

> La ingeniería de software es: 1) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software. 2) El estudio de enfoques según el punto 1.

Aun así, el enfoque "sistemático, disciplinado y cuantificable" aplicado por un equipo de software podría ser algo burdo para otro. Se necesita disciplina, pero también adaptabilidad y agilidad.

PUNTO CLAVE

La ingeniería de software incluye un proceso, métodos y herramientas para administrar y hacer ingeniería con el software.

La ingeniería de software es una tecnología con varias capas. Como se aprecia en la figura 1.3, cualquier enfoque de ingeniería (incluso la de software) debe basarse en un compromiso organizacional con la calidad. La administración total de la calidad, Six Sigma y otras filosofías similares[10] alimentan la cultura de mejora continua, y es esta cultura la que lleva en última instancia al desarrollo de enfoques cada vez más eficaces de la ingeniería de software. El fundamento en el que se apoya la ingeniería de software es el compromiso con la calidad.

El fundamento para la ingeniería de software es la capa *proceso*. El proceso de ingeniería de software es el aglutinante que une las capas de la tecnología y permite el desarrollo racional y

9 Consulte muchas otras definiciones en www.answers.com/topic/software-engineering#wp-_note-13.

10 En el capítulo 14 y toda la parte 3 del libro se estudia la administración de la calidad y los enfoques relacionados con ésta.

FIGURA 1.3
Capas de la ingeniería de software

WebRef
CrossTalk es un periódico que da información práctica sobre procesos, métodos y herramientas. Se encuentra en **www.stsc.hill.af.mil**

oportuno del software de cómputo. El proceso define una estructura que debe establecerse para la obtención eficaz de tecnología de ingeniería de software. El proceso de software forma la base para el control de la administración de proyectos de software, y establece el contexto en el que se aplican métodos técnicos, se generan productos del trabajo (modelos, documentos, datos, reportes, formatos, etc.), se establecen puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada.

Los *métodos* de la ingeniería de software proporcionan la experiencia técnica para elaborar software. Incluyen un conjunto amplio de tareas, como comunicación, análisis de los requerimientos, modelación del diseño, construcción del programa, pruebas y apoyo. Los métodos de la ingeniería de software se basan en un conjunto de principios fundamentales que gobiernan cada área de la tecnología e incluyen actividades de modelación y otras técnicas descriptivas.

Las *herramientas* de la ingeniería de software proporcionan un apoyo automatizado o semiautomatizado para el proceso y los métodos. Cuando se integran las herramientas de modo que la información creada por una pueda ser utilizada por otra, queda establecido un sistema llamado *ingeniería de software asistido por computadora* que apoya el desarrollo de software.

1.4 EL PROCESO DEL SOFTWARE

? ¿Cuáles son los elementos de un proceso de software?

Cita:
"Un proceso define quién hace *qué*, *cuándo* y *cómo*, para alcanzar cierto objetivo."
Ivar Jacobson, Grady Booch y James Rumbaugh

Un *proceso* es un conjunto de actividades, acciones y tareas que se ejecutan cuando va a crearse algún producto del trabajo. Una *actividad* busca lograr un objetivo amplio (por ejemplo, comunicación con los participantes) y se desarrolla sin importar el dominio de la aplicación, tamaño del proyecto, complejidad del esfuerzo o grado de rigor con el que se usará la ingeniería de software. Una *acción* (diseño de la arquitectura) es un conjunto de tareas que producen un producto importante del trabajo (por ejemplo, un modelo del diseño de la arquitectura). Una *tarea* se centra en un objetivo pequeño pero bien definido (por ejemplo, realizar una prueba unitaria) que produce un resultado tangible.

En el contexto de la ingeniería de software, un proceso *no* es una prescripción rígida de cómo elaborar software de cómputo. Por el contrario, es un enfoque adaptable que permite que las personas que hacen el trabajo (el equipo de software) busquen y elijan el conjunto apropiado de acciones y tareas para el trabajo. Se busca siempre entregar el software en forma oportuna y con calidad suficiente para satisfacer a quienes patrocinaron su creación y a aquellos que lo usarán.

La *estructura del proceso* establece el fundamento para el proceso completo de la ingeniería de software por medio de la identificación de un número pequeño de *actividades estructurales* que sean aplicables a todos los proyectos de software, sin importar su tamaño o complejidad. Además, la estructura del proceso incluye un conjunto de *actividades sombrilla* que son aplicables a través de todo el proceso del software. Una estructura de proceso general para la ingeniería de software consta de cinco actividades:

¿Cuáles son las cinco actividades estructurales del proceso?

Comunicación. Antes de que comience cualquier trabajo técnico, tiene importancia crítica comunicarse y colaborar con el cliente (y con otros participantes).[11] Se busca entender los objetivos de los participantes respecto del proyecto, y reunir los requerimientos que ayuden a definir las características y funciones del software.

Planeación. Cualquier viaje complicado se simplifica si existe un mapa. Un proyecto de software es un viaje difícil, y la actividad de planeación crea un "mapa" que guía al equipo mientras viaja. El mapa —llamado *plan del proyecto de software*— define el trabajo de ingeniería de software al describir las tareas técnicas por realizar, los riesgos probables, los recursos que se requieren, los productos del trabajo que se obtendrán y una programación de las actividades.

Cita:

"Einstein afirmaba que debía haber una explicación sencilla de la naturaleza porque Dios no es caprichoso o arbitrario. Al ingeniero de software no lo conforta una fe parecida. Gran parte de la complejidad que debe doblegar es de origen arbitrario."

Fred Brooks

Modelado. Ya sea usted diseñador de paisaje, constructor de puentes, ingeniero aeronáutico, carpintero o arquitecto, a diario trabaja con modelos. Crea un "bosquejo" del objeto por hacer a fin de entender el panorama general —cómo se verá arquitectónicamente, cómo ajustan entre sí las partes constituyentes y muchas características más—. Si se requiere, refina el bosquejo con más y más detalles en un esfuerzo por comprender mejor el problema y cómo resolverlo. Un ingeniero de software hace lo mismo al crear modelos a fin de entender mejor los requerimientos del software y el diseño que los satisfará.

Construcción. Esta actividad combina la generación de código (ya sea manual o automatizada) y las pruebas que se requieren para descubrir errores en éste.

Despliegue. El software (como entidad completa o como un incremento parcialmente terminado) se entrega al consumidor que lo evalúa y que le da retroalimentación, misma que se basa en dicha evaluación.

Estas cinco actividades estructurales genéricas se usan durante el desarrollo de programas pequeños y sencillos, en la creación de aplicaciones web grandes y en la ingeniería de sistemas enormes y complejos basados en computadoras. Los detalles del proceso de software serán distintos en cada caso, pero las actividades estructurales son las mismas.

Para muchos proyectos de software, las actividades estructurales se aplican en forma iterativa a medida que avanza el proyecto. Es decir, la **comunicación,** la **planeación,** el **modelado,** la **construcción** y el **despliegue** se ejecutan a través de cierto número de repeticiones del proyecto. Cada iteración produce un *incremento del software* que da a los participantes un subconjunto de características y funcionalidad generales del software. Conforme se produce cada incremento, el software se hace más y más completo.

Las actividades estructurales del proceso de ingeniería de software son complementadas por cierto número de *actividades sombrilla*. En general, las actividades sombrilla se aplican a lo largo de un proyecto de software y ayudan al equipo que lo lleva a cabo a administrar y controlar el avance, la calidad, el cambio y el riesgo. Es común que las actividades sombrilla sean las siguientes:

PUNTO CLAVE

Las actividades sombrilla ocurren a lo largo del proceso de software y se centran sobre todo en la administración, el seguimiento y el control del proyecto.

Seguimiento y control del proyecto de software: permite que el equipo de software evalúe el progreso comparándolo con el plan del proyecto y tome cualquier acción necesaria para apegarse a la programación de actividades.

Administración del riesgo: evalúa los riesgos que puedan afectar el resultado del proyecto o la calidad del producto.

11 Un *participante* es cualquier persona que tenga algo que ver en el resultado exitoso del proyecto —gerentes del negocio, usuarios finales, ingenieros de software, personal de apoyo, etc.—. Rob Thomset dice en broma que "un participante es una persona que blande una estaca grande y aguda [...] Si no vez más lejos que los participantes, ya sabes dónde terminará la estaca". (N. del T.: Esta nota es un juego de palabras: *stake* significa estaca y también *parte*, y *stakeholder* es el que blande una estaca, pero también un *participante*.)

Aseguramiento de la calidad del software: define y ejecuta las actividades requeridas para garantizar la calidad del software.

Revisiones técnicas: evalúa los productos del trabajo de la ingeniería de software a fin de descubrir y eliminar errores antes de que se propaguen a la siguiente actividad.

Medición: define y reúne mediciones del proceso, proyecto y producto para ayudar al equipo a entregar el software que satisfaga las necesidades de los participantes; puede usarse junto con todas las demás actividades estructurales y sombrilla.

Administración de la configuración del software: administra los efectos del cambio a lo largo del proceso del software.

Administración de la reutilización: define criterios para volver a usar el producto del trabajo (incluso los componentes del software) y establece mecanismos para obtener componentes reutilizables.

Preparación y producción del producto del trabajo: agrupa las actividades requeridas para crear productos del trabajo, tales como modelos, documentos, registros, formatos y listas.

Cada una de estas actividades sombrilla se analiza en detalle más adelante.

PUNTO CLAVE
La adaptación del proceso de software es esencial para el éxito del proyecto.

Ya se dijo en esta sección que el proceso de ingeniería de software no es una prescripción rígida que deba seguir en forma dogmática el equipo que lo crea. Al contrario, debe ser ágil y adaptable (al problema, al proyecto, al equipo y a la cultura organizacional). Por tanto, un proceso adoptado para un proyecto puede ser significativamente distinto de otro adoptado para otro proyecto. Entre las diferencias se encuentran las siguientes:

? ¿Qué diferencias existen entre los modelos del proceso?

- Flujo general de las actividades, acciones y tareas, así como de las interdependencias entre ellas
- Grado en el que las acciones y tareas están definidas dentro de cada actividad estructural
- Grado en el que se identifican y requieren los productos del trabajo
- Forma en la que se aplican las actividades de aseguramiento de la calidad
- Manera en la que se realizan las actividades de seguimiento y control del proyecto
- Grado general de detalle y rigor con el que se describe el proceso
- Grado con el que el cliente y otros participantes se involucran con el proyecto
- Nivel de autonomía que se da al equipo de software
- Grado con el que son prescritos la organización y los roles del equipo

Cita:

"Siento que una receta es sólo un tema que una cocinera inteligente ejecuta con una variación en cada ocasión."

Madame Benoit

En la parte 1 de este libro, se examinará el proceso de software con mucho detalle. Los *modelos de proceso prescriptivo* (capítulo 2) enfatizan la definición, la identificación y la aplicación detalladas de las actividades y tareas del proceso. Su objetivo es mejorar la calidad del sistema, desarrollar proyectos más manejables, hacer más predecibles las fechas de entrega y los costos, y guiar a los equipos de ingenieros de software cuando realizan el trabajo que se requiere para construir un sistema. Desafortunadamente, ha habido casos en los que estos objetivos no se han logrado. Si los modelos prescriptivos se aplican en forma dogmática y sin adaptación, pueden incrementar el nivel de burocracia asociada con el desarrollo de sistemas basados en computadora y crear inadvertidamente dificultades para todos los participantes.

? ¿Qué caracteriza a un proceso "ágil"?

Los *modelos de proceso ágil* (capítulo 3) ponen el énfasis en la "agilidad" del proyecto y siguen un conjunto de principios que conducen a un enfoque más informal (pero no menos efectivo, dicen sus defensores) del proceso de software. Por lo general, se dice que estos modelos del proceso son "ágiles" porque acentúan la maniobrabilidad y la adaptabilidad. Son apropiados para muchos tipos de proyectos y son útiles en particular cuando se hace ingeniería sobre aplicaciones web.

1.5 LA PRÁCTICA DE LA INGENIERÍA DE SOFTWARE

WebRef

En la dirección **www.literateprogramming.com** se encuentran varias citas provocativas sobre la práctica de la ingeniería de software.

En la sección 1.4 se introdujo un modelo general de proceso de software compuesto de un conjunto de actividades que establecen una estructura para la práctica de la ingeniería de software. Las actividades estructurales generales —**comunicación, planeación, modelado, construcción** y **despliegue**— y las actividades sombrilla establecen el esqueleto de la arquitectura para el trabajo de ingeniería de software. Pero, ¿cómo entra aquí la práctica de la ingeniería de software? En las secciones que siguen, el lector obtendrá la comprensión básica de los conceptos y principios generales que se aplican a las actividades estructurales.[12]

1.5.1 La esencia de la práctica

CONSEJO

Podría decirse que el enfoque de Polya es simple sentido común. Es verdad. Pero es sorprendente la frecuencia con la que el sentido común es poco común en el mundo del software.

En un libro clásico, *How to Solve It*, escrito antes de que existieran las computadoras modernas, George Polya [Pol45] describió la esencia de la solución de problemas y, en consecuencia, la esencia de la práctica de la ingeniería de software:

1. *Entender el problema* (comunicación y análisis).
2. *Planear la solución* (modelado y diseño del software).
3. *Ejecutar el plan* (generación del código).
4. *Examinar la exactitud del resultado* (probar y asegurar la calidad).

En el contexto de la ingeniería de software, estas etapas de sentido común conducen a una serie de preguntas esenciales [adaptado de Pol45]:

Entender el problema. En ocasiones es difícil de admitir, pero la mayor parte de nosotros adoptamos una actitud de orgullo desmedido cuando se nos presenta un problema. Escuchamos por unos segundos y después pensamos: *Claro, sí, entiendo, resolvamos esto.* Desafortunadamente, entender no siempre es fácil. Es conveniente dedicar un poco de tiempo a responder algunas preguntas sencillas:

- *¿Quiénes tienen que ver con la solución del problema?* Es decir, ¿quiénes son los participantes?
- *¿Cuáles son las incógnitas?* ¿Cuáles datos, funciones y características se requieren para resolver el problema en forma apropiada?
- *¿Puede fraccionarse el problema?* ¿Es posible representarlo con problemas más pequeños que sean más fáciles de entender?
- *¿Es posible representar gráficamente el problema?* ¿Puede crearse un modelo de análisis?

Cita:

"En la solución de cualquier problema hay un grano de descubrimiento."

George Polya

Planear la solución. Ahora entiende el problema (o es lo que piensa) y no puede esperar para escribir el código. Antes de hacerlo, cálmese un poco y haga un pequeño diseño:

- *¿Ha visto antes problemas similares?* ¿Hay patrones reconocibles en una solución potencial? ¿Hay algún software existente que implemente los datos, funciones y características que se requieren?
- *¿Ha resuelto un problema similar?* Si es así, ¿son reutilizables los elementos de la solución?
- *¿Pueden definirse problemas más pequeños?* Si así fuera, ¿hay soluciones evidentes para éstos?

12 El lector debería volver a consultar las secciones de este capítulo a medida que en el libro se describan en específico los métodos y las actividades sombrilla de la ingeniería de software.

- *¿Es capaz de representar una solución en una forma que lleve a su implementación eficaz?* ¿Es posible crear un modelo del diseño?

Ejecutar el plan. El diseño que creó sirve como un mapa de carreteras para el sistema que quiere construir. Puede haber desviaciones inesperadas y es posible que descubra un camino mejor a medida que avanza, pero el "plan" le permitirá proceder sin que se pierda.

- *¿Se ajusta la solución al plan?* ¿El código fuente puede apegarse al modelo del diseño?
- *¿Es probable que cada parte componente de la solución sea correcta?* ¿El diseño y código se han revisado o, mejor aún, se han hecho pruebas respecto de la corrección del algoritmo?

Examinar el resultado. No se puede estar seguro de que la solución sea perfecta, pero sí de que se ha diseñado un número suficiente de pruebas para descubrir tantos errores como sea posible.

- *¿Puede probarse cada parte componente de la solución?* ¿Se ha implementado una estrategia razonable para hacer pruebas?
- *¿La solución produce resultados que se apegan a los datos, funciones y características que se requieren?* ¿El software se ha validado contra todos los requerimientos de los participantes?

No debiera sorprender que gran parte de este enfoque tenga que ver con el sentido común. En realidad, es razonable afirmar que un enfoque de sentido común para la ingeniería de software hará que nunca se extravíe.

1.5.2 Principios generales

El diccionario define la palabra *principio* como "una ley importante o suposición que subyace y se requiere en un sistema de pensamiento". En este libro se analizarán principios en muchos niveles distintos de abstracción. Algunos se centran en la ingeniería de software como un todo, otros consideran una actividad estructural general específica (por ejemplo, **comunicación**), y otros más se centran en acciones de la ingeniería de software (por ejemplo, diseño de la arquitectura) o en tareas técnicas (escribir un escenario para el uso). Sin importar su nivel de enfoque, los principios lo ayudarán a establecer un conjunto de herramientas mentales para una práctica sólida de la ingeniería de software. Ésa es la razón de que sean importantes.

David Hooker [Hoo96] propuso siete principios que se centran en la práctica de la ingeniería de software como un todo. Se reproducen en los párrafos siguientes:[13]

Antes de comenzar un proyecto de software, asegúrese de que el software tenga un propósito para el negocio y que los usuarios perciben valor en él.

Primer principio: *La razón de que exista todo*

Un sistema de software existe por una razón: *dar valor a sus usuarios.* Todas las decisiones deben tomarse teniendo esto en mente. Antes de especificar un requerimiento del sistema, antes de notar la funcionalidad de una parte de él, antes de determinar las plataformas del hardware o desarrollar procesos, plantéese preguntas tales como: "¿Esto agrega valor real al sistema?" Si la respuesta es "no", entonces no lo haga. Todos los demás principios apoyan a éste.

Segundo principio: *MSE (Mantenlo sencillo, estúpido...)*

El diseño de software no es un proceso caprichoso. Hay muchos factores por considerar en cualquier actividad de diseño. *Todo diseño debe ser tan simple como sea posible, pero no más.*

13 Reproducido con permiso del autor [Hoo96]. Hooker define algunos patrones para estos principios en http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

Cita:

"Hay cierta majestad en la sencillez, que es con mucho todo lo que adorna al ingenio."

Papa Alejandro (1688-1744)

Esto facilita conseguir un sistema que sea comprendido más fácilmente y que sea susceptible de recibir mantenimiento, lo que no quiere decir que en nombre de la simplicidad deban descartarse características o hasta rasgos internos. En realidad, los diseños más elegantes por lo general son los más simples. Simple tampoco significa "rápido y sucio". La verdad es que con frecuencia se requiere mucha reflexión y trabajo con iteraciones múltiples para poder simplificar. La recompensa es un software más fácil de mantener y menos propenso al error.

Tercer principio: *Mantener la visión*

Una *visión clara es esencial para el éxito de un proyecto de software*. Sin ella, casi infaliblemente el proyecto terminará siendo un ser "con dos [o más mentes]". Sin integridad conceptual, un sistema está amenazado de convertirse en una urdimbre de diseños incompatibles unidos por tornillos del tipo equivocado [...] Comprometer la visión de la arquitectura de un sistema de software debilita y, finalmente hará que colapsen incluso los sistemas bien diseñados. Tener un arquitecto que pueda para mantener la visión y que obligue a su cumplimiento garantiza un proyecto de software muy exitoso.

Cuarto principio: *Otros consumirán lo que usted produce*

PUNTO CLAVE

Si el software tiene valor, cambiará durante su vida útil. Por esa razón, debe construirse de forma que sea fácil darle mantenimiento.

Rara vez se construye en el vacío un sistema de software con fortaleza industrial. En un modo u otro, alguien más lo usará, mantendrá, documentará o, de alguna forma, dependerá de su capacidad para entender el sistema. Así que *siempre establezca especificaciones, diseñe e implemente con la seguridad de que alguien más tendrá que entender lo que usted haga*. La audiencia para cualquier producto de desarrollo de software es potencialmente grande. Elabore especificaciones con la mirada puesta en los usuarios. Diseñe con los implementadores en mente. Codifique pensando en aquellos que deben dar mantenimiento y ampliar el sistema. Alguien debe depurar el código que usted escriba, y eso lo hace usuario de su código. Hacer su trabajo más fácil agrega valor al sistema.

Quinto principio: *Ábrase al futuro*

Un sistema con larga vida útil tiene más valor. En los ambientes de cómputo actuales, donde las especificaciones cambian de un momento a otro y las plataformas de hardware quedan obsoletas con sólo unos meses de edad, es común que la vida útil del software se mida en meses y no en años. Sin embargo, los sistemas de software con verdadera "fortaleza industrial" deben durar mucho más tiempo. Para tener éxito en esto, los sistemas deben ser fáciles de adaptar a ésos y otros cambios. Los sistemas que lo logran son los que se diseñaron para ello desde el principio. *Nunca diseñe sobre algo iniciado.* Siempre pregunte: "¿qué pasa si...?" y prepárese para todas las respuestas posibles mediante la creación de sistemas que resuelvan el problema general, no sólo uno específico.[14] Es muy posible que esto lleve a volver a usar un sistema completo.

Sexto principio: *Planee por anticipado la reutilización*

La reutilización ahorra tiempo y esfuerzo.[15] Al desarrollar un sistema de software, lograr un alto nivel de reutilización es quizá la meta más difícil de lograr. La reutilización del código y de los diseños se ha reconocido como uno de los mayores beneficios de usar tecnologías orientadas a objetos. Sin embargo, la recuperación de esta inversión no es automática. Para reforzar las posibilidades de la reutilización que da la programación orientada a objetos [o la

14 Es peligroso llevar este consejo a los extremos. Diseñar para resolver "el problema general" en ocasiones requiere compromisos de rendimiento y puede volver ineficientes las soluciones específicas.

15 Aunque esto es verdad para aquellos que reutilizan software en proyectos futuros, volver a usar puede ser caro para quienes deben diseñar y elaborar componentes reutilizables. Los estudios indican que diseñar y construir componentes reutilizables llega a costar entre 25 y 200% más que el software buscado. En ciertos casos no se justifica la diferencia de costos.

convencional], se requiere reflexión y planeación. Hay muchas técnicas para incluir la reutilización en cada nivel del proceso de desarrollo del sistema... *La planeación anticipada en busca de la reutilización disminuye el costo e incrementa el valor tanto de los componentes reutilizables como de los sistemas en los que se incorpora.*

Séptimo principio: *¡Piense!*

Este último principio es tal vez el que más se pasa por alto. *Pensar en todo con claridad antes de emprender la acción casi siempre produce mejores resultados*. Cuando se piensa en algo es más probable que se haga bien. Asimismo, también se gana conocimiento al pensar cómo volver a hacerlo bien. Si usted piensa en algo y aun así lo hace mal, eso se convierte en una experiencia valiosa. Un efecto colateral de pensar es aprender a reconocer cuando no se sabe algo, punto en el que se puede investigar la respuesta. Cuando en un sistema se han puesto pensamientos claros, el valor se manifiesta. La aplicación de los primeros seis principios requiere pensar con intensidad, por lo que las recompensas potenciales son enormes.

Si todo ingeniero y equipo de software tan sólo siguiera los siete principios de Hooker, se eliminarían muchas de las dificultades que se experimentan al construir sistemas complejos basados en computadora.

1.6 MITOS DEL SOFTWARE

Cita:

"En ausencia de estándares significativos, una industria nueva como la del software depende sólo del folklore."

Tom DeMarco

Los mitos del software —creencias erróneas sobre éste y sobre el proceso que se utiliza para obtenerlo— se remontan a los primeros días de la computación. Los mitos tienen cierto número de atributos que los hacen insidiosos. Por ejemplo, parecen enunciados razonables de hechos (a veces contienen elementos de verdad), tienen una sensación intuitiva y es frecuente que los manifiesten profesionales experimentados que "conocen la historia".

En la actualidad, la mayoría de profesionales de la ingeniería de software reconocen los mitos como lo que son: actitudes equivocadas que han ocasionado serios problemas a los administradores y a los trabajadores por igual. Sin embargo, las actitudes y hábitos antiguos son difíciles de modificar, y persisten algunos remanentes de los mitos del software.

WebRef

La Software Project Managers Network (Red de Gerentes de Proyectos de Software), en **www.spmn.com**, lo ayuda a eliminar éstos y otros mitos.

Mitos de la administración. Los gerentes que tienen responsabilidades en el software, como los de otras disciplinas, con frecuencia se hallan bajo presión para cumplir el presupuesto, mantener la programación de actividades sin desvíos y mejorar la calidad. Así como la persona que se ahoga se agarra de un clavo ardiente, no es raro que un gerente de software sostenga la creencia en un mito del software si eso disminuye la presión a que está sujeto (incluso de manera temporal).

Mito: *Tenemos un libro lleno de estándares y procedimientos para elaborar software. ¿No le dará a mi personal todo lo que necesita saber?*

Realidad: Tal vez exista el libro de estándares, pero ¿se utiliza? ¿Saben de su existencia los trabajadores del software? ¿Refleja la práctica moderna de la ingeniería de software? ¿Es completo? ¿Es adaptable? ¿Está dirigido a mejorar la entrega a tiempo y también se centra en la calidad? En muchos casos, la respuesta a todas estas preguntas es "no".

Mito: *Si nos atrasamos, podemos agregar más programadores y ponernos al corriente (en ocasiones, a esto se le llama "concepto de la horda de mongoles").*

Realidad: El desarrollo del software no es un proceso mecánico similar a la manufactura. En palabras de Brooks [Bro95]: "agregar personal a un proyecto de software atrasado lo atrasará más". Al principio, esta afirmación parece ir contra la intuición. Sin embargo, a medida que se agregan personas, las que ya se

encontraban trabajando deben dedicar tiempo para enseñar a los recién llegados, lo que disminuye la cantidad de tiempo dedicada al esfuerzo de desarrollo productivo. Pueden agregarse individuos, pero sólo en forma planeada y bien coordinada.

Mito: *Si decido subcontratar el proyecto de software a un tercero, puedo descansar y dejar que esa compañía lo elabore.*

Realidad: Si una organización no comprende cómo administrar y controlar proyectos de software internamente, de manera invariable tendrá dificultades cuando subcontrate proyectos de software.

Mitos del cliente. El cliente que requiere software de computadora puede ser la persona en el escritorio de al lado, un grupo técnico en el piso inferior, el departamento de mercadotecnia y ventas, o una compañía externa que solicita software mediante un contrato. En muchos casos, el cliente sostiene mitos sobre el software porque los gerentes y profesionales de éste hacen poco para corregir la mala información. Los mitos generan falsas expectativas (por parte del cliente) y, en última instancia, la insatisfacción con el desarrollador.

Trabaje muy duro para entender qué es lo que tiene que hacer antes de empezar. Quizás no pueda desarrollarlo a detalle, pero entre más sepa, menor será el riesgo que tome.

Mito: *Para comenzar a escribir programas, es suficiente el enunciado general de los objetivos —podremos entrar en detalles más adelante.*

Realidad: Aunque no siempre es posible tener el enunciado exhaustivo y estable de los requerimientos, un "planteamiento de objetivos" ambiguo es una receta para el desastre. Los requerimientos que no son ambiguos (que por lo general se obtienen en forma iterativa) se desarrollan sólo por medio de una comunicación eficaz y continua entre el cliente y el desarrollador.

Mito: *Los requerimientos del software cambian continuamente, pero el cambio se asimila con facilidad debido a que el software es flexible.*

Realidad: Es verdad que los requerimientos del software cambian, pero el efecto que los cambios tienen varía según la época en la que se introducen. Cuando se solicitan al principio cambios en los requerimientos (antes de que haya comenzado el diseño o la elaboración de código), el efecto sobre el costo es relativamente pequeño.[16] Sin embargo, conforme pasa el tiempo, el costo aumenta con rapidez: los recursos ya se han comprometido, se ha establecido la estructura del diseño y el cambio ocasiona perturbaciones que exigen recursos adicionales y modificaciones importantes del diseño.

Siempre que piense que no hay tiempo para la ingeniería de software, pregúntese: "¿tendremos tiempo de hacerlo otra vez?".

Mitos del profesional. Los mitos que aún sostienen los trabajadores del software han sido alimentados por más de 50 años de cultura de programación. Durante los primeros días, la programación se veía como una forma del arte. Es difícil que mueran los hábitos y actitudes arraigados.

Mito: *Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.*

Realidad: Alguien dijo alguna vez que "entre más pronto se comience a 'escribir el código', más tiempo tomará hacer que funcione". Los datos de la industria indican que entre 60 y 80% de todo el esfuerzo dedicado al software ocurrirá después de entregarlo al cliente por primera vez.

Mito: *Hasta que no se haga "correr" el programa, no hay manera de evaluar su calidad.*

16 Muchos ingenieros de software han adoptado un enfoque "ágil" que asimila los cambios en forma gradual y creciente, con lo que controlan su efecto y costo. Los métodos ágiles se estudian en el capítulo 3.

Realidad: Uno de los mecanismos más eficaces de asegurar la calidad del software puede aplicarse desde la concepción del proyecto: *la revisión técnica*. Las revisiones del software (descritas en el capítulo 15) son un "filtro de la calidad" que se ha revelado más eficaz que las pruebas para encontrar ciertas clases de defectos de software.

Mito: *El único producto del trabajo que se entrega en un proyecto exitoso es el programa que funciona.*

Realidad: Un programa que funciona sólo es una parte de una configuración de software que incluye muchos elementos. Son varios los productos terminados (modelos, documentos, planes) que proporcionan la base de la ingeniería exitosa y, lo más importante, que guían el apoyo para el software.

Mito: *La ingeniería de software hará que generemos documentación voluminosa e innecesaria, e invariablemente nos retrasará.*

Realidad: La ingeniería de software no consiste en producir documentos. Se trata de crear un producto de calidad. La mejor calidad conduce a menos repeticiones, lo que da como resultado tiempos de entrega más cortos.

Muchos profesionales del software reconocen la falacia de los mitos mencionados. Es lamentable que las actitudes y métodos habituales nutran la administración y las prácticas técnicas deficientes, aun cuando la realidad dicta un enfoque mejor. El primer paso hacia la formulación de soluciones prácticas para la ingeniería de software es el reconocimiento de las realidades en este campo.

1.7 CÓMO COMIENZA TODO

Todo proyecto de software se desencadena por alguna necesidad de negocios: la de corregir un defecto en una aplicación existente, la de adaptar un "sistema heredado" a un ambiente de negocios cambiante, la de ampliar las funciones y características de una aplicación ya existente o la necesidad de crear un producto, servicio o sistema nuevo.

Al comenzar un proyecto de software, es frecuente que las necesidades del negocio se expresen de manera informal como parte de una simple conversación. La plática que se presenta en el recuadro que sigue es muy común.

CASASEGURA[17]

Cómo se inicia un proyecto

La escena: Sala de juntas en CPI Corporation, empresa (ficticia) que manufactura productos de consumo para uso doméstico y comercial.

Participantes: Mal Golden, alto directivo de desarrollo de productos; Lisa Pérez, gerente comercial; Lee Warren, gerente de ingeniería; Joe Camalleri, VP ejecutivo, desarrollo de negocios.

La conversación:

Joe: Oye, Lee, ¿qué es eso que oí acerca de que tu gente va a desarrollar no sé qué? ¿Una caja inalámbrica universal general?

Lee: Es sensacional… más o menos del tamaño de una caja de cerillos pequeña… podemos conectarla a sensores de todo tipo, una cámara digital… a cualquier cosa. Usa el protocolo 802.11g inalámbrico. Permite el acceso a la salida de dispositivos sin cables. Pensamos que llevará a toda una nueva generación de productos.

Joe: ¿Estás de acuerdo, Mal?

Mal: Sí. En realidad, con las ventas tan planas que hemos tenido este año necesitamos algo nuevo. Lisa y yo hemos hecho algo de investigación del mercado y pensamos que tenemos una línea de productos que podría ser algo grande.

17 El proyecto *CasaSegura* se usará en todo el libro para ilustrar los entretelones de un equipo de proyecto que elabora un producto de software. La compañía, el proyecto y las personas son ficticias, pero las situaciones y problemas son reales.

Joe: ¿Cuán grande… tanto como el renglón de utilidades?

Mal (que evita el compromiso directo): Cuéntale nuestra idea, Lisa.

Lisa: Es toda una nueva generación que hemos llamado "productos para la administración del hogar". Le dimos el nombre de *CasaSegura*. Usan la nueva interfaz inalámbrica, proporcionan a los dueños de viviendas o pequeños negocios un sistema controlado por su PC —seguridad del hogar, vigilancia, control de aparatos y equipos—, tú sabes, apaga el aire acondicionado cuando sales de casa, esa clase de cosas.

Lee (dando un brinco): La oficina de ingeniería hizo un estudio de factibilidad técnica de esta idea, Joe. Es algo realizable con un costo bajo de manufactura. La mayor parte del hardware es de línea. Queda pendiente el software, pero no es algo que no podamos hacer.

Joe: Interesante. Pero pregunté sobre las utilidades.

Mal: Las PC han penetrado a 70 por ciento de los hogares de Estados Unidos. Si lo vendemos en el precio correcto, podría ser una aplicación sensacional. Nadie tiene nuestra caja inalámbrica… somos dueños. Nos adelantaremos dos años a la competencia. ¿Las ganancias? Quizá tanto como 30 a 40 millones de dólares en el segundo año.

Joe (sonriente): Llevemos esto al siguiente nivel. Estoy interesado.

Con excepción de una referencia casual, el software no se mencionó en la conversación. Y, sin embargo, es lo que hará triunfar o fracasar la línea de productos *CasaSegura*. El esfuerzo de ingeniería tendrá éxito sólo si también lo tiene el software de *CasaSegura*. El mercado aceptará el producto sólo si el software incrustado en éste satisface las necesidades del cliente (aún no establecidas). En muchos de los capítulos siguientes continuaremos el avance de la ingeniería del software en *CasaSegura*.

1.8 Resumen

El software es un elemento clave en la evolución de sistemas y productos basados en computadoras, y una de las tecnologías más importantes en todo el mundo. En los últimos 50 años, el software ha pasado de ser la solución de un problema especializado y herramienta de análisis de la información a una industria en sí misma. No obstante, aún hay problemas para desarrollar software de alta calidad a tiempo y dentro del presupuesto asignado.

El software —programas, datos e información descriptiva— se dirige a una gama amplia de tecnología y campos de aplicación. El software heredado sigue planteando retos especiales a quienes deben darle mantenimiento.

Los sistemas y aplicaciones basados en web han evolucionado de simples conjuntos de contenido de información a sistemas sofisticados que presentan una funcionalidad compleja y contenido en multimedios. Aunque dichas *webapps* tienen características y requerimientos únicos, son software.

La ingeniería de software incluye procesos, métodos y herramientas que permiten elaborar a tiempo y con calidad sistemas complejos basados en computadoras. El proceso de software incorpora cinco actividades estructurales: comunicación, planeación, modelado, construcción y despliegue que son aplicables a todos los proyectos de software. La práctica de la ingeniería de software es una actividad para resolver problemas, que sigue un conjunto de principios fundamentales.

Muchos mitos del software todavía hacen que administradores y trabajadores se equivoquen, aun cuando ha aumentado nuestro conocimiento colectivo del software y las tecnologías requeridas para elaborarlo. Conforme el lector aprenda más sobre ingeniería de software, comenzará a entender por qué deben rebatirse estos mitos cada vez que surjan.

Problemas y puntos por evaluar

1.1. Dé al menos cinco ejemplos de la forma en que se aplica la ley de las consecuencias imprevistas al software de cómputo.

1.2. Diga algunos ejemplos (tanto positivos como negativos) que indiquen el efecto del software en nuestra sociedad.

1.3. Desarrolle sus propias respuestas a las cinco preguntas planteadas al principio de la sección 1.1. Analícelas con sus compañeros estudiantes.

1.4. Muchas aplicaciones modernas cambian con frecuencia, antes de que se presenten al usuario final y después de que la primera versión ha entrado en uso. Sugiera algunos modos de elaborar software para detener el deterioro que produce el cambio.

1.5. Considere las siete categorías de software presentadas en la sección 1.1.2. ¿Piensa que puede aplicarse a cada una el mismo enfoque de ingeniería de software? Explique su respuesta.

1.6. La figura 1.3 muestra las tres capas de la ingeniería de software arriba de otra llamada "compromiso con la calidad". Esto implica un programa de calidad organizacional como el enfoque de la administración total de la calidad. Haga un poco de investigación y desarrolle los lineamientos de los elementos clave de un programa para la administración de la calidad.

1.7. ¿Es aplicable la ingeniería de software cuando se elaboran *webapps*? Si es así, ¿cómo puede modificarse para que asimile las características únicas de éstas?

1.8. A medida que el software gana ubicuidad, los riesgos para el público (debidos a programas defectuosos) se convierten en motivo de preocupación significativa. Desarrolle un escenario catastrófico pero realista en el que la falla de un programa de cómputo pudiera ocasionar un gran daño (económico o humano).

1.9. Describa con sus propias palabras una estructura de proceso. Cuando se dice que las actividades estructurales son aplicables a todos los proyectos, ¿significa que se realizan las mismas tareas en todos los proyectos sin que importe su tamaño y complejidad? Explique su respuesta.

1.10. Las actividades sombrilla ocurren a través de todo el proceso del software. ¿Piensa usted que son aplicables por igual a través del proceso, o que algunas se concentran en una o más actividades estructurales?

1.11. Agregue dos mitos adicionales a la lista presentada en la sección 1.6. También diga la realidad que acompaña al mito.

Lecturas adicionales y fuentes de información[18]

Hay literalmente miles de libros escritos sobre software de cómputo. La gran mayoría analiza lenguajes de programación o aplicaciones de software, pero algunos estudian al software en sí mismo. Pressman y Herron (*Software Shock*, Dorset House, 1991) presentaron un estudio temprano (dirigido a las personas comunes) sobre el software y la forma en la que lo elaboran los profesionales. El libro de Negroponte que se convirtió en un éxito de ventas (*Being Digital*, Alfred A. Knopf, Inc., 1995) describe el panorama de la computación y su efecto general en el siglo XXI. DeMarco (*Why Does Software Cost So Much?*, Dorset House, 1995) ha producido varios ensayos amenos y profundos sobre el software y el proceso con el que se elabora.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) afirma que el "flagelo moderno" de los errores en el software puede eliminarse y sugiere formas de lograrlo. Compaine (*Digital Divide: Facing A Crisis or Creating a Myth*, MIT Press, 2001) asegura que la "división" entre aquellos que tienen acceso a recursos de la información (por ejemplo, la web) y los que no lo tienen se está estrechando conforme avanzamos en la primera década de este siglo. Los libros escritos por Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) y Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introducen el concepto de software de "mundo abierto" y predicen un ambiente inalámbrico en el que el software deba adaptarse a los requerimientos que surjan en tiempo real.

---

18 La sección de "Lecturas adicionales y fuentes de información" que se presenta al final de cada capítulo expone un panorama breve de fuentes impresas que ayudan a aumentar la comprensión de los principales temas presentados. El autor ha creado un sitio web para apoyar al libro *Ingeniería de software: enfoque del profesional* en www.mhhe.com/compsci/pressman. Entre los muchos temas que se abordan en dicho sitio, se encuentran desde los recursos de la ingeniería de software capítulo por capítulo hasta información basada en web que complementa el material presentado. Como parte de esos recursos se halla un vínculo hacia Amazon.com para cada libro citado en esta sección.

El estado actual de la ingeniería y del proceso de software se determina mejor a partir de publicaciones tales como *IEEE Software, IEEE Computer, CrossTalk* y *IEEE Transactions on Software Engineering*. Publicaciones periódicas como *Application Development Trends* y *Cutter IT Journal* con frecuencia contienen artículos sobre temas de ingeniería de software. La disciplina se "resume" cada año en *Proceeding of the International Conference on Software Engineering*, patrocinada por IEEE y ACM, y se analiza a profundidad en revistas tales como *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes* y *Annals of Software Engineering*. Hay decenas de miles de sitios web dedicados a la ingeniería y al proceso de software.

En años recientes se han publicado muchos libros que abordan el proceso y la ingeniería de software. Algunos presentan un panorama de todo el proceso, mientras otros profundizan en algunos temas importantes y omiten otros. Entre los más populares (¡además del que tiene usted en sus manos!) se encuentran los siguientes:

Abran, A., and J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge,* IEEE, 2002.

Andersson, E., et al., *Software Engineering for Internet Applications,* The MIT Press, 2006.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2d ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice,* 3d ed., Prentice-Hall, 2005.

Schach, S., *Object-Oriented and Classical Software Engineering,* 7th ed., McGraw-Hill, 2006.

Sommerville, I., *Software Engineering,* 8th ed., Addison-Wesley, 2006.

Tsui, F., and O. Karam, *Essentials of Software Engineering,* Jones & Bartlett Publishers, 2006.

En las últimas décadas, son muchos los estándares para la ingeniería de software que han sido publicados por IEEE, ISO y sus organizaciones. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) proporciona una revisión útil de los estándares relevantes y la forma en la que se aplican a proyectos reales.

En internet se encuentra disponible una amplia variedad de fuentes acerca de la ingeniería y el proceso de software. Una lista actualizada de referencias en la Red Mundial que son útiles para el proceso de software se encuentra en el sitio web del libro, en la dirección **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PARTE

Uno

El proceso del software

En esta parte de la obra, aprenderá sobre el proceso que genera una estructura para la práctica de la ingeniería de software. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Qué es el proceso del software?
- ¿Cuáles son las actividades estructurales generales que están presentes en todo proceso del software?
- ¿Cómo se modelan los procesos y cuáles son los patrones del proceso?
- ¿Cuáles son los modelos prescriptivos del proceso y cuáles son sus fortalezas y debilidades?
- ¿Por qué la *agilidad* es un imperativo en la ingeniería de software moderna?
- ¿Qué es un desarrollo ágil del software y en qué se diferencia de los modelos más tradicionales del proceso?

Una vez respondidas estas preguntas, el lector estará mejor preparado para entender el contexto en el que se aplica la práctica de la ingeniería de software.

CAPÍTULO

2 MODELOS DEL PROCESO

CONCEPTOS CLAVE

En un libro fascinante que expone el punto de vista de un economista sobre el software y su ingeniería, Howard Baetjer, Jr. [Bae98] comenta acerca del proceso del software.

> Debido a que el software, como todo capital, es conocimiento incorporado y a que el conocimiento originalmente se halla disperso, tácito, latente e incompleto en gran medida, el desarrollo de software es un proceso de aprendizaje social. El proceso es un diálogo en el que el conocimiento que debe convertirse en software se reúne e incorpora en éste. El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes, y entre diseñadores y herramientas en evolución [tecnología]. Es un proceso que se repite y en el que la herramienta que evoluciona sirve por sí misma como medio para la comunicación: con cada nueva ronda del diálogo se genera más conocimiento útil a partir de las personas involucradas.

En realidad, la elaboración de software de computadora es un proceso reiterativo de aprendizaje social, y el resultado, algo que Baetjer llamaría "capital de software", es la reunión de conocimiento recabado, depurado y organizado a medida que se realiza el proceso.

Pero desde el punto de vista técnico, ¿qué es exactamente un proceso del software? En el contexto de este libro, se define *proceso del software* como una estructura para las actividades, acciones y tareas que se requieren a fin de construir software de alta calidad. ¿"Proceso" es sinónimo de "ingeniería de software"? La respuesta es "sí y no". Un proceso del software define el enfoque adoptado mientras se hace ingeniería sobre el software. Pero la ingeniería de software también incluye tecnologías que pueblan el proceso: métodos técnicos y herramientas automatizadas.

Más importante aún, la ingeniería de software es llevada a cabo por personas creativas y preparadas que deben adaptar un proceso maduro de software a fin de que resulte apropiado para los productos que construyen y para las demandas de su mercado.

UNA MIRADA RÁPIDA

¿Qué es? Cuando se trabaja en la construcción de un producto o sistema, es importante ejecutar una serie de pasos predecibles —el mapa de carreteras que lo ayuda a obtener a tiempo un resultado de alta calidad—. El mapa que se sigue se llama "proceso del software".

¿Quién lo hace? Los ingenieros de software y sus gerentes adaptan el proceso a sus necesidades y luego lo siguen. Además, las personas que solicitaron el software tienen un papel en el proceso de definición, elaboración y prueba.

¿Por qué es importante? Porque da estabilidad, control y organización a una actividad que puede volverse caótica si se descontrola. Sin embargo, un enfoque moderno de ingeniería de software debe ser "ágil". Debe incluir sólo aquellas actividades, controles y productos del trabajo que sean apropiados para el equipo del proyecto y para el producto que se busca obtener.

¿Cuáles son los pasos? En un nivel detallado, el proceso que se adopte depende del software que se esté elaborando. Un proceso puede ser apropiado para crear software destinado a un sistema de control electrónico de un aeroplano, mientras que para la creación de un sitio web será necesario un proceso completamente distinto.

¿Cuál es el producto final? Desde el punto de vista de un ingeniero de software, los productos del trabajo son los programas, documentos y datos que se producen como consecuencia de las actividades y tareas definidas por el proceso.

¿Cómo me aseguro de que lo hice bien? Hay cierto número de mecanismos de evaluación del proceso del software que permiten que las organizaciones determinen la "madurez" de su proceso. Sin embargo, la calidad, oportunidad y viabilidad a largo plazo del producto que se elabora son los mejores indicadores de la eficacia del proceso que se utiliza.

2.1 UN MODELO GENERAL DE PROCESO

En el capítulo 1 se definió un proceso como la colección de actividades de trabajo, acciones y tareas que se realizan cuando va a crearse algún producto terminado. Cada una de las actividades, acciones y tareas se encuentra dentro de una estructura o modelo que define su relación tanto con el proceso como entre sí.

En la figura 2.1 se representa el proceso del software de manera esquemática. En dicha figura, cada actividad estructural está formada por un conjunto de acciones de ingeniería de software y cada una de éstas se encuentra definida por un *conjunto de tareas* que identifica las tareas del trabajo que deben realizarse, los productos del trabajo que se producirán, los puntos de aseguramiento de la calidad que se requieren y los puntos de referencia que se utilizarán para evaluar el avance.

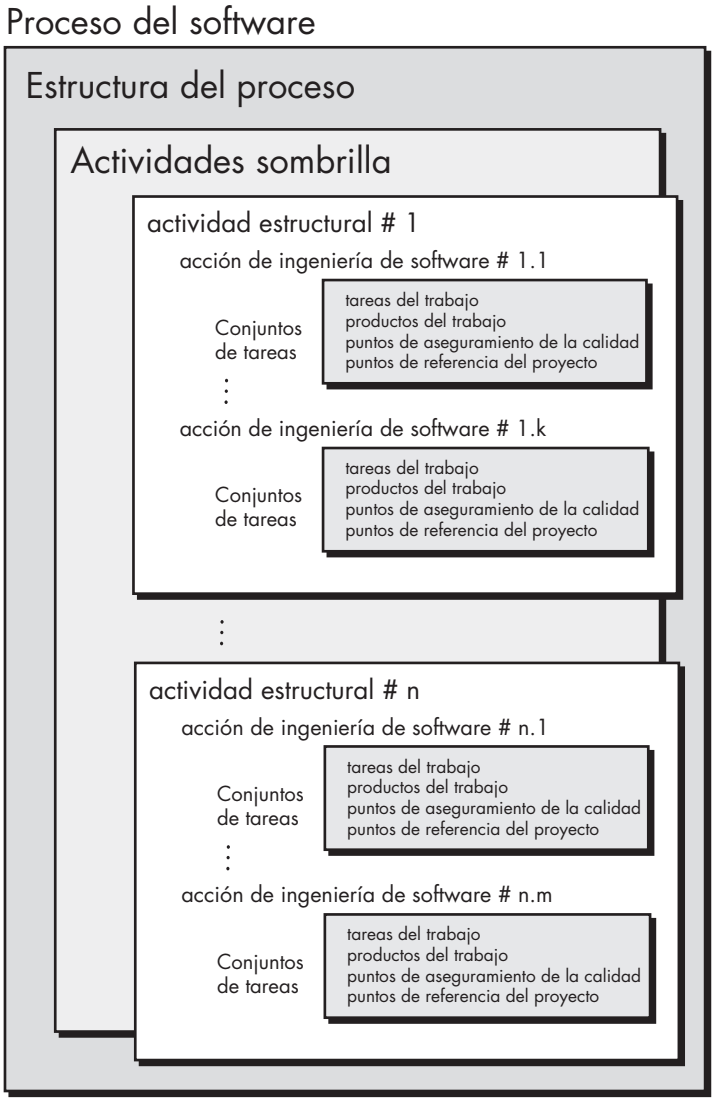
PUNTO CLAVE

La jerarquía del trabajo técnico dentro del proceso del software es: actividades, acciones que contiene y tareas constituyentes.

Como se dijo en el capítulo 1, una estructura general para la ingeniería de software define cinco actividades estructurales: **comunicación, planeación, modelado, construcción** y **despliegue**. Además, a lo largo de todo el proceso se aplica un conjunto de actividades som-

FIGURA 2.1

Estructura de un proceso del software



brilla: seguimiento y control del proyecto, administración de riesgos, aseguramiento de la calidad, administración de la configuración, revisiones técnicas, entre otras.

Cita:

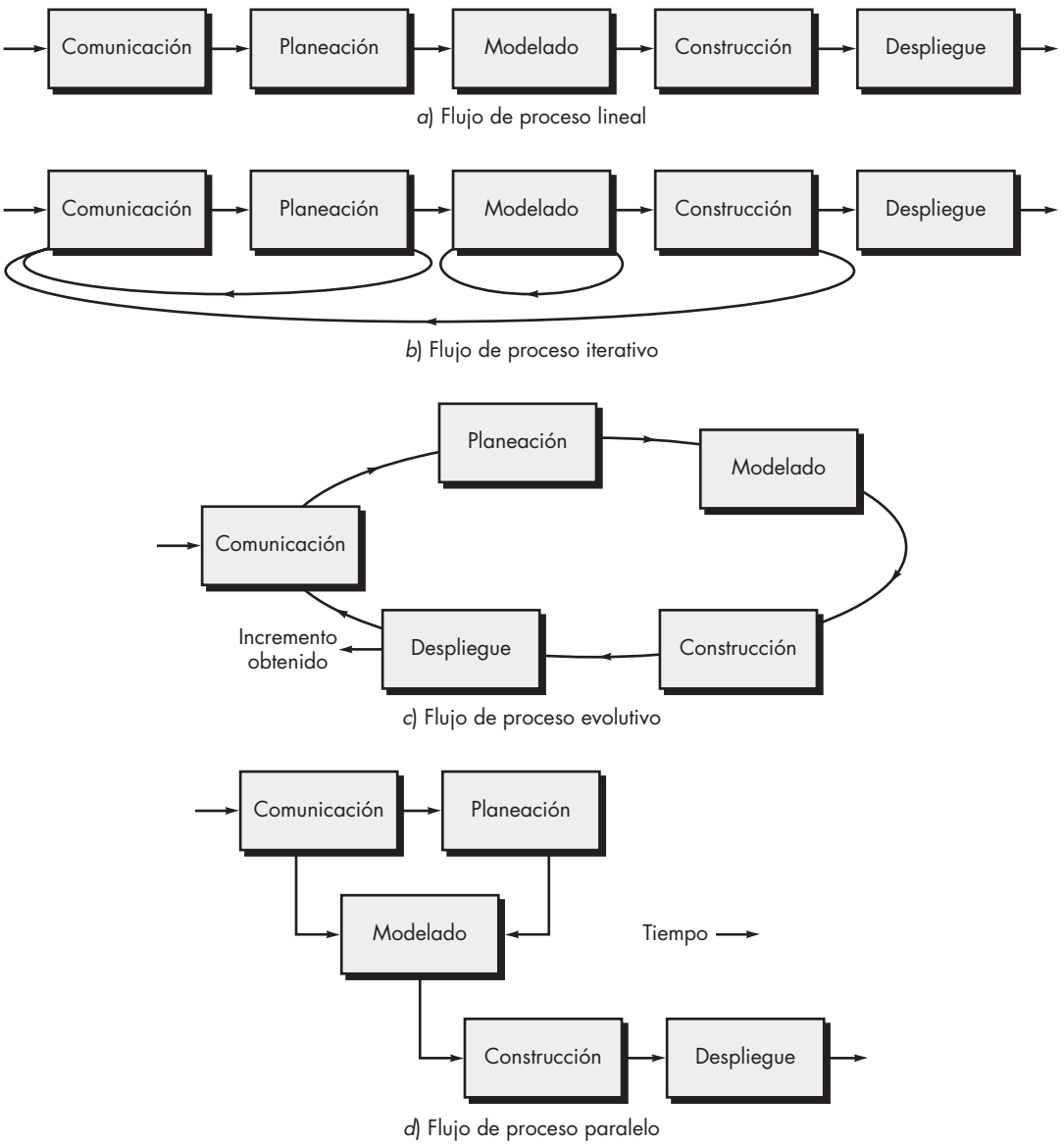
"Pensamos que los desarrolladores de software pierden de vista una verdad fundamental: la mayor parte de organizaciones no saben lo que hacen. Piensan que lo saben, pero no es así."

Tom DeMarco

El lector debe observar que aún no se menciona un aspecto importante del proceso del software. En la figura 2.2 se ilustra dicho aspecto —llamado *flujo del proceso*— y se describe la manera en que están organizadas las actividades estructurales y las acciones y tareas que ocurren dentro de cada una con respecto de la secuencia y el tiempo.

Un *flujo de proceso lineal* ejecuta cada una de las cinco actividades estructurales en secuencia, comenzando por la comunicación y terminando con el despliegue (véase la figura 2.2*a*). Un *flujo de proceso iterativo* repite una o más de las actividades antes de pasar a la siguiente (véase la figura 2.2*b*). Un *flujo de proceso evolutivo* realiza las actividades en forma "circular". A través de las cinco actividades, cada circuito lleva a una versión más completa del software (véase la figura 2.2*c*). Un *flujo de proceso paralelo* (véase la figura 2.2*d*) ejecuta una o más actividades en

FIGURA 2.2 Flujo del proceso



paralelo con otras (por ejemplo, el modelado de un aspecto del software tal vez se ejecute en paralelo con la construcción de otro aspecto del software).

2.1.1 Definición de actividad estructural

Aunque en el capítulo 1 se describieron cinco actividades estructurales y se dio una definición básica de cada una, un equipo de software necesitará mucha más información antes de poder ejecutar de manera apropiada cualquiera de ellas como parte del proceso del software. Por tanto, surge una pregunta clave: *¿qué acciones son apropiadas para una actividad estructural, dados la naturaleza del problema por resolver, las características de las personas que hacen el trabajo y los participantes que patrocinan el proyecto?*

¿Cómo se transforma una actividad estructural cuando cambia la naturaleza del proyecto?

Para un proyecto de software pequeño solicitado por una persona (en una ubicación remota) con requerimientos sencillos y directos, la actividad de comunicación tal vez no incluya algo más que una llamada telefónica con el participante apropiado. Entonces, la única acción necesaria es una *conversación telefónica*, y las tareas del trabajo (el *conjunto de tareas*) que engloba son las siguientes:

1. Hacer contacto con el participante por vía telefónica.
2. Analizar los requerimientos y tomar notas.
3. Organizar las notas por escrito en una formulación breve de los requerimientos.
4. Enviar correo electrónico al participante para que revise y apruebe.

Si el proyecto fuera considerablemente más complejo, con muchos participantes y cada uno con un distinto conjunto de requerimientos (a veces en conflicto), la actividad de comunicación puede tener seis acciones distintas (descritas en el capítulo 5): *concepción, indagación, elaboración, negociación, especificación* y *validación*. Cada una de estas acciones de la ingeniería del software tendrá muchas tareas de trabajo y un número grande de diferentes productos finales.

PUNTO CLAVE

Diferentes proyectos demandan diferentes conjuntos de tareas. El equipo de software elige el conjunto de tareas con base en las características del problema y el proyecto.

2.1.2 Identificación de un conjunto de tareas

En relación con la figura 2.1, cada acción de la ingeniería de software (por ejemplo, *obtención*, asociada a la actividad de comunicación) se representa por cierto número de distintos *conjuntos de tareas*, cada uno de los cuales es una colección de tareas de trabajo de la ingeniería de software, relacionadas con productos del trabajo, puntos de aseguramiento de la calidad y puntos de referencia del proyecto. Debe escogerse el conjunto de tareas que se adapte mejor a las necesidades del proyecto y a las características del equipo. Esto implica que una acción de la ingeniería de software puede adaptarse a las necesidades específicas del proyecto de software y a las características del equipo del proyecto.

2.1.3 Patrones del proceso

¿Qué es un patrón del proceso?

Cada equipo de software se enfrenta a problemas conforme avanza en el proceso del software. Si se demostrara que existen soluciones fáciles para dichos problemas, sería útil para el equipo abordarlos y resolverlos rápidamente. Un *patrón del proceso*[1] describe un problema relacionado con el proceso que se encuentra durante el trabajo de ingeniería de software, identifica el ambiente en el que surge el problema y sugiere una o más soluciones para el mismo. Dicho de manera general, un patrón de proceso da un formato [Amb98]: un método consistente para describir soluciones del problema en el contexto del proceso del software. Al combinar patrones, un equipo de software resuelve problemas y construye el proceso que mejor satisfaga las necesidades de un proyecto.

1 En el capítulo 12 se hace el análisis detallado de los patrones.

INFORMACIÓN

Conjunto de tareas

Un conjunto de tareas define el trabajo real por efectuar a fin de cumplir los objetivos de una acción de ingeniería de software. Por ejemplo, la *indagación* (mejor conocida como "recabar los requerimientos") es una acción importante de la ingeniería de software que ocurre durante la actividad de comunicación. La meta al recabar los requerimientos es entender lo que los distintos participantes desean del software que se va a elaborar.

Para un proyecto pequeño y relativamente sencillo, el conjunto de tareas para la indagación de requerimientos tendrá un aspecto parecido al siguiente:

1. Elaborar la lista de participantes del proyecto.
2. Invitar a todos los participantes a una reunión informal.
3. Pedir a cada participante que haga una relación de las características y funciones que requiere.
4. Analizar los requerimientos y construir la lista definitiva.
5. Ordenar los requerimientos según su prioridad.
6. Identificar las áreas de incertidumbre.

Para un proyecto de software más grande y complejo se requerirá de un conjunto de tareas diferente que quizá esté constituido por las siguientes tareas de trabajo:

1. Hacer la lista de participantes del proyecto.
2. Entrevistar a cada participante por separado a fin de determinar los deseos y necesidades generales.
3. Formar la lista preliminar de las funciones y características con base en las aportaciones del participante.
4. Programar una serie de reuniones para facilitar la elaboración de las especificaciones de la aplicación.
5. Celebrar las reuniones.
6. Producir en cada reunión escenarios informales de usuario.
7. Afinar los escenarios del usuario con base en la retroalimentación de los participantes.
8. Formar una lista revisada de los requerimientos de los participantes.
9. Usar técnicas de despliegue de la función de calidad para asignar prioridades a los requerimientos.
10. Agrupar los requerimientos de modo que puedan entregarse en forma paulatina y creciente.
11. Resaltar las limitantes y restricciones que se introducirán al sistema.
12. Analizar métodos para validar el sistema.

Los dos conjuntos de tareas mencionados sirven para "recabar los requerimientos", pero son muy distintos en profundidad y formalidad. El equipo de software elige el conjunto de tareas que le permita alcanzar la meta de cada acción con calidad y agilidad.

Cita:

"La repetición de patrones es algo muy diferente de la repetición de las partes. En realidad, las distintas partes serán únicas porque los patrones son los mismos."

Christopher Alexander

Los patrones se definen en cualquier nivel de abstracción.[2] En ciertos casos, un patrón puede usarse para describir un problema (y su solución) asociado con un modelo completo del proceso (por ejemplo, hacer prototipos). En otras situaciones, los patrones se utilizan para describir un problema (y su solución) asociado con una actividad estructural (por ejemplo, **planeación**) o una acción dentro de una actividad estructural (estimación de proyectos).

Ambler [Amb98] ha propuesto un formato para describir un patrón del proceso:

Nombre del patrón. El patrón recibe un nombre significativo que lo describe en el contexto del proceso del software (por ejemplo, **RevisionesTécnicas**).

Fuerzas. El ambiente en el que se encuentra el patrón y los aspectos que hacen visible el problema y afectan su solución.

Tipo. Se especifica el tipo de patrón. Ambler [Amb98] sugiere tres tipos:

1. *Patrón de etapa*: define un problema asociado con una actividad estructural para el proceso. Como una actividad estructural incluye múltiples acciones y tareas del trabajo, un patrón de la etapa incorpora múltiples patrones de la tarea (véase a continuación) que son relevantes para la etapa (actividad estructural). Un ejemplo de patrón de etapa sería **EstablecerComunicación**. Este patrón incorporaría el patrón de tarea **RecabarRequerimientos** y otros más.
2. *Patrón de tarea*: define un problema asociado con una acción o tarea de trabajo de la ingeniería de software y que es relevante para el éxito de la práctica de ingeniería de software (por ejemplo, **RecabarRequerimientos** es un patrón de tarea).

PUNTO CLAVE

Un formato de patrón proporciona un medio consistente para describir al patrón.

2 Los patrones son aplicables a muchas actividades de la ingeniería de software. El análisis, el diseño y la prueba de patrones se estudian en los capítulos 7, 9, 10, 12 y 14. Los patrones y "antipatrones" para las actividades de administración de proyectos se analizan en la parte 4 del libro.

3. *Patrón de fase*: define la secuencia de las actividades estructurales que ocurren dentro del proceso, aun cuando el flujo general de las actividades sea de naturaleza iterativa. Un ejemplo de patrón de fase es **ModeloEspiral** o **Prototipos**.[3]

Contexto inicial. Describe las condiciones en las que se aplica el patrón. Antes de iniciar el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo han ocurrido? 2) ¿Cuál es el estado de entrada para el proceso? 3) ¿Qué información de ingeniería de software o del proyecto ya existe?

Por ejemplo, el patrón **Planeación** (patrón de etapa) requiere que: 1) los clientes y los ingenieros de software hayan establecido una comunicación colaboradora; 2) haya terminado con éxito cierto número de patrones de tarea [especificado] para el patrón **Comunicación**; y 3) se conozcan el alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

Problema. El problema específico que debe resolver el patrón.

Solución. Describe cómo implementar con éxito el patrón. Esta sección describe la forma en la que se modifica el estado inicial del proceso (que existe antes de implementar el patrón) como consecuencia de la iniciación del patrón. También describe cómo se transforma la información sobre la ingeniería de software o sobre el proyecto, disponible antes de que inicie el patrón, como consecuencia de la ejecución exitosa del patrón.

Contexto resultante. Describe las condiciones que resultarán una vez que se haya implementado con éxito el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo deben haber ocurrido? 2) ¿Cuál es el estado de salida del proceso? 3) ¿Qué información sobre la ingeniería de software o sobre el proyecto se ha desarrollado?

Patrones relacionados. Proporciona una lista de todos los patrones de proceso directamente relacionados con éste. Puede representarse como una jerarquía o en alguna forma de diagrama. Por ejemplo, el patrón de etapa **Comunicación** incluye los patrones de tarea: **EquipoDelProyecto, LineamientosDeColaboración, DefiniciónDeAlcances, RecabarRequerimientos, DescripciónDeRestricciones** y **CreaciónDeEscenarios**.

Usos y ejemplos conocidos. Indica las instancias específicas en las que es aplicable el patrón. Por ejemplo, **Comunicación** es obligatoria al principio de todo proyecto de software, es recomendable a lo largo del proyecto y de nuevo obligatoria una vez alcanzada la actividad de despliegue.

WebRef

En la dirección **www.ambysoft.com/processPatternsPage.html** se encuentran recursos amplios sobre los patrones de proceso.

Los patrones de proceso dan un mecanismo efectivo para enfrentar problemas asociados con cualquier proceso del software. Los patrones permiten desarrollar una descripción jerárquica del proceso, que comienza en un nivel alto de abstracción (un patrón de fase). Después se mejora la descripción como un conjunto de patrones de etapa que describe las actividades estructurales y se mejora aún más en forma jerárquica en patrones de tarea más detallados para cada patrón de etapa. Una vez desarrollados los patrones de proceso, pueden reutilizarse para la definición de variantes del proceso, es decir, un equipo de software puede definir un modelo de proceso específico con el empleo de los patrones como bloques constituyentes del modelo del proceso.

2.2 Evaluación y mejora del proceso

La existencia de un proceso del software no es garantía de que el software se entregue a tiempo, que satisfaga las necesidades de los consumidores o que tenga las características técnicas que

3 Estos patrones de fase se estudian en la sección 2.3.3.

INFORMACIÓN

Ejemplo de patrón de proceso

El siguiente patrón de proceso abreviado describe un enfoque aplicable en el caso en el que los participantes tienen una idea general de lo que debe hacerse, pero no están seguros de los requerimientos específicos de software.

Nombre del patrón. RequerimientosPocoClaros

Intención. Este patrón describe un enfoque para construir un modelo (un prototipo) que los participantes pueden evaluar en forma iterativa, en un esfuerzo por identificar o solidificar los requerimientos de software.

Tipo. Patrón de fase.

Contexto inicial. Antes de iniciar este patrón deben cumplirse las siguientes condiciones: 1) se ha identificado a los participantes; 2) se ha establecido un modo de comunicación entre los participantes y el equipo de software; 3) los participantes han identificado el problema general de software que se va a resolver; 4) se ha obtenido el entendimiento inicial del alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

Problema. Los requerimientos son confusos o inexistentes, pero hay un reconocimiento claro de que existe un problema por resolver y que debe hacerse con una solución de software. Los participantes no están seguros de lo que quieren, es decir, no pueden describir con detalle los requerimientos del software.

Solución. Aquí se presentaría una descripción del proceso prototipo, que se describirá más adelante, en la sección 2.3.3.

Contexto resultante. Los participantes aprueban un prototipo de software que identifica los requerimientos básicos (por ejemplo, modos de interacción, características computacionales, funciones de procesamiento). Después de esto, 1) el prototipo quizá evolucione a través de una serie de incrementos para convertirse en el software de producción, o 2) tal vez se descarte el prototipo y el software de producción se elabore con el empleo de otro proceso de patrón.

Patrones relacionados. Los patrones siguientes están relacionados con este patrón: **ComunicaciónConClientes, DiseñoIterativo, DesarrolloIterativo, EvaluaciónDelCliente, ObtenciónDeRequerimientos.**

Usos y ejemplos conocidos. Cuando los requerimientos sean inciertos, es recomendable hacer prototipos.

PUNTO CLAVE

La evaluación busca entender el estado actual del proceso del software con el objeto de mejorarlo.

conducirán a características de calidad de largo plazo (véanse los capítulos 14 y 16). Los patrones de proceso deben acoplarse con una práctica sólida de ingeniería de software (véase la parte 2 del libro). Además, el proceso en sí puede evaluarse para garantizar que cumple con ciertos criterios de proceso básicos que se haya demostrado que son esenciales para el éxito de la ingeniería de software.[4]

En las últimas décadas se han propuesto numerosos enfoques para la evaluación y mejora de un proceso del software:

¿De qué técnicas formales se dispone para evaluar el proceso del software?

Método de evaluación del estándar CMMI para el proceso de mejora (**SCAMPI,** por sus siglas en inglés): proporciona un modelo de cinco fases para evaluar el proceso: inicio, diagnóstico, establecimiento, actuación y aprendizaje. El método SCAMPI emplea el SEI CMMI como la base de la evaluación [SEI00].

Evaluación basada en CMM para la mejora del proceso interno (**CBA IPI,** por sus siglas en inglés): proporciona una técnica de diagnóstico para evaluar la madurez relativa de una organización de software; usa el SEI CMM como la base de la evaluación [Dun01].

SPICE (ISO/IEC 15504): estándar que define un conjunto de requerimientos para la evaluación del proceso del software. El objetivo del estándar es ayudar a las organizaciones a desarrollar una evaluación objetiva de cualquier proceso del software definido [ISO08].

ISO9001:2000 para software: estándar genérico que se aplica a cualquier organización que desee mejorar la calidad general de los productos, sistemas o servicios que proporciona. Por tanto, el estándar es directamente aplicable a las organizaciones y compañías de software [Ant06].

Cita:

"Las organizaciones de software tienen deficiencias significativas en su capacidad de capitalizar las experiencias obtenidas de los proyectos terminados."

NASA

En el capítulo 30 se presenta un análisis detallado de los métodos de evaluación del software y del proceso de mejora.

4 En la publicación CMMI [CMM07] del SEI, se describen con muchos detalles las características de un proceso del software y los criterios para un proceso exitoso.

2.3 Modelos de proceso prescriptivo

Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software. La historia indica que estos modelos tradicionales han dado cierta estructura útil al trabajo de ingeniería de software y que constituyen un mapa razonablemente eficaz para los equipos de software. Sin embargo, el trabajo de ingeniería de software y el producto que genera siguen "al borde del caos".

Cita:

"Si el proceso está bien, los resultados cuidarán de sí mismos."

Takashi Osada

En un artículo intrigante sobre la extraña relación entre el orden y el caos en el mundo del software, Nogueira y sus colegas [Nog00] afirman lo siguiente:

> El borde del caos se define como "el estado natural entre el orden y el caos, un compromiso grande entre la estructura y la sorpresa" [Kau95]. El borde del caos se visualiza como un estado inestable y parcialmente estructurado [...] Es inestable debido a que se ve atraído constantemente hacia el caos o hacia el orden absoluto.
>
> Tenemos la tendencia de pensar que el orden es el estado ideal de la naturaleza. Esto podría ser un error [...] las investigaciones apoyan la teoría de que la operación que se aleja del equilibrio genera creatividad, procesos autoorganizados y rendimientos crecientes [Roo96]. El orden absoluto significa ausencia de variabilidad, que podría ser una ventaja en los ambientes impredecibles. El cambio ocurre cuando hay cierta estructura que permita que el cambio pueda organizarse, pero que no sea tan rígida como para que no pueda suceder. Por otro lado, demasiado caos hace imposible la coordinación y la coherencia. La falta de estructura no siempre significa desorden.

PUNTO CLAVE

Los modelos de proceso prescriptivo definen un conjunto prescrito de elementos del proceso y un flujo predecible para el trabajo del proceso.

Las implicaciones filosóficas de este argumento son significativas para la ingeniería de software. Si los modelos de proceso prescriptivo[5] buscan generar estructura y orden, ¿son inapropiados para el mundo del software, que se basa en el cambio? Pero si rechazamos los modelos de proceso tradicional (y el orden que implican) y los reemplazamos con algo menos estructurado, ¿hacemos imposible la coordinación y coherencia en el trabajo de software?

No hay respuestas fáciles para estas preguntas, pero existen alternativas disponibles para los ingenieros de software. En las secciones que siguen se estudia el enfoque de proceso prescriptivo en el que los temas dominantes son el orden y la consistencia del proyecto. El autor los llama "prescriptivos" porque prescriben un conjunto de elementos del proceso: actividades estructurales, acciones de ingeniería de software, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo del proceso también prescribe un flujo del proceso (también llamado *flujo de trabajo*), es decir, la manera en la que los elementos del proceso se relacionan entre sí.

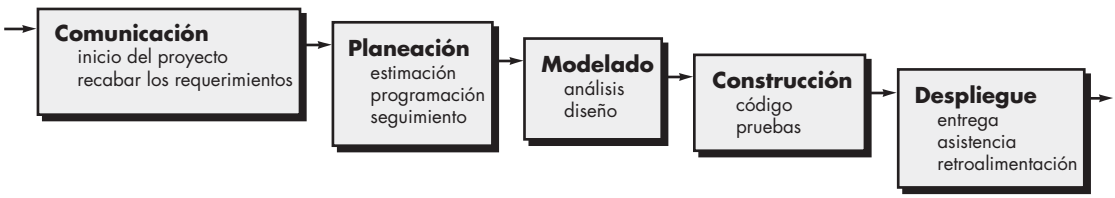
Todos los modelos del proceso del software pueden incluir las actividades estructurales generales descritas en el capítulo 1, pero cada una pone distinto énfasis en ellas y define en forma diferente el flujo de proceso que invoca cada actividad estructural (así como acciones y tareas de ingeniería de software).

2.3.1 Modelo de la cascada

Hay veces en las que los requerimientos para cierto problema se comprenden bien: cuando el trabajo desde la **comunicación** hasta el **despliegue** fluye en forma razonablemente lineal. Esta situación se encuentra en ocasiones cuando deben hacerse adaptaciones o mejoras bien definidas a un sistema ya existente (por ejemplo, una adaptación para software de contabilidad que es obligatorio hacer debido a cambios en las regulaciones gubernamentales). También ocurre en cierto número limitado de nuevos esfuerzos de desarrollo, pero sólo cuando los requerimientos están bien definidos y tienen una estabilidad razonable.

5 Los modelos de proceso prescriptivo en ocasiones son denominados modelos de proceso "tradicional".

FIGURA 2.3 Modelo de la cascada



El *modelo de la cascada*, a veces llamado *ciclo de vida clásico*, sugiere un enfoque sistemático y secuencial[6] para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado (véase la figura 2.3).

PUNTO CLAVE

El modelo en V ilustra la forma en la que se asocian las acciones de verificación y validación con las primeras acciones de ingeniería.

Una variante de la representación del modelo de la cascada se denomina *modelo en V*. En la figura 2.4 se ilustra el modelo en V [Buc99], donde se aprecia la relación entre las acciones para el aseguramiento de la calidad y aquellas asociadas con la comunicación, modelado y construcción temprana. A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución. Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas (acciones para asegurar la calidad) que validan cada uno de los modelos creados cuando el equipo fue hacia abajo por el lado izquierdo.[7] En realidad, no hay diferencias fundamentales entre el ciclo de vida clásico y el modelo en V. Este último proporciona una forma de visualizar el modo de aplicación de las acciones de verificación y validación al trabajo de ingeniería inicial.

El modelo de la cascada es el paradigma más antiguo de la ingeniería de software. Sin embargo, en las últimas tres décadas, las críticas hechas al modelo han ocasionado que incluso sus defensores más obstinados cuestionen su eficacia [Han95]. Entre los problemas que en ocasiones surgen al aplicar el modelo de la cascada se encuentran los siguientes:

¿Por qué a veces falla el modelo de la cascada?

1. Es raro que los proyectos reales sigan el flujo secuencial propuesto por el modelo. Aunque el modelo lineal acepta repeticiones, lo hace en forma indirecta. Como resultado, los cambios generan confusión conforme el equipo del proyecto avanza.
2. A menudo, es difícil para el cliente enunciar en forma explícita todos los requerimientos. El modelo de la cascada necesita que se haga y tiene dificultades para aceptar la incertidumbre natural que existe al principio de muchos proyectos.
3. El cliente debe tener paciencia. No se dispondrá de una versión funcional del(de los) programa(s) hasta que el proyecto esté muy avanzado. Un error grande sería desastroso si se detectara hasta revisar el programa en funcionamiento.

Cita:

"Con demasiada frecuencia, el trabajo de software sigue la primera ley del ciclismo: no importa hacia dónde te dirijas, vas hacia arriba y contra el viento."

Anónimo

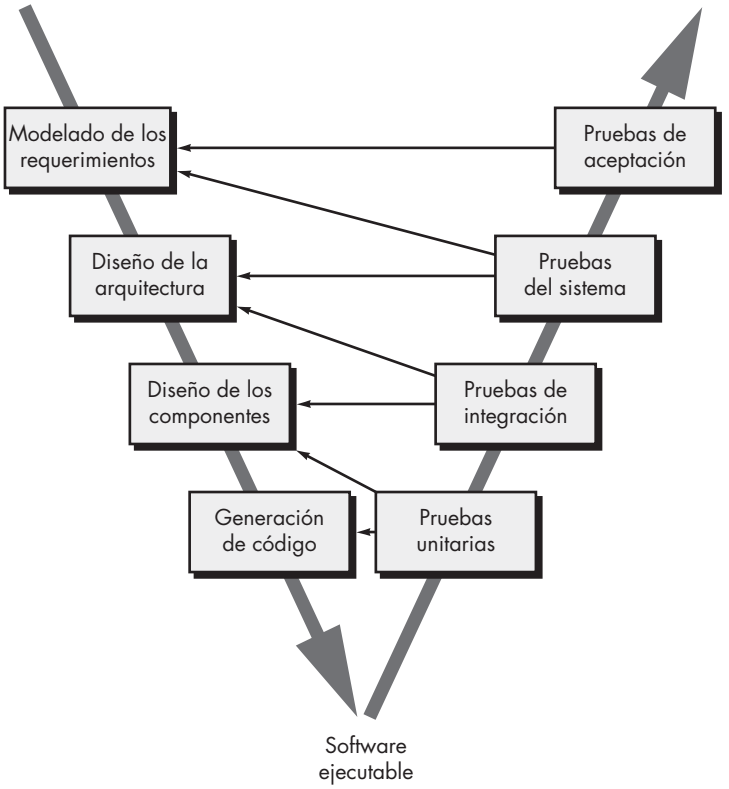
En un análisis interesante de proyectos reales, Bradac [Bra94] encontró que la naturaleza lineal del ciclo de vida clásico llega a "estados de bloqueo" en los que ciertos miembros del equipo de proyecto deben esperar a otros a fin de terminar tareas interdependientes. En realidad, ¡el tiempo de espera llega a superar al dedicado al trabajo productivo! Los estados de bloqueo tienden a ocurrir más al principio y al final de un proceso secuencial lineal.

Hoy en día, el trabajo de software es acelerado y está sujeto a una corriente sin fin de cambios (en las características, funciones y contenido de información). El modelo de la cascada suele ser

6 Aunque el modelo de la cascada propuesto originalmente por Winston Royce [Roy70] prevé los "bucles de retroalimentación", la gran mayoría de organizaciones que aplican este modelo de proceso lo tratan como si fuera estrictamente lineal.

7 En la parte 3 del libro se estudian con detalle las acciones de aseguramiento de la calidad.

FIGURA 2.4
El modelo en V



inapropiado para ese tipo de labor. No obstante, sirve como un modelo de proceso útil en situaciones en las que los requerimientos son fijos y el trabajo avanza en forma lineal hacia el final.

2.3.2 Modelos de proceso incremental

PUNTO CLAVE

El modelo incremental ejecuta una serie de avances, llamados incrementos, que en forma progresiva dan más funcionalidad al cliente conforme se le entrega cada incremento.

Hay muchas situaciones en las que los requerimientos iniciales del software están razonablemente bien definidos, pero el alcance general del esfuerzo de desarrollo imposibilita un proceso lineal. Además, tal vez haya una necesidad imperiosa de dar rápidamente cierta funcionalidad limitada de software a los usuarios y aumentarla en las entregas posteriores de software. En tales casos, se elige un modelo de proceso diseñado para producir el software en incrementos.

El modelo *incremental* combina elementos de los flujos de proceso lineal y paralelo estudiados en la sección 2.1. En relación con la figura 2.5, el modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce "incrementos" de software susceptibles de entregarse [McD93] de manera parecida a los incrementos producidos en un flujo de proceso evolutivo (sección 2.3.3).

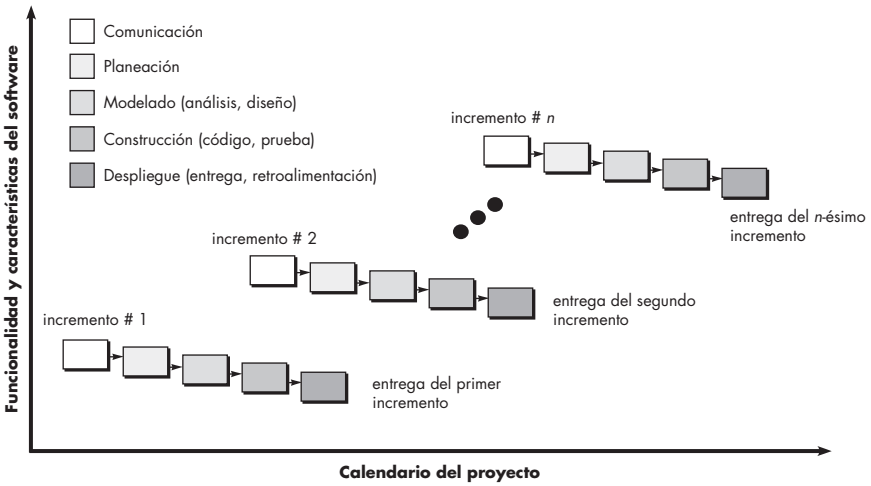
Por ejemplo, un software para procesar textos que se elabore con el paradigma incremental quizá entregue en el primer incremento las funciones básicas de administración de archivos, edición y producción del documento; en el segundo dará herramientas más sofisticadas de edición y producción de documentos; en el tercero habrá separación de palabras y revisión de la ortografía; y en el cuarto se proporcionará la capacidad para dar formato avanzado a las páginas. Debe observarse que el flujo de proceso para cualquier incremento puede incorporar el paradigma del prototipo.

CONSEJO

Su cliente solicita la entrega para una fecha que es imposible de cumplir. Sugiera entregar uno o más incrementos en la fecha que pide, y el resto del software (incrementos adicionales) en un momento posterior.

Cuando se utiliza un modelo incremental, es frecuente que el primer incremento sea el *producto fundamental*. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias (algunas conocidas y otras no). El cliente usa el producto fundamental (o lo somete a una evaluación detallada). Como resultado del uso y/o evaluación,

FIGURA 2.5
El modelo incremental



se desarrolla un plan para el incremento que sigue. El plan incluye la modificación del producto fundamental para cumplir mejor las necesidades del cliente, así como la entrega de características adicionales y más funcionalidad. Este proceso se repite después de entregar cada incremento, hasta terminar el producto final.

El modelo de proceso incremental se centra en que en cada incremento se entrega un producto que ya opera. Los primeros incrementos son versiones desnudas del producto final, pero proporcionan capacidad que sirve al usuario y también le dan una plataforma de evaluación.[8]

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande tal vez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante.

2.3.3 Modelos de proceso evolutivo

PUNTO CLAVE
El modelo del proceso evolutivo genera en cada iteración una versión final cada vez más completa del software.

El software, como todos los sistemas complejos, evoluciona en el tiempo. Es frecuente que los requerimientos del negocio y del producto cambien conforme avanza el desarrollo, lo que hace que no sea realista trazar una trayectoria rectilínea hacia el producto final; los plazos apretados del mercado hacen que sea imposible la terminación de un software perfecto, pero debe lanzarse una versión limitada a fin de aliviar la presión de la competencia o del negocio; se comprende bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse. En estas situaciones y otras parecidas se necesita un modelo de proceso diseñado explícitamente para adaptarse a un producto que evoluciona con el tiempo.

Los modelos evolutivos son iterativos. Se caracterizan por la manera en la que permiten desarrollar versiones cada vez más completas del software. En los párrafos que siguen se presentan dos modelos comunes de proceso evolutivo.

8 Es importante observar que para todos los modelos de proceso "ágiles" que se estudian en el capítulo 3 también se usa la filosofía incremental.

Cita:

"Planee para lanzar uno. De todos modos hará eso. Su única elección es si tratará de vender a sus clientes lo que lanzó."

Frederick P. Brooks

Hacer prototipos. Es frecuente que un cliente defina un conjunto de objetivos generales para el software, pero que no identifique los requerimientos detallados para las funciones y características. En otros casos, el desarrollador tal vez no esté seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debe adoptar la interacción entre el humano y la máquina. En estas situaciones, y muchas otras, el *paradigma de hacer prototipos* tal vez ofrezca el mejor enfoque.

Aunque es posible hacer prototipos como un modelo de proceso aislado, es más común usarlo como una técnica que puede implementarse en el contexto de cualquiera de los modelos de proceso descritos en este capítulo. Sin importar la manera en la que se aplique, el paradigma de hacer prototipos le ayudará a usted y a otros participantes a mejorar la comprensión de lo que hay que elaborar cuando los requerimientos no están claros.

CONSEJO

Cuando su cliente tiene una necesidad legítima, pero ignora los detalles, como primer paso desarrolle un prototipo.

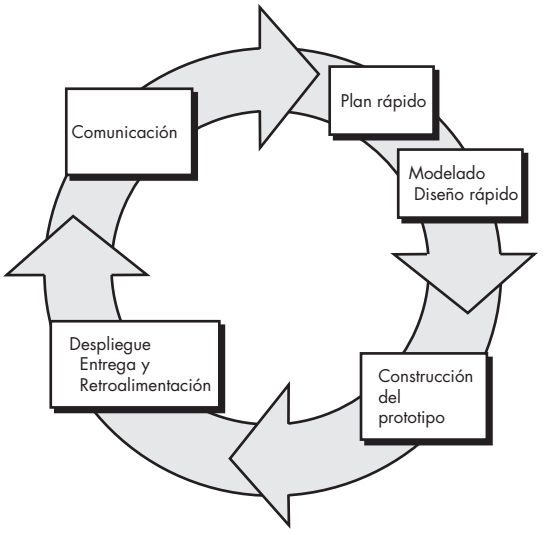
El paradigma de hacer prototipos (véase la figura 2.6) comienza con comunicación. Usted se reúne con otros participantes para definir los objetivos generales del software, identifica cualesquiera requerimientos que conozca y detecta las áreas en las que es imprescindible una mayor definición. Se planea rápidamente una iteración para hacer el prototipo, y se lleva a cabo el modelado (en forma de un "diseño rápido"). Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales (por ejemplo, disposición de la interfaz humana o formatos de la pantalla de salida). El diseño rápido lleva a la construcción de un prototipo. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos. La iteración ocurre a medida de que el prototipo es afinado para satisfacer las necesidades de distintos participantes, y al mismo tiempo le permite a usted entender mejor lo que se necesita hacer.

El ideal es que el prototipo sirva como mecanismo para identificar los requerimientos del software. Si va a construirse un prototipo, pueden utilizarse fragmentos de programas existentes o aplicar herramientas (por ejemplo, generadores de reportes y administradores de ventanas) que permitan generar rápidamente programas que funcionen.

Pero, ¿qué hacer con el prototipo cuando ya sirvió para el propósito descrito? Brooks [Bro95] da una respuesta:

> En la mayoría de proyectos es raro que el primer sistema elaborado sea utilizable. Tal vez sea muy lento, muy grande, difícil de usar o todo a la vez. No hay más alternativa que comenzar de nuevo, con más inteligencia, y construir una versión rediseñada en la que se resuelvan los problemas.

FIGURA 2.6
El paradigma de hacer prototipos



El prototipo sirve como "el primer sistema". Lo que Brooks recomienda es desecharlo. Pero esto quizá sea un punto de vista idealizado. Aunque algunos prototipos se construyen para ser "desechables", otros son evolutivos; es decir, poco a poco se transforman en el sistema real.

Tanto a los participantes como a los ingenieros de software les gusta el paradigma de hacer prototipos. Los usuarios adquieren la sensación del sistema real, y los desarrolladores logran construir algo de inmediato. No obstante, hacer prototipos llega a ser problemático por las siguientes razones:

Resista la presión para convertir un prototipo burdo en un producto terminado. Como resultado de ello, casi siempre disminuye la calidad.

1. Los participantes ven lo que parece ser una versión funcional del software, sin darse cuenta de que el prototipo se obtuvo de manera caprichosa; no perciben que en la prisa por hacer que funcionara, usted no consideró la calidad general del software o la facilidad de darle mantenimiento a largo plazo. Cuando se les informa que el producto debe rehacerse a fin de obtener altos niveles de calidad, los participantes gritan que es usted un tonto y piden "unos cuantos arreglos" para hacer del prototipo un producto funcional. Con demasiada frecuencia, el gerente de desarrollo del software cede.
2. Como ingeniero de software, es frecuente que llegue a compromisos respecto de la implementación a fin de hacer que el prototipo funcione rápido. Quizá utilice un sistema operativo inapropiado, o un lenguaje de programación tan sólo porque cuenta con él y lo conoce; tal vez implementó un algoritmo ineficiente sólo para demostrar capacidad. Después de un tiempo, quizá se sienta cómodo con esas elecciones y olvide todas las razones por las que eran inadecuadas. La elección de algo menos que lo ideal ahora ha pasado a formar parte del sistema.

Aunque puede haber problemas, hacer prototipos es un paradigma eficaz para la ingeniería de software. La clave es definir desde el principio las reglas del juego; es decir, todos los participantes deben estar de acuerdo en que el prototipo sirva como el mecanismo para definir los requerimientos. Después se descartará (al menos en parte) y se hará la ingeniería del software real con la mirada puesta en la calidad.

CasaSegura

Selección de un modelo de proceso, parte 1

La escena: Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía (ficticia) que manufactura artículos de consumo para el hogar y para uso comercial.

Participantes: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; y Ed Robbins, miembro del equipo de software.

La conversación:

Lee: Recapitulemos. He dedicado algún tiempo al análisis de la línea de productos *CasaSegura*, según la vemos hasta el momento. No hay duda de que hemos efectuado mucho trabajo tan sólo para definir el concepto, pero me gustaría que ustedes comenzaran a pensar en cómo van a enfocar la parte del software de este proyecto.

Doug: Pareciera que en el pasado hemos estado muy desorganizados en nuestro enfoque del software.

Ed: No sé, Doug, siempre sacamos el producto.

Doug: Es cierto, pero no sin muchos sobresaltos, y este proyecto parece más grande y complejo que cualquier cosa que hayamos hecho antes.

Jamie: No parece tan mal, pero estoy de acuerdo… nuestro enfoque *ad hoc* de los proyectos anteriores no funcionará en éste, en particular si tenemos una fecha de entrega muy apretada.

Doug (sonríe): Quiero ser un poco más profesional en nuestro enfoque. La semana pasada asistí a un curso breve y aprendí mucho sobre ingeniería de software… algo bueno. Aquí necesitamos un proceso.

Jamie (con el ceño fruncido): Mi trabajo es producir programas de computadora, no papel.

Doug: Den una oportunidad antes de ser tan negativos conmigo. Lo que quiero decir es esto: [Doug pasa a describir la estructura del proceso vista en este capítulo y los modelos de proceso prescriptivo presentados hasta el momento.]

Doug: De cualquier forma, parece que un modelo lineal no es para nosotros… pues supone que conocemos todos los requerimientos y, conociendo esta empresa, eso no parece probable.

Vinod: Sí, y parece demasiado orientado a las tecnologías de información… tal vez sea bueno para hacer un sistema de control de inventarios o algo así, pero no parece bueno para *CasaSegura*.

Doug: Estoy de acuerdo.

Ed: Ese enfoque de hacer prototipos parece bueno. En todo caso, se asemeja mucho a lo que hacemos aquí.

Vinod: Eso es un problema. Me preocupa que no nos dé suficiente estructura.

Doug: No te preocupes. Tenemos muchas opciones más, y quisiera que ustedes, muchachos, elijan la que sea mejor para el equipo y para el proyecto.

El modelo espiral. Propuesto en primer lugar por Barry Boehm [Boe88], el *modelo espiral* es un modelo evolutivo del proceso del software y se acopla con la naturaleza iterativa de hacer prototipos con los aspectos controlados y sistémicos del modelo de cascada. Tiene el potencial para hacer un desarrollo rápido de versiones cada vez más completas. Boehm [Boe01a] describe el modelo del modo siguiente:

> El modelo de desarrollo espiral es un generador de *modelo de proceso* impulsado por el *riesgo*, que se usa para guiar la ingeniería concurrente con participantes múltiples de sistemas intensivos en software. Tiene dos características distintivas principales. La primera es el enfoque *cíclico* para el crecimiento incremental del grado de definición de un sistema y su implementación, mientras que disminuye su grado de riesgo. La otra es un conjunto de *puntos de referencia de anclaje puntual* para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

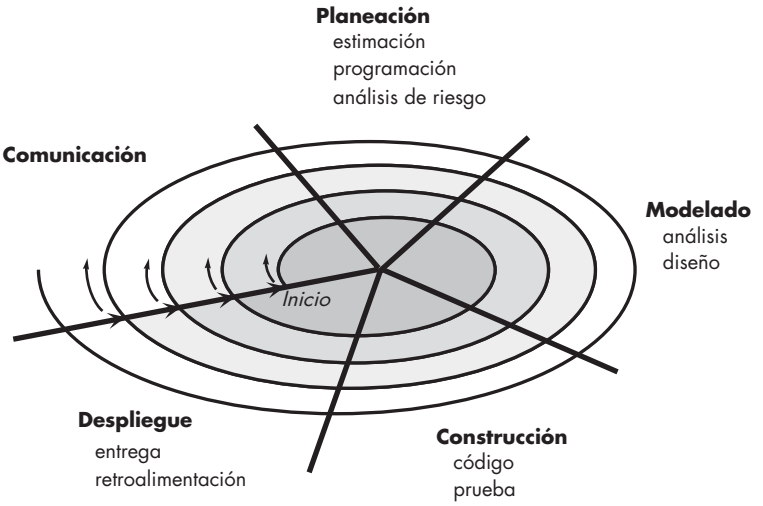
PUNTO CLAVE

El modelo en espiral se adapta para emplearse a lo largo de todo el ciclo de vida de una aplicación, desde el desarrollo del concepto hasta el mantenimiento.

Con el empleo del modelo espiral, el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del sistema cuya ingeniería se está haciendo.

Un modelo en espiral es dividido por el equipo de software en un conjunto de actividades estructurales. Para fines ilustrativos, se utilizan las actividades estructurales generales ya analizadas.[9] Cada una de ellas representa un segmento de la trayectoria espiral ilustrada en la figura 2.7. Al comenzar el proceso evolutivo, el equipo de software realiza actividades implícitas en un

FIGURA 2.7
Modelo de espiral común



9 El modelo espiral estudiado en esta sección es una variante del propuesto por Boehm. Para más información acerca del modelo espiral original, consulte [Boe88]. En [Boe98] se encuentra un análisis más reciente del modelo espiral del mismo autor.

circuito alrededor de la espiral en el sentido horario, partiendo del centro. El riesgo se considera conforme se desarrolla cada revolución (capítulo 28). En cada paso evolutivo se marcan *puntos de referencia puntuales*: combinación de productos del trabajo y condiciones que se encuentran a lo largo de la trayectoria de la espiral.

El primer circuito alrededor de la espiral da como resultado el desarrollo de una especificación del producto; las vueltas sucesivas se usan para desarrollar un prototipo y, luego, versiones cada vez más sofisticadas del software. Cada paso por la región de planeación da como resultado ajustes en el plan del proyecto. El costo y la programación de actividades se ajustan con base en la retroalimentación obtenida del cliente después de la entrega. Además, el gerente del proyecto ajusta el número planeado de iteraciones que se requieren para terminar el software.

WebRef

En la dirección **www.sei.cmu.edu/publications/documents/00.reports/00sr008.html** se encuentra información útil sobre el modelo espiral.

A diferencia de otros modelos del proceso que finalizan cuando se entrega el software, el modelo espiral puede adaptarse para aplicarse a lo largo de toda la vida del software de cómputo. Entonces, el primer circuito alrededor de la espiral quizá represente un "proyecto de desarrollo del concepto" que comienza en el centro de la espiral y continúa por iteraciones múltiples[10] hasta que queda terminado el desarrollo del concepto. Si el concepto va a desarrollarse en un producto real, el proceso sigue hacia fuera de la espiral y comienza un "proyecto de desarrollo de producto nuevo". El nuevo producto evolucionará a través de cierto número de iteraciones alrededor de la espiral. Más adelante puede usarse un circuito alrededor de la espiral para que represente un "proyecto de mejora del producto". En esencia, la espiral, cuando se caracteriza de este modo, sigue operativa hasta que el software se retira. Hay ocasiones en las que el proceso está inmóvil, pero siempre que se inicia un cambio comienza en el punto de entrada apropiado (por ejemplo, mejora del producto).

Si su administración pide un desarrollo apegado al presupuesto (mala idea, por lo general), la espiral se convierte en un problema. El costo se revisa y modifica cada vez que se termina un circuito.

El modelo espiral es un enfoque realista para el desarrollo de sistemas y de software a gran escala. Como el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor ante los riesgos en cada nivel de evolución. El modelo espiral usa los prototipos como mecanismo de reducción de riesgos, pero, más importante, permite aplicar el enfoque de hacer prototipos en cualquier etapa de la evolución del producto. Mantiene el enfoque de escalón sistemático sugerido por el ciclo de vida clásico, pero lo incorpora en una estructura iterativa que refleja al mundo real en una forma más realista. El modelo espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica de manera apropiada, debe reducir los riesgos antes de que se vuelvan un problema.

Cita:

"Sólo voy aquí y sólo el mañana me guía."

Dave Matthews Band

Pero, como otros paradigmas, el modelo espiral no es una panacea. Es difícil convencer a los clientes (en particular en situaciones bajo contrato) de que el enfoque evolutivo es controlable. Demanda mucha experiencia en la evaluación del riesgo y se basa en ella para llegar al éxito. No hay duda de que habrá problemas si algún riesgo importante no se descubre y administra.

2.3.4 Modelos concurrentes

Con frecuencia, el modelo concurrente es más apropiado para proyectos de ingeniería de productos en los que se involucran varios equipos de trabajo.

El *modelo de desarrollo concurrente*, en ocasiones llamado *ingeniería concurrente*, permite que un equipo de software represente elementos iterativos y concurrentes de cualquiera de los modelos de proceso descritos en este capítulo. Por ejemplo, la actividad de modelado definida para el modelo espiral se logra por medio de invocar una o más de las siguientes acciones de software: hacer prototipos, análisis y diseño.[11]

La figura 2.8 muestra la representación esquemática de una actividad de ingeniería de software dentro de la actividad de modelado con el uso del enfoque de modelado concurrente. La

10 Las flechas que apuntan hacia dentro a lo largo del eje que separa la región del **despliegue** de la de **comunicación** indican un potencial para la iteración local a lo largo de la misma trayectoria espiral.

11 Debe observarse que el análisis y diseño son tareas complejas que requieren mucho análisis. La parte 2 de este libro considera en detalle dichos temas.

CASASEGURA

Selección de un modelo de proceso, parte 2

La escena: Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía que manufactura productos de consumo para uso doméstico y comercial.

Participantes: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Vinod y Jamie, miembros del equipo de ingeniería de software.

La conversación: [Doug describe las opciones de proceso evolutivo.]

Jamie: Ahora me doy cuenta de algo. El enfoque incremental tiene sentido, y en verdad me gusta el flujo del modelo en espiral. Es bastante realista.

Vinod: De acuerdo. Entregamos un incremento, aprendemos de la retroalimentación del cliente, volvemos a planear y luego entregamos otro incremento. También se ajusta a la naturaleza del producto. Podemos lanzar con rapidez algo al mercado y luego agregar funcionalidad con cada versión, digo… con cada incremento.

Lee: Un momento. Doug, ¿dijiste que volveríamos a hacer el plan a cada vuelta de la espiral? Eso no es nada bueno; necesitamos un plan, un programa de actividades y apegarnos a ellos.

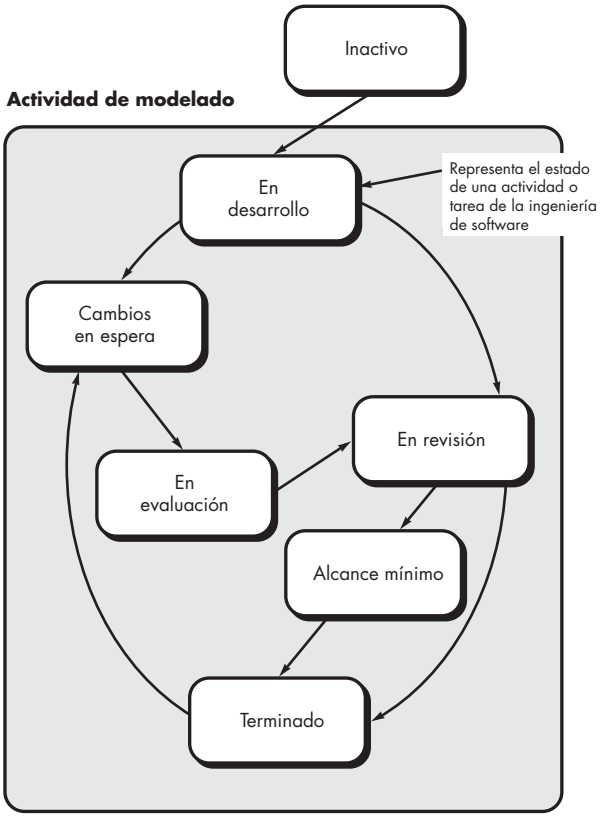
Doug: Ésa es la vieja escuela, Lee. Como dijeron los chicos, tenemos que hacerlo apegado a la realidad. Afirmo que es mejor afinar el plan a medida de que aprendamos más y conforme se soliciten cambios. Eso es más realista. ¿Qué sentido tiene un plan si no refleja la realidad?

Lee (con el ceño fruncido): Supongo, pero… a la alta dirección no le va a gustar… quieren un plan fijo.

Doug (sonriente): Entonces tendrás que reeducarlos, amigo.

actividad **—modelado—** puede estar en cualquiera de los estados[12] mencionados en un momento dado. En forma similar, es posible representar de manera análoga otras actividades, acciones o tareas (por ejemplo, **comunicación** o **construcción**). Todas las actividades de ingeniería de software existen de manera concurrente, pero se hallan en diferentes estados.

FIGURA 2.8
Un elemento del modelo de proceso concurrente



12 Un *estado* es algún modo de comportamiento observable externamente.

Por ejemplo, la actividad de comunicación (no se muestra en la figura) termina su primera iteración al principio de un proyecto y existe en el estado de **cambios en espera**. La actividad de modelado (que existía en estado **inactivo** mientras concluía la comunicación inicial, ahora hace una transición al estado **en desarrollo**. Sin embargo, si el cliente indica que deben hacerse cambios en los requerimientos, la actividad de modelado pasa del estado **en desarrollo** al de **cambios en espera**.

El modelado concurrente define una serie de eventos que desencadenan transiciones de un estado a otro para cada una de las actividades, acciones o tareas de la ingeniería de software. Por ejemplo, durante las primeras etapas del diseño (acción importante de la ingeniería de software que ocurre durante la actividad de modelado), no se detecta una inconsistencia en el modelo de requerimientos. Esto genera el evento *corrección del modelo de análisis*, que disparará la acción de análisis de requerimientos del estado **terminado** al de **cambios en espera**.

Cita:

"Todo proceso en su organización tiene un cliente, y un proceso sin cliente no tiene propósito."

V. Daniel Hunt

El modelado concurrente es aplicable a todos los tipos de desarrollo de software y proporciona un panorama apropiado del estado actual del proyecto. En lugar de confinar las actividades, acciones y tareas de la ingeniería de software a una secuencia de eventos, define una red del proceso. Cada actividad, acción o tarea de la red existe simultáneamente con otras actividades, acciones o tareas. Los eventos generados en cierto punto de la red del proceso desencadenan transiciones entre los estados.

2.3.5 Una última palabra acerca de los procesos evolutivos

Ya se dijo que el software de cómputo moderno se caracteriza por el cambio continuo, por tiempos de entrega muy apretados y por una necesidad apremiante de la satisfacción del cliente o usuario. En muchos casos, el tiempo para llegar al mercado es el requerimiento administrativo más importante. Si se pierde un nicho de mercado, todo el proyecto de software podría carecer de sentido.[13]

Los modelos de proceso evolutivo fueron concebidos para cumplir esos requisitos, pero, aun así, como clase general de modelos de proceso tienen demasiadas debilidades, que fueron resumidas por Nogueira y sus colegas [Nog00]:

> A pesar de los beneficios incuestionables de los procesos evolutivos de software, existen algunas preocupaciones. La primera es que hacer prototipos (y otros procesos evolutivos más sofisticados) plantea un problema para la planeación del proyecto debido a la incertidumbre en el número de ciclos que se requieren para elaborar el producto. La mayor parte de técnicas de administración y estimación de proyectos se basa en un planteamiento lineal de las actividades, por lo que no se ajustan por completo.
>
> En segundo lugar, los procesos evolutivos de software no establecen la velocidad máxima de la evolución. Si las evoluciones ocurren demasiado rápido, sin un periodo de relajamiento, es seguro que el proceso se volverá un caos. Por otro lado, si la velocidad es muy lenta, se verá perjudicada la productividad…
>
> En tercer lugar, los procesos de software deben centrarse en la flexibilidad y capacidad de extensión en lugar de en la alta calidad. Esto suena preocupante. Sin embargo, debe darse prioridad a la velocidad del desarrollo con el enfoque de cero defectos. Extender el desarrollo a fin de lograr alta calidad podría dar como resultado la entrega tardía del producto, cuando haya desaparecido el nicho de oportunidad. Este cambio de paradigma es impuesto por la competencia al borde del caos.

En realidad, sí parece preocupante un proceso del software que se centre en la flexibilidad, expansión y velocidad del desarrollo por encima de la calidad. No obstante, esta idea ha sido propuesta por varios expertos en ingeniería de software muy respetados ([You95], [Bac97]).

13 Sin embargo, es importante notar que ser el primero en llegar al mercado no es garantía de éxito. En realidad, muchos productos de software muy exitosos han llegado en segundo o hasta en tercer lugar al mercado (aprenden de los errores de sus antecesores).

El objetivo de los modelos evolutivos es desarrollar software de alta calidad[14] en forma iterativa o incremental. Sin embargo, es posible usar un proceso evolutivo para hacer énfasis en la flexibilidad, expansibilidad y velocidad del desarrollo. El reto para los equipos de software y sus administradores es establecer un balance apropiado entre estos parámetros críticos del proyecto y el producto, y la satisfacción del cliente (árbitro definitivo de la calidad del software).

2.4 MODELOS DE PROCESO ESPECIALIZADO

Los modelos de proceso especializado tienen muchas de las características de uno o más de los modelos tradicionales que se presentaron en las secciones anteriores. Sin embargo, dichos modelos tienden a aplicarse cuando se elige un enfoque de ingeniería de software especializado o definido muy específicamente.[15]

2.4.1 Desarrollo basado en componentes

WebRef

En la dirección **www.cbd-hq.com** hay información útil sobre el desarrollo basado en componentes.

Los componentes comerciales de software general (COTS, por sus siglas en inglés), desarrollados por vendedores que los ofrecen como productos, brindan una funcionalidad que se persigue con interfaces bien definidas que permiten que el componente se integre en el software que se va a construir. El *modelo de desarrollo basado en componentes* incorpora muchas de las características del modelo espiral. Es de naturaleza evolutiva [Nie92] y demanda un enfoque iterativo para la creación de software. Sin embargo, el modelo de desarrollo basado en componentes construye aplicaciones a partir de fragmentos de software prefabricados.

Las actividades de modelado y construcción comienzan con la identificación de candidatos de componentes. Éstos pueden diseñarse como módulos de software convencional o clases orientadas a objetos o paquetes[16] de clases. Sin importar la tecnología usada para crear los componentes, el modelo de desarrollo basado en componentes incorpora las etapas siguientes (se implementan con el uso de un enfoque evolutivo):

1. Se investigan y evalúan, para el tipo de aplicación de que se trate, productos disponibles basados en componentes.
2. Se consideran los aspectos de integración de los componentes.
3. Se diseña una arquitectura del software para que reciba los componentes.
4. Se integran los componentes en la arquitectura.
5. Se efectúan pruebas exhaustivas para asegurar la funcionalidad apropiada.

El modelo del desarrollo basado en componentes lleva a la reutilización del software, y eso da a los ingenieros de software varios beneficios en cuanto a la mensurabilidad. Si la reutilización de componentes se vuelve parte de la cultura, el equipo de ingeniería de software tiene la posibilidad tanto de reducir el ciclo de tiempo del desarrollo como el costo del proyecto. En el capítulo 10 se analiza con más detalle el desarrollo basado en componentes.

---

14 En este contexto, la calidad del software se define con mucha amplitud para que agrupe no sólo la satisfacción del cliente sino también varios criterios técnicos que se estudian en los capítulos 14 y 16.

15 En ciertos casos, los modelos de proceso especializado pueden caracterizarse mejor como un conjunto de técnicas o "metodología" para alcanzar una meta específica de desarrollo de software. No obstante, implican un proceso.

16 En el apéndice 2 se estudian los conceptos orientados a objetos, y se utilizan en toda la parte 2 del libro. En este contexto, una clase agrupa un conjunto de datos y los procedimientos para procesarlos. Un paquete de clases es un conjunto de clases relacionadas que funcionan juntas para alcanzar cierto resultado final.

2.4.2 El modelo de métodos formales

El *modelo de métodos formales* agrupa actividades que llevan a la especificación matemática formal del software de cómputo. Los métodos formales permiten especificar, desarrollar y verificar un sistema basado en computadora por medio del empleo de una notación matemática rigurosa. Ciertas organizaciones de desarrollo de software aplican una variante de este enfoque, que se denomina *ingeniería de software de quirófano* [Mil87, Dye92].

Cuando durante el desarrollo se usan métodos formales (capítulo 21), se obtiene un mecanismo para eliminar muchos de los problemas difíciles de vencer con otros paradigmas de la ingeniería de software. Lo ambiguo, incompleto e inconsistente se descubre y corrige con más facilidad, no a través de una revisión *ad hoc* sino con la aplicación de análisis matemático. Si durante el diseño se emplean métodos formales, éstos sirven como base para la verificación del programa, y así permiten descubrir y corregir errores que de otro modo no serían detectados.

Aunque el modelo de los métodos formales no es el más seguido, promete un software libre de defectos. Sin embargo, se han expresado preocupaciones acerca de su aplicabilidad en un ambiente de negocios:

? Si con los métodos formales puede demostrarse lo correcto de un software, ¿por qué no son ampliamente utilizados?

- El desarrollo de modelos formales consume mucho tiempo y es caro.
- Debido a que pocos desarrolladores de software tienen la formación necesaria para aplicar métodos formales, se requiere mucha capacitación.
- Es difícil utilizar los modelos como mecanismo de comunicación para clientes sin complejidad técnica.

A pesar de estas preocupaciones, el enfoque de los métodos formales ha ganado partidarios entre los desarrolladores que deben construir software de primera calidad en seguridad (por ejemplo, control electrónico de aeronaves y equipos médicos), y entre los desarrolladores que sufrirían graves pérdidas económicas si ocurrieran errores en su software.

2.4.3 Desarrollo de software orientado a aspectos

WebRef
Existen muchos recursos e información sobre SOA en la dirección: **aosd.net**

Sin importar el proceso del software que se elija, los constructores de software complejo implementan de manera invariable un conjunto de características, funciones y contenido de información localizados. Estas características localizadas del software se modelan como componentes (clases orientadas a objetos) y luego se construyen dentro del contexto de una arquitectura de sistemas. A medida que los sistemas modernos basados en computadora se hacen más sofisticados (y complejos), ciertas *preocupaciones* —propiedades que requiere el cliente o áreas de interés técnico— se extienden a toda la arquitectura. Algunas de ellas son las propiedades de alto nivel de un sistema (por ejemplo, seguridad y tolerancia a fallas). Otras afectan a funciones (aplicación de las reglas de negocios), mientras que otras más son sistémicas (sincronización de la tarea o administración de la memoria).

PUNTO CLAVE
El DSOA define "aspectos" que expresan preocupaciones del cliente que afectan múltiples funciones, características e información del sistema.

Cuando las preocupaciones afectan múltiples funciones, características e información del sistema, es frecuente que se les llame *preocupaciones globales*. Los *requerimientos del aspecto* definen aquellas preocupaciones globales que tienen algún efecto a través de la arquitectura del software. El *desarrollo de software orientado a aspectos* (DSOA), conocido también como *programación orientada a aspectos* (POA), es un paradigma de ingeniería de software relativamente nuevo que proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir *aspectos*: "mecanismos más allá de subrutinas y herencia para localizar la expresión de una preocupación global" [Elr01].

Grundy [Gru02] analiza con más profundidad los aspectos en el contexto de lo que denomina *ingeniería de componentes orientada a aspectos* (ICOA):

> La ICOA usa el concepto de rebanadas horizontales a través de componentes de software descompuestos verticalmente, llamados "aspectos", para caracterizar las propiedades globales funcionales y

no funcionales de los componentes. Los aspectos comunes y sistémicos incluyen interfaces de usuario, trabajo en colaboración, distribución, persistencia, administración de la memoria, procesamiento de las transacciones, seguridad, integridad, etc. Los componentes pueden proveer o requerir uno o más "detalles de aspectos" en relación con un aspecto particular, como un mecanismo de visión, alcance extensible y clase de interfaz (aspectos de la interfaz de usuario); generación de eventos, transporte y recepción (aspectos de distribución); almacenamiento, recuperación e indización de datos (aspectos de persistencia); autenticación, encriptación y derechos de acceso (aspectos de seguridad); descomposición de las transacciones, control de concurrencia y estrategia de registro (aspectos de las transacciones), entre otros. Cada detalle del aspecto tiene cierto número de propiedades relacionadas con las características funcionales o no del detalle del aspecto.

Aún no madura un proceso distinto orientado a aspectos. Sin embargo, es probable que un proceso así adopte características tanto de los modelos de proceso evolutivo como concurrente. El modelo evolutivo es apropiado en tanto los aspectos se identifican y después se construyen. La naturaleza paralela del desarrollo concurrente es esencial porque la ingeniería de aspectos se hace en forma independiente de los componentes de software localizados; aun así, los aspectos tienen un efecto directo sobre éstos. De esta forma, es esencial disponer de comunicación asincrónica entre las actividades de proceso del software aplicadas a la ingeniería, y la construcción de los aspectos y componentes.

El análisis detallado del desarrollo de software orientado al aspecto se deja a libros especializados en el tema. Si el lector tiene interés en profundizar, se le invita a consultar [Saf08], [Cla05], [Jac04] y [Gra03].

HERRAMIENTAS DE SOFTWARE

Administración del proceso

Objetivo: Ayudar a la definición, ejecución y administración de modelos de proceso prescriptivo.

Mecánica: Las herramientas de administración del proceso permiten que una organización o equipo de software defina un modelo completo del proceso (actividades estructurales, acciones, tareas, aseguramiento de la calidad, puntos de revisión, referencias y productos del trabajo). Además, las herramientas proporcionan un mapa conforme los ingenieros de software realizan el trabajo técnico, y una plantilla para los gerentes que deben dar seguimiento y controlar el proceso del software.

Herramientas representativas:[17]

GDPA, grupo de herramientas de investigación de definición del proceso, desarrollada por la Universidad de Bremen, en Alemania (**www.informatik.uni-bremen.de/uniform/gdpa/home.htm**), proporciona una amplia variedad de funciones para modelar y administrar procesos.

SpeeDev, desarrollada por SpeeDev Corporation (**www.speedev.com**), incluye un conjunto de herramientas para la definición del proceso, administración de los requerimientos, resolución de problemas, y planeación y seguimiento del proyecto.

ProVision BPMx, desarrollado por Proforma (**www.proformacorp.com**), es representativo de muchas herramientas que ayudan a definir el proceso y que automatizan el flujo del trabajo.

En la dirección **www.processwave.net/Links/tool_links.htm**, se encuentra una lista extensa de muchas herramientas diferentes asociadas con el proceso del software.

2.5 EL PROCESO UNIFICADO

En su libro fundamental, *Unified Process*, Ivar Jacobson, Grady Booch y James Rumbaugh [Jac99] analizan la necesidad de un proceso del software "impulsado por el caso de uso, centrado en la arquitectura, iterativo e incremental", con la afirmación siguiente:

17 Las herramientas mencionadas aquí no representan una obligación; sólo son una muestra de las de esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus desarrolladores respectivos.

> En la actualidad, la tendencia en el software es hacia sistemas más grandes y complejos. Eso se debe en parte al hecho de que año tras año las computadoras son más poderosas, lo que hace que los usuarios esperen más de ellas. Esta tendencia también se ha visto influida por el uso creciente de internet para intercambiar toda clase de información [...] Nuestro apetito por software cada vez más sofisticado aumenta conforme aprendemos, entre un lanzamiento y otro de un producto, cómo mejorar éste. Queremos software que se adapte mejor a nuestras necesidades, pero eso a su vez lo hace más complejo. En pocas palabras, queremos más.

En cierto modo, el proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software (véase el capítulo 3). El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso).[18] Hace énfasis en la importancia de la arquitectura del software y "ayuda a que el arquitecto se centre en las metas correctas, tales como que sea comprensible, permita cambios futuros y la reutilización" [Jac99]: Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.

2.5.1 Breve historia

Al principio de la década de 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] comenzaron a trabajar en un "método unificado" que combinaría lo mejor de cada uno de sus métodos individuales de análisis y diseño orientado a objetos. El resultado fue un UML, *lenguaje de modelado unificado*, que contiene una notación robusta para el modelado y desarrollo de los sistemas orientados a objetos.

El UML se utiliza en toda la parte 2 del libro para representar tanto los modelos de requerimientos como el diseño. En el apéndice 1 se presenta un método introductorio a la enseñanza para quienes no están familiarizados con las reglas básicas de notación y modelado con el UML. El estudio exhaustivo del UML se deja a libros dedicados al tema. En el apéndice 1 se enlistan los textos recomendables.

El UML brinda la tecnología necesaria para apoyar la práctica de la ingeniería de software orientada a objetos, pero no da la estructura del proceso que guíe a los equipos del proyecto cuando aplican la tecnología. En los siguientes años, Jacobson, Rumbaugh y Booch desarrollaron el *proceso unificado*, estructura para la ingeniería de software orientado a objetos que utiliza UML. Actualmente, el proceso unificado (PU) y el UML se usan mucho en proyectos de toda clase orientados a objetos. El modelo iterativo e incremental propuesto por el PU puede y debe adaptarse para que satisfaga necesidades específicas del proyecto.

2.5.2 Fases del proceso unificado[19]

PUNTO CLAVE

Las fases del PU tienen un objetivo similar al de las actividades estructurales generales definidas en este libro.

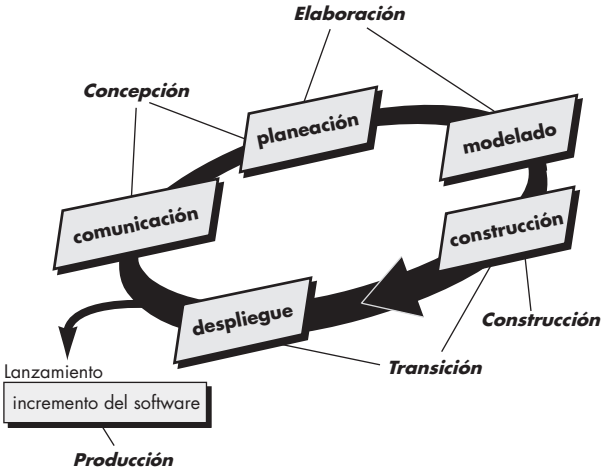
Al principio de este capítulo se estudiaron cinco actividades estructurales generales y se dijo que podían usarse para describir cualquier modelo de proceso del software. El proceso unificado no es la excepción. La figura 2.9 ilustra las "fases" del PU y las relaciona con las actividades generales estudiadas en el capítulo 1 y al inicio de éste.

La *fase de concepción* del PU agrupa actividades tanto de comunicación con el cliente como de planeación. Al colaborar con los participantes, se identifican los requerimientos del negocio,

18 El *caso de uso* (véase el capítulo 5) es la narración o plantilla que describe una función o rasgo de un sistema desde el punto de vista del usuario. Éste escribe un caso en uso que sirve como base para la creación de un modelo de requerimientos más completos.

19 El proceso unificado en ocasiones recibe el nombre de *Proceso Racional Unificado* (PRU), acuñado por Rational Corporation (adquirida posteriormente por IBM), que contribuyó desde el principio al desarrollo y mejora del PU y a la elaboración de ambientes completos (herramientas y tecnología) que apoyan el proceso.

FIGURA 2.9
El proceso unificado



se propone una arquitectura aproximada para el sistema y se desarrolla un plan para la naturaleza iterativa e incremental del proyecto en cuestión. Los requerimientos fundamentales del negocio se describen por medio de un conjunto de casos de uso preliminares (véase el capítulo 5) que detallan las características y funciones que desea cada clase principal de usuarios. En este punto, la arquitectura no es más que un lineamiento tentativo de subsistemas principales y la función y rasgos que tienen. La arquitectura se mejorará después y se expandirá en un conjunto de modelos que representarán distintos puntos de vista del sistema. La planeación identifica los recursos, evalúa los riesgos principales, define un programa de actividades y establece una base para las fases que se van a aplicar a medida que avanza el incremento del software.

La *fase de elaboración* incluye las actividades de comunicación y modelado del modelo general del proceso (véase la figura 2.9). La elaboración mejora y amplía los casos de uso preliminares desarrollados como parte de la fase de concepción y aumenta la representación de la arquitectura para incluir cinco puntos de vista distintos del software: los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue. En ciertos casos, la elaboración crea una "línea de base de la arquitectura ejecutable" [Arl02] que representa un sistema ejecutable de "primer corte".[20] La línea de base de la arquitectura demuestra la viabilidad de ésta, pero no proporciona todas las características y funciones que se requieren para usar el sistema. Además, al terminar la fase de elaboración se revisa con cuidado el plan a fin de asegurar que el alcance, riesgos y fechas de entrega siguen siendo razonables. Es frecuente que en este momento se hagan modificaciones al plan.

WebRef

En la dirección **www.ambysoft.com/unifiedprocess/agileUP.html**, se encuentra un análisis interesante del PU en el contexto del desarrollo ágil.

La *fase de construcción* del PU es idéntica a la actividad de construcción definida para el proceso general del software. Con el uso del modelo de arquitectura como entrada, la fase de construcción desarrolla o adquiere los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Para lograrlo, se completan los modelos de requerimientos y diseño que se comenzaron durante la fase de elaboración, a fin de que reflejen la versión final del incremento de software. Después se implementan en código fuente todas las características y funciones necesarias para el incremento de software (por ejemplo, el lanzamiento). A medida de que se implementan los componentes, se diseñan y efectúan pruebas unitarias[21] para cada uno. Además, se realizan actividades de integración (ensamble de compo-

20 Es importante darse cuenta de que la línea de base de la arquitectura no es un prototipo y que no se desecha. Por el contrario, es revestida durante la fase siguiente del PU.

21 En los capítulos 17 a 20 se presenta el análisis exhaustivo de las pruebas del software (incluso las *pruebas unitarias*).

nentes y pruebas de integración). Se emplean casos de uso para obtener un grupo de pruebas de aceptación que se ejecutan antes de comenzar la siguiente fase del PU.

La *fase de transición* del PU incluye las últimas etapas de la actividad general de construcción y la primera parte de la actividad de despliegue general (entrega y retroalimentación). Se da el software a los usuarios finales para las pruebas beta, quienes reportan tanto los defectos como los cambios necesarios. Además, el equipo de software genera la información de apoyo necesaria (por ejemplo, manuales de usuario, guías de solución de problemas, procedimientos de instalación, etc.) que se requiere para el lanzamiento. Al finalizar la fase de transición, el software incrementado se convierte en un producto utilizable que se lanza.

La *fase de producción* del PU coincide con la actividad de despliegue del proceso general. Durante esta fase, se vigila el uso que se da al software, se brinda apoyo para el ambiente de operación (infraestructura) y se reportan defectos y solicitudes de cambio para su evaluación.

Es probable que al mismo tiempo que se llevan a cabo las fases de construcción, transición y producción, comience el trabajo sobre el siguiente incremento del software. Esto significa que las cinco fases del PU no ocurren en secuencia sino que concurren en forma escalonada.

El flujo de trabajo de la ingeniería de software está distribuido a través de todas las fases del PU. En el contexto de éste, un *flujo de trabajo* es análogo al conjunto de tareas (que ya se describió en este capítulo). Es decir, un flujo de trabajo identifica las tareas necesarias para completar una acción importante de la ingeniería de software y los productos de trabajo que se generan como consecuencia de la terminación exitosa de aquéllas. Debe notarse que no toda tarea identificada para el flujo de trabajo del PU es realizada en todos los proyectos de software. El equipo adapta el proceso (acciones, tareas, subtareas y productos del trabajo) a fin de que cumpla sus necesidades.

2.6 Modelos del proceso personal y del equipo

El mejor proceso del software es el que está cerca de las personas que harán el trabajo. Si un modelo del proceso del software se ha desarrollado en un nivel corporativo u organizacional, será eficaz sólo si acepta una adaptación significativa para que cubra las necesidades del equipo de proyecto que en realidad hace el trabajo de ingeniería de software. En la situación ideal se crearía un proceso que se ajustara del mejor modo a los requerimientos, y al mismo tiempo cubriera las más amplias necesidades del equipo y de la organización. En forma alternativa, el equipo crearía un proceso propio que satisficiera las necesidades más estrechas de los individuos y las más generales de la organización. Watts Humphrey ([Hum97] y [Hum00]) afirma que es posible crear un "proceso personal de software" y/o un "proceso del equipo de software". Ambos requieren trabajo duro, capacitación y coordinación, pero los dos son asequibles.[22]

Cita:

"La persona que es exitosa tan sólo se ha hecho el hábito de hacer las cosas que no hacen las personas que no tienen éxito."

Dexter Yager

2.6.1 Proceso personal del software (PPS)

Todo desarrollador utiliza algún proceso para elaborar software de cómputo. El proceso puede ser caprichoso o *ad hoc*; quizá cambie a diario; tal vez no sea eficiente, eficaz o incluso no sirva; pero sí existe un "proceso". Watts Humphrey [Hum97] sugiere que a fin de cambiar un proceso personal ineficaz, un individuo debe pasar por las cuatro fases, cada una de las cuales requiere capacitación e instrumentación cuidadosa. El *proceso personal del software* (PPS) pone el énfasis en la medición personal tanto del producto del trabajo que se genera como de su calidad. Además, el PPS responsabiliza al profesional acerca de la planeación del proyecto (por ejemplo,

WebRef

En la dirección **www.ipd.uka.de/PSP**, se hallan muchos recursos para el PPS.

22 Es útil notar que quienes proponen un desarrollo ágil del software (véase el capítulo 3) también plantean que el proceso debe ser cercano al equipo. Para lograr esto sugieren un método alternativo.

estimación y programación de actividades) y delega en el practicante el poder de controlar la calidad de todos los productos del trabajo de software que se desarrollen. El modelo del PPS define cinco actividades estructurales:

Planeación. Esta actividad aísla los requerimientos y desarrolla las estimaciones tanto del tamaño como de los recursos. Además, realiza la estimación de los defectos (el número de defectos proyectados para el trabajo). Todas las mediciones se registran en hojas de trabajo o plantillas. Por último, se identifican las tareas de desarrollo y se crea un programa para el proyecto.

¿Qué actividades estructurales se usan durante el PPS?

Diseño de alto nivel. Se desarrollan las especificaciones externas para cada componente que se va a construir y se crea el diseño de componentes. Si hay incertidumbre, se elaboran prototipos. Se registran todos los aspectos relevantes y se les da seguimiento.

Revisión del diseño de alto nivel. Se aplican métodos de verificación formal (véase el capítulo 21) para descubrir errores en el diseño. Se mantienen las mediciones para todas las tareas y resultados del trabajo importantes.

Desarrollo. Se mejora y revisa el diseño del componente. El código se genera, revisa, compila y prueba. Las mediciones se mantienen para todas las tareas y resultados de trabajo de importancia.

Post mórtem. Se determina la eficacia del proceso por medio de medidas y mediciones obtenidas (ésta es una cantidad sustancial de datos que deben analizarse con métodos estadísticos). Las medidas y mediciones deben dar la guía para modificar el proceso a fin de mejorar su eficacia.

El PPS enfatiza la necesidad de detectar pronto los errores; de igual importancia es entender los tipos de ellos que es probable cometer. Esto se logra a través de una actividad de evaluación rigurosa ejecutada para todos los productos del trabajo que se generen.

PUNTO CLAVE

El PPS pone el énfasis en la necesidad de registrar y analizar los tipos de errores que se cometen, de modo que se desarrollen estrategias para eliminarlos.

El PPS representa un enfoque disciplinado basado en la medición para la ingeniería de software que quizá sea un choque cultural para muchos de sus practicantes. Sin embargo, cuando se introduce el PPS en forma apropiada en los ingenieros de software [Hum96], es significativa la mejora resultante en la productividad de la ingeniería respectiva y en la calidad del software [Fer97]. No obstante, el PPS no ha sido adoptado con amplitud por la industria. Es triste reconocer que las razones de esto tienen que ver más con la naturaleza humana y la inercia organizacional que con las fortalezas y debilidades del enfoque del PPS. Dicho enfoque plantea desafíos intelectuales y demanda un nivel de compromiso (por parte de los practicantes y sus administradores) que no siempre es posible obtener. La capacitación es relativamente larga y sus costos elevados. El nivel requerido de las mediciones es culturalmente difícil para muchas personas de la comunidad del software.

¿Es posible usar el PPS como un proceso eficaz de software a nivel personal? La respuesta es un rotundo "sí". Pero aun si no se adoptara por completo el PPS, muchos de los conceptos del proceso de mejora personal que introduce constituyen un aprendizaje provechoso.

2.6.2 Proceso del equipo de software (PES)

WebRef

En la dirección **www.sei.cmu.edu/tsp/**, hay información sobre la formación de equipos de alto rendimiento que usan PES y PPS.

Debido a que muchos proyectos de software industrial son elaborados por un equipo de profesionales, Watts Humphrey extendió las lecciones aprendidas de la introducción del PPS y propuso un *proceso del equipo de software* (PES). El objetivo de éste es construir un equipo "autodirigido" para el proyecto, que se organice para producir software de alta calidad. Humphrey [Hum98] define los objetivos siguientes para el PES:

- Formar equipos autodirigidos que planeen y den seguimiento a su trabajo, que establezcan metas y que sean dueños de sus procesos y planes. Éstos pueden ser equipos de software puros o de productos integrados (EPI) constituidos por 3 a 20 ingenieros.

- Mostrar a los gerentes cómo dirigir y motivar a sus equipos y cómo ayudarlos a mantener un rendimiento máximo.
- Acelerar la mejora del proceso del software, haciendo del modelo de madurez de la capacidad, CMM,[23] nivel 5, el comportamiento normal y esperado.
- Brindar a las organizaciones muy maduras una guía para la mejora.
- Facilitar la enseñanza universitaria de aptitudes de equipo con grado industrial.

Para formar un equipo autodirigido, usted debe colaborar bien en lo interno y comunicarse bien en lo externo.

Un equipo autodirigido tiene la comprensión consistente de sus metas y objetivos generales; define el papel y responsabilidad de cada miembro del equipo; da seguimiento cuantitativo a los datos del proyecto (sobre la productividad y calidad); identifica un proceso de equipo que sea apropiado para el proyecto y una estrategia para implementarlo; define estándares locales aplicables al trabajo de ingeniería de software del equipo; evalúa en forma continua el riesgo y reacciona en consecuencia; y da seguimiento, administra y reporta el estado del proyecto.

El PES define las siguientes actividades estructurales: **inicio del proyecto, diseño de alto nivel, implementación, integración y pruebas,** y **post mórtem.** Como sus contrapartes del PPS (observe que la terminología es algo diferente), estas actividades permiten que el equipo planee, diseñe y construya software en forma disciplinada, al mismo tiempo que mide cuantitativamente el proceso y el producto. La etapa post mórtem es el escenario de las mejoras del proceso.

El PES utiliza una variedad amplia de *scripts*, formatos y estándares que guían a los miembros del equipo en su trabajo. Los *scripts* definen actividades específicas del proceso (por ejemplo, inicio del proyecto, diseño, implementación, integración y pruebas del sistema, y post mórtem), así como otras funciones más detalladas del trabajo (planeación del desarrollo, desarrollo de requerimientos, administración de la configuración del software y prueba unitaria) que forman parte del proceso de equipo.

Los scripts del PES definen elementos del proceso del equipo y de las actividades que ocurren dentro del proceso.

El PES reconoce que los mejores equipos de software son los autodirigidos.[24] Los miembros del equipo establecen los objetivos del proyecto, adaptan el proceso para que cubra las necesidades, controlan la programación de actividades del proyecto y, con la medida y análisis de las mediciones efectuadas, trabajan de manera continua en la mejora del enfoque de ingeniería de software que tiene el equipo.

Igual que el PPS, el PES es un enfoque riguroso para la ingeniería de software y proporciona beneficios distintivos y cuantificables en productividad y calidad. El equipo debe tener un compromiso total con el proceso y recibir capacitación completa para asegurar que el enfoque se aplique en forma apropiada.

2.7 Tecnología del proceso

El equipo del software debe adaptar uno o más de los modelos del proceso estudiados en las secciones precedentes. Para ello, se han desarrollado *herramientas de tecnología del proceso* que ayudan a las organizaciones de software a analizar su proceso actual, organizar las tareas de trabajo, controlar y vigilar el avance, y administrar la calidad técnica.

Las herramientas de tecnología del proceso permiten que una organización de software construya un modelo automatizado de la estructura del proceso, conjuntos de tareas y actividades sombrilla, estudiados en la sección 2.1. El modelo, que normalmente se representa como

23 El modelo de madurez de la capacidad (CMM), que es una medida de la eficacia de un proceso del software, se estudia en el capítulo 30.

24 En el capítulo 31 se analiza la importancia de los equipos "autoorganizados" como elemento clave del desarrollo ágil del software.

una red, se analiza para determinar el flujo de trabajo normal y se examinan estructuras alternativas del proceso que podrían llevar a disminuir el tiempo o costo del desarrollo.

Una vez creado un proceso aceptable, se emplean otras herramientas de tecnología para asignar, vigilar e incluso controlar todas las actividades, acciones y tareas de la ingeniería de software definidas como parte del modelo del proceso. Cada miembro de un equipo de software utiliza dichas herramientas para desarrollar una lista de verificación de las tareas de trabajo que deben realizarse. La herramienta de tecnología del proceso también se usa para coordinar el empleo de otras herramientas de la ingeniería de software que sean apropiadas para una tarea particular del trabajo.

HERRAMIENTAS DE SOFTWARE

Herramientas de modelado del proceso

Objetivo: Si una organización trabaja para mejorar un proceso (o software) de negocios, primero debe entenderlo. Las herramientas de modelado del proceso (también llamadas herramientas de *tecnología del proceso* o *de administración del proceso*) se usan para representar los elementos clave de un proceso, de modo que se entienda mejor. Dichas herramientas también se relacionan con descripciones del proceso que ayudan a los involucrados a entender las acciones y tareas del trabajo que se requieren para llevarlo a cabo. Las herramientas de modelado del proceso tienen vínculos con otras que dan apoyo a las actividades del proceso definido.

Mecánica: Las herramientas en esta categoría permiten que un equipo defina los elementos de un modelo de proceso único (acciones, tareas, productos del trabajo, puntos de aseguramiento de la calidad, etc.), dan una guía detallada acerca del contenido o descripción de cada elemento del proceso, y después administran el proceso conforme se realiza. En ciertos casos, las herramientas de tecnología del proceso incorporan tareas estándar de administración de proyectos, tales como estimación, programación, seguimiento y control.

Herramientas representativas:[25]

Igrafx Process Tools: herramientas que permiten que un equipo mapee, mida y modele el proceso del software (**www.micrografx.com**)

Adeptia BPM Server: diseñado para administrar, automatizar y optimizar procesos de negocios (**www.adptia.com**)

SpeedDev Suite: conjunto de seis herramientas con mucho énfasis en las actividades de administración de la comunicación y modelado (**www.speedev.com**)

2.8 PRODUCTO Y PROCESO

Si el proceso es deficiente, no cabe duda de que el producto final sufrirá. Pero también es peligrosa la dependencia excesiva del proceso. En un ensayo corto escrito hace muchos años, Margaret Davis [Dav95a] hace comentarios atemporales sobre la dualidad del producto y del proceso:

> Cada diez años, más o menos, la comunidad del software redefine "el problema" por medio de cambiar su atención de aspectos del producto a aspectos del proceso. Así, hemos adoptado lenguajes de programación estructurada (producto) seguidos de métodos de análisis estructurados (proceso) que van seguidos por el encapsulamiento de datos (producto) a los que siguieron el énfasis actual en el modelo de madurez de la capacidad, del Instituto de Ingeniería de Software para el Desarrollo de Software (proceso) (seguido por métodos orientados a objetos, a los que sigue el desarrollo ágil de software).
>
> En tanto que la tendencia natural de un péndulo es alcanzar el estado de reposo en el punto medio entre dos extremos, la atención de la comunidad del software cambia constantemente porque se aplica una nueva fuerza al fallar la última oscilación. Estos vaivenes son dañinos en sí mismos porque confunden al profesional promedio del software al cambiar en forma radical lo que significa hacer el trabajo bien. Los cambios periódicos no resuelven "el problema" porque están predestinados a fallar toda vez que el producto y el proceso son tratados como si fueran una dicotomía en lugar de una dualidad.

25 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

> En la comunidad científica existe el precedente de adoptar nociones de dualidad cuando las contradicciones en las observaciones no pueden ser explicadas por alguna teoría alternativa. La naturaleza dual de la luz, que parece ser al mismo tiempo onda y partícula, ha sido aceptada desde la década de 1920, cuando la propuso Louis de Broglie. Pienso que las observaciones que podemos hacer sobre el conjunto del software y su desarrollo demuestran una dualidad fundamental entre el producto y el proceso. Nunca es posible derivar u obtener todo el conjunto, su contexto, uso, significado y beneficios si se le ve sólo como proceso o sólo como producto...
>
> Toda la actividad humana es un proceso, pero cada uno de nosotros obtiene un sentido de beneficio propio gracias a aquellas actividades que dan como resultado una representación o instancia que puede usar o apreciar más de una persona, utilizarla una y otra vez, o emplearla en algún otro contexto no considerado. Es decir, obtenemos sentimientos de satisfacción por la reutilización de nuestros productos, ya sea que lo hagamos nosotros u otras personas.
>
> Entonces, si bien la rápida asimilación de las metas de reutilización en el desarrollo del software incrementa potencialmente la satisfacción que obtienen los profesionales del software en su trabajo, también aumenta la urgencia de la aceptación de la dualidad de producto y proceso. Pensar en un artefacto reutilizable como si fuera sólo un producto o sólo un proceso oscurece el contexto y las formas de emplearlo, o bien oculta el hecho de que cada uso da como resultado un producto que a su vez será utilizado como entrada para alguna otra actividad de desarrollo de software. Privilegiar un punto de vista sobre el otro reduce mucho las oportunidades para la reutilización y, por tanto, se pierde la oportunidad de aumentar la satisfacción por el trabajo.

La gente obtiene tanta (o más) satisfacción del proceso creativo como del producto final. Un artista disfruta las pinceladas tanto como el resultado que enmarca. Un escritor goza de la búsqueda de la metáfora apropiada tanto como del libro terminado. Como profesional creativo del software, usted también debe obtener tanta satisfacción del proceso como del producto final. La dualidad de producto y proceso es un elemento importante para hacer que personas creativas se involucren conforme la ingeniería de software evoluciona.

2.9 Resumen

Un modelo general del proceso para la ingeniería de software incluye un conjunto de actividades estructurales y sombrilla, acciones y tareas de trabajo. Cada uno de los modelos de proceso puede describirse por un flujo distinto del proceso: descripción de cómo se organizan secuencial y cronológicamente las actividades estructurales, acciones y tareas. Los patrones del proceso pueden utilizarse para resolver los problemas comunes que surgen como parte del proceso del software.

Los modelos de proceso prescriptivo se han aplicado durante muchos años en un esfuerzo por introducir orden y estructura al desarrollo de software. Cada uno de dichos modelos sugiere un flujo de proceso algo distinto, pero todos llevan a cabo el mismo conjunto de actividades estructurales generales: comunicación, planeación, modelado, construcción y desarrollo.

Los modelos de proceso secuencial, como el de la cascada y en V, son los paradigmas más antiguos del software. Sugieren un flujo lineal del proceso que con frecuencia no es congruente con las realidades modernas (cambio continuo, sistemas en evolución, plazos ajustados, etc.) del mundo del software. Sin embargo, tienen aplicación en situaciones en las que los requerimientos están bien definidos y son estables.

Los modelos de proceso incremental son de naturaleza iterativa y producen con mucha rapidez versiones funcionales del software. Los modelos de proceso evolutivo reconocen la naturaleza iterativa e incremental de la mayoría de proyectos de ingeniería de software y están diseñados para aceptar los cambios. Los modelos evolutivos, tales como el de hacer prototipos y el espiral, generan rápido productos de trabajo incremental (o versiones funcionales del software). Estos modelos se adoptan para aplicarse a lo largo de todas las actividades de la inge-

niería de software, desde el desarrollo del concepto hasta el mantenimiento del sistema a largo plazo.

El modelo de proceso concurrente permite que un equipo de software represente los elementos iterativos y concurrentes de cualquier modelo de proceso. Los modelos especializados incluyen el basado en componentes, que pone el énfasis en la reutilización y ensamble de los componentes; el modelo de métodos formales consiste en un enfoque basado en matemáticas para desarrollar y verificar el software; y el modelo orientado a aspectos implica preocupaciones globales que afectan toda la arquitectura del sistema. El proceso unificado es un proceso del software diseñado como estructura para los métodos y herramientas del UML, y está "impulsado por el caso de uso, centrado en la arquitectura, y es iterativo e incremental".

Se han propuesto modelos personal y del equipo para el proceso del software. Ambos enfatizan la medición, planeación y autodirección como los ingredientes clave para un proceso exitoso del software.

PROBLEMAS Y PUNTOS POR EVALUAR

2.1. En la introducción de este capítulo, Baetjer afirma que: "El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes [tecnología]." Enliste cinco preguntas que *a*) los diseñadores deben responder a los usuarios, *b*) los usuarios deben plantear a los diseñadores, *c*) los usuarios deben hacerse a sí mismos sobre el producto de software que ha de elaborarse, *d*) los diseñadores deben plantearse acerca del producto de software que va a construirse y del proceso que se usará para ello.

2.2. Trate de desarrollar un conjunto de acciones para la actividad de comunicación. Seleccione una acción y defina un conjunto de tareas para ella.

2.3. Un problema común durante la **comunicación** ocurre cuando se encuentra a dos participantes que tienen ideas en conflicto sobre lo que debe ser el software, es decir, que tienen requerimientos mutuamente conflictivos. Desarrolle un patrón del proceso (esto sería un patrón de la etapa) con el empleo de la plantilla presentada en la sección 2.1.3 que aborda este problema y sugiera un enfoque eficaz para él.

2.4. Investigue un poco sobre el PPS y haga una breve presentación que describa los tipos de mediciones que se pide hacer a un ingeniero individual de software y la forma en la que pueden usarse para mejorar la eficacia personal.

2.5. El uso de scripts (mecanismo requerido en el PES) no es apreciado de manera universal en la comunidad del software. Haga una lista de pros y contras en relación con los scripts y sugiera al menos dos situaciones en las que serían útiles, y otras dos en las que generarían menos beneficios.

2.6. Lea a [Nog00] y escriba un ensayo de dos o tres páginas donde analice el efecto que tiene el "caos" en la ingeniería de software.

2.7. Dé tres ejemplos de proyectos de software que podrían efectuarse con el modelo de cascada. Sea específico.

2.8. Proporcione tres ejemplos de proyectos de software que podrían abordarse con el modelo de hacer prototipos. Sea específico.

2.9. ¿Qué adaptaciones del proceso se requerirían si el proyecto evolucionara en un sistema o producto que se entregase?

2.10. Diga tres ejemplos de proyectos de software que podrían realizarse con el modelo incremental. Sea específico.

2.11. Conforme avanza hacia fuera por el flujo de proceso en espiral, ¿qué puede decirse sobre el software que se está desarrollando o que está en mantenimiento?

2.12. ¿Es posible combinar modelos de proceso? Si es así, diga un ejemplo.

2.13. El modelo de proceso concurrente define un conjunto de "estados". Describa con sus propias palabras qué es lo que representan, y después indique cómo entran en juego dentro del modelo de proceso concurrente.

2.14. ¿Cuáles son las ventajas y desventajas de desarrollar software en el que la calidad no es "suficientemente buena"? Es decir, ¿qué pasa cuando se pone el énfasis en la velocidad de desarrollo sobre la calidad del producto?

2.15. Dé tres ejemplos de proyectos de software que serían abordables con el modelo basado en componentes. Sea específico.

2.16. ¿Es posible demostrar que un componente de software, o incluso un programa completo, es correcto? Entonces, ¿por qué no todos lo hacen?

2.17. ¿Son lo mismo el proceso unificado y el UML? Explique su respuesta.

Lecturas adicionales y fuentes de información

La mayor parte de los libros de ingeniería de software consideran en detalle los modelos de proceso tradicionales. Libros como el de Sommerville (*Software Engineering*, 8a. ed., Addison-Wesley, 2006), Pfleeger y Atlee (*Software Engineering*, 3a. ed., Prentice-Hall, 2005), y Schach (*Object-Oriented and Classical Software Engineering*, 7a. ed., McGraw-Hill, 2006) consideran los paradigmas tradicionales y estudian sus fortalezas y debilidades. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) da un punto de vista pragmático y crudo del proceso de ingeniería de software. Aunque no se dedica específicamente al proceso, Brooks (*The Mythical Man-Month*, 2a. ed., Addison-Wesley, 1995) presenta la sabiduría antigua sobre los proyectos y plantea que todo tiene que ver con el proceso.

Firesmith y Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) presenta una plantilla general para crear "procesos de software flexibles pero con disciplina" y analiza los atributos y objetivos del proceso. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) estudia técnicas de modelado que permiten analizar los elementos técnicos y sociales interrelacionados del proceso del software. Sharpe y McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) presentan herramientas para modelar procesos tanto de software como de negocios.

Lim (*Managing Software Reuse*, Prentice-Hall, 2004) estudia la reutilización desde la perspectiva del gerente. Ezran, Morisio y Tully (*Practical Software Reuse*, Springer, 2002) y Jacobson, Griss y Jonsson (*Software Reuse*, Addison-Wesley, 1997) presentan mucha información útil sobre el desarrollo basado en componentes. Heineman y Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describen el proceso requerido para implementar sistemas basados en componentes. Kenett y Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) analizan la manera en la que se conectan íntimamente la administración de la calidad y el diseño del proceso.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007) y Richardson y Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) presentan una amplia colección de lineamientos útiles aplicables a la actividad de despliegue.

Además del libro fundamental de Jacobson, Rumbaugh y Booch acerca del proceso unificado [Jac99], los libros de Arlow y Neustadt (*UML 2 and the Unified Process,* Addison-Wesley, 2005), Kroll y Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) y Farve (*UML and the Unified Process*, IRM Press, 2003) proveen información complementaria excelente. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) analiza la administración de proyectos dentro del contexto del PU.

En internet existe una amplia variedad de fuentes de información sobre la ingeniería de software y el proceso del software. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso del software.

CAPÍTULO

3

DESARROLLO ÁGIL

CONCEPTOS CLAVE

En 2001, Kent Beck y otros 16 notables desarrolladores de software, escritores y consultores [Bec01a] (grupo conocido como la "Alianza Ágil") firmaron el "Manifiesto por el desarrollo ágil de software". En él se establecía lo siguiente:

> Estamos descubriendo formas mejores de desarrollar software, por medio de hacerlo y de dar ayuda a otros para que lo hagan. Ese trabajo nos ha hecho valorar:
>
> *Los individuos y sus interacciones*, sobre los procesos y las herramientas
>
> *El software que funciona*, más que la documentación exhaustiva
>
> *La colaboración con el cliente*, y no tanto la negociación del contrato
>
> *Responder al cambio*, mejor que apegarse a un plan
>
> Es decir, si bien son valiosos los conceptos que aparecen en segundo lugar, valoramos más los que aparecen en primer sitio.

Un manifiesto normalmente se asocia con un movimiento político emergente: ataca a la vieja guardia y sugiere un cambio revolucionario (se espera que para mejorar). En cierta forma, de eso es de lo que trata el desarrollo ágil.

Aunque las ideas subyacentes que lo guían han estado durante muchos años entre nosotros, ha sido en menos de dos décadas que cristalizaron en un "movimiento". Los métodos ágiles[1] se desarrollaron como un esfuerzo por superar las debilidades reales y percibidas de la ingeniería de software convencional. El desarrollo ágil proporciona beneficios importantes, pero no es

UNA MIRADA RÁPIDA

¿Qué es? La ingeniería de software ágil combina una filosofía con un conjunto de lineamientos de desarrollo. La filosofía pone el énfasis en: la satisfacción del cliente y en la entrega rápida de software incremental, los equipos pequeños y muy motivados para efectuar el proyecto, los métodos informales, los productos del trabajo con mínima ingeniería de software y la sencillez general en el desarrollo. Los lineamientos de desarrollo enfatizan la entrega sobre el análisis y el diseño (aunque estas actividades no se desalientan) y la comunicación activa y continua entre desarrolladores y clientes.

¿Quién lo hace? Los ingenieros de software y otros participantes en el proyecto (gerentes, clientes, usuarios finales, etc.) trabajan juntos en un proyecto ágil, formando un equipo con organización propia y que controla su propio destino. Un equipo ágil facilita la comunicación y colaboración entre aquellos a quienes sirve.

¿Por qué es importante? El ambiente moderno de negocios que genera sistemas basados en computadora y productos de software evoluciona rápida y constantemente. La ingeniería de software ágil representa una alternativa razonable a la ingeniería de software convencional para ciertas clases de software y en algunos tipos de proyectos. Asimismo, se ha demostrado que concluye con rapidez sistemas exitosos.

¿Cuáles son los pasos? Un nombre más apropiado para el desarrollo ágil sería "ingeniería de software ligero". Permanecen las actividades estructurales fundamentales: comunicación, planeación, modelado, construcción y despliegue. Pero se transforman en un conjunto mínimo de tareas que lleva al equipo del proyecto hacia la construcción y entrega (algunas personas dirían que esto se hace a costa del análisis del problema y del diseño de la solución).

¿Cuál es el producto final? Tanto el cliente como el ingeniero de software tienen la misma perspectiva: el único producto del trabajo realmente importante es un "incremento de software" operativo que se entrega al cliente exactamente en la fecha acordada.

¿Cómo me aseguro de que lo hice bien? El trabajo estará bien hecho si el equipo ágil concuerda en que el proceso funciona y en que produce incrementos de software utilizables que satisfagan al cliente.

1 En ocasiones se conoce a los métodos ágiles como *métodos ligeros* o *métodos esbeltos*.

aplicable a todos los proyectos, productos, personas y situaciones. *No* es la antítesis de la práctica de la ingeniería de software sólida y puede aplicarse como filosofía general para todo el trabajo de software.

Es frecuente que en la economía moderna sea difícil o imposible predecir la forma en la que evolucionará un sistema basado en computadora (por ejemplo, una aplicación con base en web). Las condiciones del mercado cambian con rapidez, las necesidades de los usuarios finales se transforman y emergen nuevas amenazas competitivas sin previo aviso. En muchas situaciones no será posible definir los requerimientos por completo antes de que el proyecto comience. Se debe ser suficientemente ágil para responder a lo fluido que se presenta el ambiente de negocios.

La fluidez implica cambio, y el cambio es caro, en particular si es descontrolado o si se administra mal. Una de las características más atractivas del enfoque ágil es su capacidad de reducir los costos del cambio durante el proceso del software.

¿Significa esto que el reconocimiento de los retos planteados por las realidades modernas hace que sean descartables los valiosos principios, conceptos, métodos y herramientas de la ingeniería del software? No, en absoluto… Igual que todas las disciplinas de la ingeniería, la del software evoluciona en forma continua. Puede adaptarse con facilidad para que satisfaga los desafíos que surgen de la demanda de agilidad.

Cita:

"Agilidad: 1, todo lo demás: 0."

Tom DeMarco

En un libro que suscita la reflexión sobre el desarrollo de software ágil, Alistair Cockburn [CocO2] argumenta que los modelos de proceso prescriptivo, introducidos en el capítulo 2, tienen una falla grande: *olvidan las flaquezas de las personas cuando construyen software*. Los ingenieros de software no son robots. Sus estilos de trabajo varían mucho; tienen diferencias significativas en habilidad, creatividad, orden, consistencia y espontaneidad. Algunos se comunican bien por escrito, pero otros no. Cockburn afirma que los modelos de proceso pueden "manejar las carencias de disciplina o tolerancia de las personas comunes" y que los modelos de proceso más prescriptivo eligen la disciplina. Dice: "Como la consistencia de las acciones es una debilidad humana, las metodología que requieren mucha disciplina son frágiles."

Para funcionar, los modelos de proceso deben proveer un mecanismo realista que estimule la disciplina necesaria, o deben caracterizarse por la "tolerancia" con las personas que hacen el trabajo de ingeniería de software. Invariablemente, las prácticas tolerantes son más fáciles de adoptar y sostener por parte de la comunidad del software, pero son menos productivas (como admite Cockburn). Debe considerarse la negociación entre ellas, como en todas las cosas de la vida.

3.1 ¿Qué es la agilidad?

Pero, ¿qué es la agilidad en el contexto del trabajo de la ingeniería de software? Ivar Jacobson [Jac02a] hace un análisis útil:

> La *agilidad* se ha convertido en la palabra mágica de hoy para describir un proceso del software moderno. Todos son ágiles. Un equipo ágil es diestro y capaz de responder de manera apropiada a los cambios. El cambio es de lo que trata el software en gran medida. Hay cambios en el software que se construye, en los miembros del equipo, debidos a las nuevas tecnologías, de todas clases y que tienen un efecto en el producto que se elabora o en el proyecto que lo crea. Deben introducirse apoyos para el cambio en todo lo que se haga en el software; en ocasiones se hace porque es el alma y corazón de éste. Un equipo ágil reconoce que el software es desarrollado por individuos que trabajan en equipo, y que su capacidad, su habilidad para colaborar, es el fundamento para el éxito del proyecto.

Desde el punto de vista de Jacobson, la ubicuidad del cambio es el motor principal de la agilidad. Los ingenieros de software deben ir rápido si han de adaptarse a los cambios veloces que describe Jacobson.

CONSEJO

No cometa el error de suponer que la agilidad le da permiso para improvisar soluciones. Se requiere de un proceso, y la disciplina es esencial.

Pero la agilidad es algo más que una respuesta efectiva al cambio. También incluye la filosofía expuesta en el manifiesto citado al principio de este capítulo. Ésta recomienda las estructuras de equipo y las actitudes que hacen más fácil la comunicación (entre los miembros del equipo, tecnólogos y gente de negocios, entre los ingenieros de software y sus gerentes, etc.); pone el énfasis en la entrega rápida de software funcional y resta importancia a los productos intermedios del trabajo (lo que no siempre es bueno); adopta al cliente como parte del equipo de desarrollo y trabaja para eliminar la actitud de "nosotros y ellos" que todavía invade muchos proyectos de software; reconoce que la planeación en un mundo incierto tiene sus límites y que un plan de proyecto debe ser flexible.

La agilidad puede aplicarse a cualquier proceso del software. Sin embargo, para lograrlo es esencial que éste se diseñe en forma que permita al equipo del proyecto adaptar las tareas y hacerlas directas, ejecutar la planeación de manera que entienda la fluidez de un enfoque ágil del desarrollo, eliminar todos los productos del trabajo excepto los más esenciales y mantenerlos esbeltos, y poner el énfasis en una estrategia de entrega incremental que haga trabajar al software tan rápido como sea posible para el cliente, según el tipo de producto y el ambiente de operación.

3.2 LA AGILIDAD Y EL COSTO DEL CAMBIO

La sabiduría convencional del desarrollo de software (apoyada por décadas de experiencia) señala que el costo se incrementa en forma no lineal a medida que el proyecto avanza (véase la figura 3.1, curva continua negra). Es relativamente fácil efectuar un cambio cuando el equipo de software reúne los requerimientos (al principio de un proyecto). El escenario de uso tal vez tenga que modificarse, la lista de funciones puede aumentar, o editarse una especificación escrita. Los costos de hacer que esto funcione son mínimos, y el tiempo requerido no perjudicará el resultado del proyecto. Pero, ¿qué pasa una vez transcurridos algunos meses? El equipo está a la mitad de las pruebas de validación (algo que ocurre cuando el proyecto está relativamente avanzado) y un participante de importancia solicita que se haga un cambio funcional grande. El cambio requiere modificar el diseño de la arquitectura del software, el diseño y construcción de tres componentes nuevos, hacer cambios en otros cinco componentes, diseñar nuevas pruebas, etc. Los costos aumentan con rapidez, y no son pocos el tiempo y el dinero requeridos para asegurar que se haga el cambio sin efectos colaterales no intencionados.

Cita:

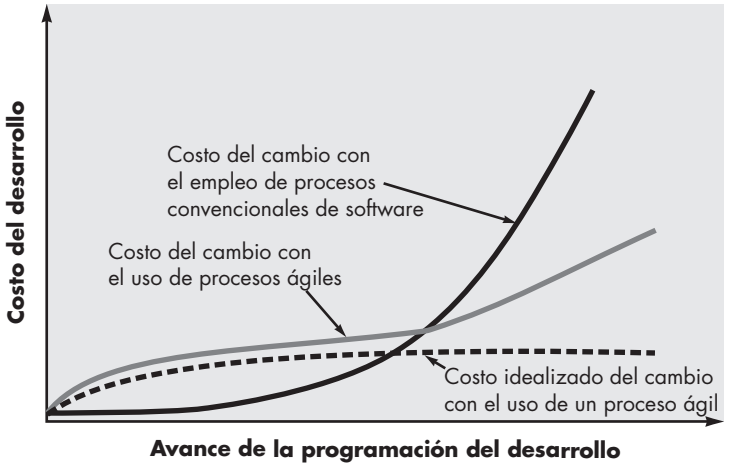
"La agilidad es dinámica, específica en el contenido, acepta con entusiasmo el cambio y se orienta al crecimiento."

Steven Goldman et al.

Los defensores de la agilidad (por ejemplo [Bec001] y [Amb04]) afirman que un proceso ágil bien diseñado "aplana" el costo de la curva de cambio (véase la figura 3.1, curva continua y

FIGURA 3.1

Cambio de los costos como función del tiempo transcurrido en el desarrollo



sombreada), lo que permite que el equipo de software haga cambios en una fase tardía de un proyecto de software sin que haya un efecto notable en el costo y en el tiempo. El lector ya sabe que el proceso ágil incluye la entrega incremental. Cuando ésta se acopla con otras prácticas ágiles, como las pruebas unitarias continuas y la programación por parejas (que se estudia más adelante, en este capítulo), el costo de hacer un cambio disminuye. Aunque hay debate sobre el grado en el que se aplana la curva de costo, existen evidencias [Coc01a] que sugieren que es posible lograr una reducción significativa del costo.

PUNTO CLAVE

Un proceso ágil reduce el costo del cambio porque el software se entrega en incrementos y en esta forma el cambio se controla mejor.

3.3 ¿Qué es un proceso ágil?

Cualquier proceso del software ágil se caracteriza por la forma en la que aborda cierto número de suposiciones clave [Fow02] acerca de la mayoría de proyectos de software:

1. Es difícil predecir qué requerimientos de software persistirán y cuáles cambiarán. También es difícil pronosticar cómo cambiarán las prioridades del cliente a medida que avanza el proyecto.
2. Para muchos tipos de software, el diseño y la construcción están imbricados. Es decir, ambas actividades deben ejecutarse en forma simultánea, de modo que los modelos de diseño se prueben a medida que se crean. Es difícil predecir cuánto diseño se necesita antes de que se use la construcción para probar el diseño.
3. El análisis, el diseño, la construcción y las pruebas no son tan predecibles como nos gustaría (desde un punto de vista de planeación).

WebRef

En la dirección **www.aanpo.org/articles/index** hay una colección completa de artículos sobre el proceso ágil.

Dadas estas tres suposiciones, surge una pregunta importante: ¿cómo crear un proceso que pueda manejar lo *impredecible*? La respuesta, como ya se dijo, está en la adaptabilidad del proceso (al cambio rápido del proyecto y a las condiciones técnicas). Por tanto, un proceso ágil debe ser *adaptable*.

Pero la adaptación continua logra muy poco si no hay avance. Entonces, un proceso de software ágil debe adaptarse *incrementalmente*. Para lograr la adaptación incremental, un equipo ágil requiere retroalimentación con el cliente (de modo que sea posible hacer las adaptaciones apropiadas). Un catalizador eficaz para la retroalimentación con el cliente es un prototipo operativo o una porción de un sistema operativo. Así, debe instituirse una *estrategia de desarrollo incremental*. Deben entregarse *incrementos de software* (prototipos ejecutables o porciones de un sistema operativo) en periodos cortos de tiempo, de modo que la adaptación vaya a ritmo con el cambio (impredecible). Este enfoque iterativo permite que el cliente evalúe en forma regular el incremento de software, dé la retroalimentación necesaria al equipo de software e influya en las adaptaciones del proceso que se realicen para aprovechar la retroalimentación.

3.3.1 Principios de agilidad

PUNTO CLAVE

Aunque los procesos ágiles aceptan el cambio, es importante examinar las razones de éste.

La Alianza Ágil (véase [Agi03]), [Fow01]) define 12 principios de agilidad para aquellos que la quieran alcanzar:

1. La prioridad más alta es satisfacer al cliente a través de la entrega pronta y continua de software valioso.
2. Son bienvenidos los requerimientos cambiantes, aun en una etapa avanzada del desarrollo. Los procesos ágiles dominan el cambio para provecho de la ventaja competitiva del cliente.
3. Entregar con frecuencia software que funcione, de dos semanas a un par de meses, de preferencia lo más pronto que se pueda.

4. Las personas de negocios y los desarrolladores deben trabajar juntos, a diario y durante todo el proyecto.
5. Hay que desarrollar los proyectos con individuos motivados. Debe darse a éstos el ambiente y el apoyo que necesiten, y confiar en que harán el trabajo.
6. El método más eficiente y eficaz para transmitir información a los integrantes de un equipo de desarrollo, y entre éstos, es la conversación cara a cara.
7. La medida principal de avance es el software que funciona.
8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben poder mantener un ritmo constante en forma indefinida.
9. La atención continua a la excelencia técnica y el buen diseño mejora la agilidad.
10. Es esencial la simplicidad: el arte de maximizar la cantidad de trabajo no realizado.
11. Las mejores arquitecturas, requerimientos y diseños surgen de los equipos con organización propia.
12. El equipo reflexiona a intervalos regulares sobre cómo ser más eficaz, para después afinar y ajustar su comportamiento en consecuencia.

El software que funciona es importante, pero no olvide que también debe poseer varios atributos de calidad, como ser confiable, utilizable y susceptible de recibir mantenimiento.

No todo modelo de proceso ágil aplica estos 12 principios con igual intensidad y algunos eligen ignorar (o al menos soslayar) la importancia de uno o más de ellos. Sin embargo, los principios definen un *espíritu ágil* que se mantiene en cada uno de los modelos de proceso que se presentan en este capítulo.

3.3.2 La política del desarrollo ágil

Hay mucho debate (a veces estridente) sobre los beneficios y aplicabilidad del desarrollo de software ágil como oposición a los procesos más convencionales. Jim Highsmith [Hig02a] señala (en tono de burla) los extremos cuando caracteriza la posición del campo a favor de la agilidad ("agilistas"). "Los metodólogos tradicionales están atrapados en un pantano y producirán una documentación sin defectos en vez de un sistema funcional que satisfaga las necesidades del negocio." Como contrapunto, plantea (de nuevo como burla) la posición del campo de la ingeniería de software tradicional: "Los metodólogos ligeros, perdón, 'ágiles', son un grupo de remendones famosos que se van a llevar una sorpresa cuando intenten convertir sus juguetes en software a la medida de la empresa."

No tiene que elegirse entre la agilidad y la ingeniería de software. En vez de ello, hay que definir un enfoque de ingeniería de software que sea ágil.

Como todos los argumentos sobre la tecnología de software, este debate sobre la metodología corre el riesgo de degenerar en una guerra religiosa. Si estalla, desaparece el pensamiento racional y lo que guía la toma de decisiones son las creencias y no los hechos.

Nadie está contra la agilidad. La pregunta real es: ¿cuál es la mejor forma de lograrla? De igual importancia: ¿cómo construir software que satisfaga en el momento las necesidades de los clientes y que tenga características de calidad que permitan ampliarlo y escalarlo para que también las satisfaga en el largo plazo?

No hay respuestas absolutas a ninguna de estas preguntas. Aun dentro de la escuela ágil hay muchos modelos de proceso propuestos (véase la sección 3.4), cada uno con un enfoque algo diferente para el problema de la agilidad. Dentro de cada modelo hay un conjunto de "ideas" (los agilistas las llaman "tareas del trabajo") que representan un alejamiento significativo de la ingeniería de software tradicional. No obstante, muchos conceptos ágiles sólo son adaptaciones de algunos que provienen de la buena ingeniería de software. En resumen: hay mucho por ganar si se considera lo mejor de ambas escuelas, y virtualmente no se gana nada si se denigra cualquiera de los enfoques.

Si el lector está interesado, consulte [Hig01], [Hig02a] y [DeM02] para ver un resumen ameno de otros aspectos técnicos y políticos importantes.

3.3.3 Factores humanos

Cita:

"Los métodos ágiles obtienen gran parte de su agilidad por basarse en el conocimiento tácito incorporado en el equipo, más que en escribir el conocimiento en planes."

Barry Boehm

Los defensores del desarrollo de software ágil se toman muchas molestias para enfatizar la importancia de los "factores personales". Como dicen Cockburn y Highsmith [Coc01a]: "El desarrollo ágil se centra en los talentos y habilidades de los individuos, y adapta el proceso a personas y equipos específicos." El punto clave de esta afirmación es que *el proceso se adapta a las necesidades de las personas y del equipo*, no al revés.[2]

Si los miembros del equipo de software son los que van a generar las características del proceso que van a aplicarse a la elaboración de software, entre ellos debe existir cierto número de características clave, mismas que debe compartir el equipo ágil como tal:

¿Qué características clave deben existir entre los integrantes de un equipo eficaz de software?

Competencia. En un contexto de desarrollo ágil (así como en la ingeniería de software), la "competencia" incluye el talento innato, las habilidades específicas relacionadas con el software y el conocimiento general del proceso que el equipo haya elegido aplicar. La habilidad y el conocimiento del proceso pueden y deben considerarse para todas las personas que sean miembros ágiles del equipo.

Enfoque común. Aunque los miembros del equipo ágil realicen diferentes tareas y aporten habilidades distintas al proyecto, todos deben centrarse en una meta: entregar al cliente en la fecha prometida un incremento de software que funcione. Para lograrlo, el equipo también se centrará en adaptaciones continuas (pequeñas y grandes) que hagan que el proceso se ajuste a las necesidades del equipo.

Colaboración. La ingeniería de software (sin importar el proceso) trata de evaluar, analizar y usar la información que se comunica al equipo de software; crear información que ayudará a todos los participantes a entender el trabajo del equipo; y generar información (software de cómputo y bases de datos relevantes) que aporten al cliente valor del negocio. Para efectuar estas tareas, los miembros del equipo deben colaborar, entre sí y con todos los participantes.

Cita:

"Lo que para un equipo es apenas suficiente, para otro es más que suficiente y para otro más resulta insuficiente."

Alistair Cockburn

Habilidad para tomar decisiones. Cualquier equipo bueno de software (incluso los equipos ágiles) debe tener libertad para controlar su destino. Esto implica que se dé autonomía al equipo: autoridad para tomar decisiones sobre asuntos tanto técnicos como del proyecto.

Capacidad para resolver problemas difusos. Los gerentes de software deben reconocer que el equipo ágil tendrá que tratar en forma continua con la ambigüedad y que será sacudido de manera permanente por el cambio. En ciertos casos, el equipo debe aceptar el hecho de que el problema que resuelven ahora tal vez no sea el que se necesite resolver mañana. Sin embargo, las lecciones aprendidas de cualquier actividad relacionada con la solución de problemas (incluso aquellas que resuelven el problema equivocado) serán benéficas para el equipo en una etapa posterior del proyecto.

Confianza y respeto mutuos. El equipo ágil debe convertirse en lo que DeMarco y Lister [DeM98] llaman "pegado" (véase el capítulo 24). Un equipo pegado tiene la confianza y respeto que son necesarios para hacer "su tejido tan fuerte que el todo es más que la suma de sus partes" [DeM98].

PUNTO CLAVE

Un equipo con organización propia tiene el control del trabajo que realiza. Establece sus propios compromisos y define los planes para lograrlo.

Organización propia. En el contexto del desarrollo ágil, la organización propia implica tres cosas: 1) el equipo ágil se organiza a sí mismo para hacer el trabajo, 2) el equipo organiza el proceso que se adapte mejor a su ambiente local, 3) el equipo organiza la programación del trabajo a fin de que se logre del mejor modo posible la entrega del incremento

2 Las organizaciones exitosas de ingeniería de software reconocen esta realidad sin importar el modelo de proceso que elijan.

de software. La organización propia tiene cierto número de beneficios técnicos, pero, lo que es más importante, sirve para mejorar la colaboración y elevar la moral del equipo. En esencia, el equipo sirve como su propio gerente. Ken Schwaber [Sch02] aborda estos aspectos cuando escribe: "El equipo selecciona cuánto trabajo cree que puede realizar en cada iteración, y se compromete con la labor. Nada desmotiva tanto a un equipo como que alguien establezca compromisos por él. Nada motiva más a un equipo como aceptar la responsabilidad de cumplir los compromisos que haya hecho él mismo."

3.4 Programación extrema (XP)

A fin de ilustrar un proceso ágil con más detalle, daremos un panorama de la *programación extrema* (XP), el enfoque más utilizado del desarrollo de software ágil. Aunque las primeras actividades con las ideas y los métodos asociados a XP ocurrieron al final de la década de 1980, el trabajo fundamental sobre la materia había sido escrito por Kent Beck [Bec04a]. Una variante de XP llamada *XP industrial* [IXP] se propuso en una época más reciente [Ker05]. IXP mejora la XP y tiene como objetivo el proceso ágil para ser usado específicamente en organizaciones grandes.

3.4.1 Valores XP

Beck [Bec04a] define un conjunto de cinco *valores* que establecen el fundamento para todo trabajo realizado como parte de XP: comunicación, simplicidad, retroalimentación, valentía y respeto. Cada uno de estos valores se usa como un motor para actividades, acciones y tareas específicas de XP.

A fin de lograr la *comunicación* eficaz entre los ingenieros de software y otros participantes (por ejemplo, para establecer las características y funciones requeridas para el software), XP pone el énfasis en la colaboración estrecha pero informal (verbal) entre los clientes y los desarrolladores, en el establecimiento de metáforas[3] para comunicar conceptos importantes, en la retroalimentación continua y en evitar la documentación voluminosa como medio de comunicación.

Mantenlo sencillo siempre que se pueda, pero reconoce que el "rediseño" continuo consume mucho tiempo y recursos.

Para alcanzar la *simplicidad*, XP restringe a los desarrolladores para que diseñen sólo para las necesidades inmediatas, en lugar de considerar las del futuro. El objetivo es crear un diseño sencillo que se implemente con facilidad en forma de código. Si hay que mejorar el diseño, se rediseñará[4] en un momento posterior.

La *retroalimentación* se obtiene de tres fuentes: el software implementado, el cliente y otros miembros del equipo de software. Al diseñar e implementar una estrategia de pruebas eficaz (capítulos 17 a 20), el software (por medio de los resultados de las pruebas) da retroalimentación al equipo ágil. XP usa la *prueba unitaria* como su táctica principal de pruebas. A medida que se desarrolla cada clase, el equipo implementa una prueba unitaria para ejecutar cada operación de acuerdo con su funcionalidad especificada. Cuando se entrega un incremento a un cliente, las *historias del usuario* o *casos de uso* (véase el capítulo 5) que se implementan con el incremento se utilizan como base para las pruebas de aceptación. El grado en el que el software implementa la salida, función y comportamiento del caso de uso es una forma de retroalimentación. Por último, conforme se obtienen nuevos requerimientos como parte de la planeación iterativa, el equipo da al cliente una retroalimentación rápida con miras al costo y al efecto en la programación de actividades.

---

3 En el contexto de XP, una *metáfora* es "una historia que cada quien —clientes, programadores y gerentes— narra, acerca de cómo funciona el sistema" [Bec04a].

4 El rediseño permite que un ingeniero mejore la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externos. En esencia, el rediseño puede utilizarse para mejorar la eficiencia, disponibilidad o rendimiento de un diseño o del código que lo implementa.

Cita:

"XP es la respuesta a la pregunta: '¿Cuán pequeño podemos hacer un gran software?'."

Anónimo

Beck [Bec04a] afirma que la adhesión estricta a ciertas prácticas de XP requiere *valentía*. Un término más apropiado sería *disciplina*. Por ejemplo, es frecuente que haya mucha presión para diseñar requerimientos futuros. La mayor parte de equipos de software sucumben a ella y se justifican porque "diseñar para el mañana" ahorrará tiempo y esfuerzo en el largo plazo. Un equipo XP ágil debe tener la disciplina (valentía) para diseñar para hoy y reconocer que los requerimientos futuros tal vez cambien mucho, por lo que demandarán repeticiones sustanciales del diseño y del código implementado.

Al apegarse a cada uno de estos valores, el equipo ágil inculca *respeto* entre sus miembros, entre otros participantes y los integrantes del equipo, e indirectamente para el software en sí mismo. Conforme logra la entrega exitosa de incrementos de software, el equipo desarrolla más respeto para el proceso XP.

3.4.2 El proceso XP

WebRef

En la dirección **www.extremeprogramming.org/rules.html**, se encuentra un panorama excelente de las "reglas" de XP.

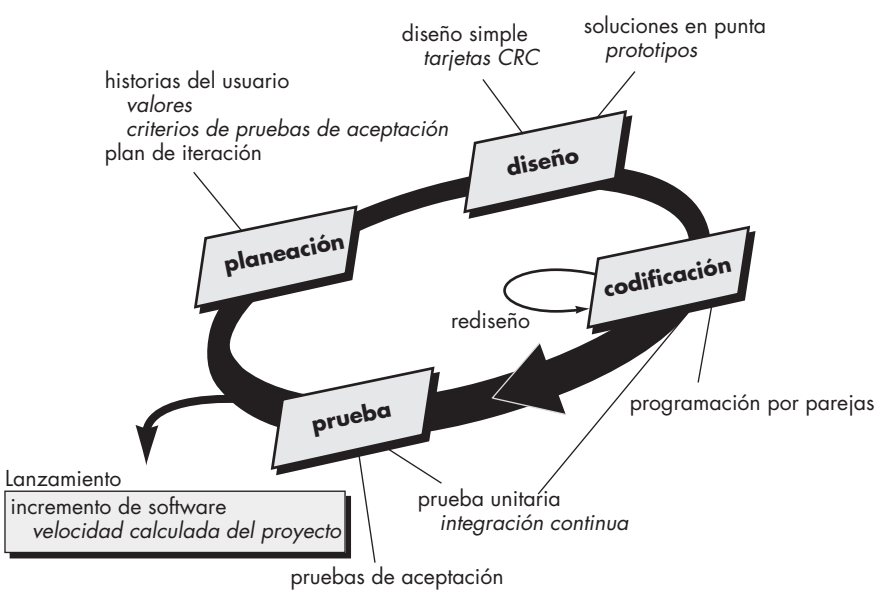
La programación extrema usa un enfoque orientado a objetos (véase el apéndice 2) como paradigma preferido de desarrollo, y engloba un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades estructurales: planeación, diseño, codificación y pruebas. La figura 3.2 ilustra el proceso XP y resalta algunas de las ideas y tareas clave que se asocian con cada actividad estructural. En los párrafos que siguen se resumen las actividades de XP clave.

Planeación. La actividad de planeación (también llamada *juego de planeación*) comienza *escuchando* —actividad para recabar requerimientos que permite que los miembros técnicos del equipo XP entiendan el contexto del negocio para el software y adquieran la sensibilidad de la salida y características principales y funcionalidad que se requieren—. Escuchar lleva a la creación de algunas "historias" (también llamadas *historias del usuario*) que describen la salida necesaria, características y funcionalidad del software que se va a elaborar. Cada *historia* (similar a los casos de uso descritos en el capítulo 5) es escrita por el cliente y colocada en una tarjeta indizada. El cliente asigna un *valor* (es decir, una prioridad) a la historia con base en el valor general de la característica o función para el negocio.[5] Después, los miembros del equipo XP

? ¿Qué es una "historia" XP?

FIGURA 3.2

El proceso de la programación extrema



5 El valor de una historia también puede depender de la presencia de otra historia.

evalúan cada historia y le asignan un costo, medido en semanas de desarrollo. Si se estima que la historia requiere más de tres semanas de desarrollo, se pide al cliente que la descomponga en historias más chicas y de nuevo se asigna un valor y costo. Es importante observar que en cualquier momento es posible escribir nuevas historias.

WebRef

En la dirección **c2.com/cgi/wiki?planningGame**, se halla un "juego de planeación" XP provechoso.

Los clientes y desarrolladores trabajan juntos para decidir cómo agrupar las historias en la siguiente entrega (el siguiente incremento de software) que desarrollará el equipo XP. Una vez que se llega a un *compromiso* sobre la entrega (acuerdo sobre las historias por incluir, la fecha de entrega y otros aspectos del proyecto), el equipo XP ordena las historias que serán desarrolladas en una de tres formas: 1) todas las historias se implementarán de inmediato (en pocas semanas), 2) las historias con más valor entrarán a la programación de actividades y se implementarán en primer lugar o 3) las historias más riesgosas formarán parte de la programación de actividades y se implementarán primero.

La velocidad del proyecto es una medición sutil de la productividad del equipo.

Después de la primera entrega del proyecto (también llamada incremento de software), el equipo XP calcula la velocidad de éste. En pocas palabras, la *velocidad del proyecto* es el número de historias de los clientes implementadas durante la primera entrega. La velocidad del proyecto se usa para: 1) ayudar a estimar las fechas de entrega y programar las actividades para las entregas posteriores, y 2) determinar si se ha hecho un gran compromiso para todas las historias durante todo el desarrollo del proyecto. Si esto ocurre, se modifica el contenido de las entregas o se cambian las fechas de entrega final.

A medida que avanza el trabajo, el cliente puede agregar historias, cambiar el valor de una ya existente, descomponerlas o eliminarlas. Entonces, el equipo XP reconsidera todas las entregas faltantes y modifica sus planes en consecuencia.

Diseño. El diseño XP sigue rigurosamente el principio MS (mantenlo sencillo). Un diseño sencillo siempre se prefiere sobre una representación más compleja. Además, el diseño guía la implementación de una historia conforme se escribe: nada más y nada menos. Se desalienta el diseño de funcionalidad adicional porque el desarrollador supone que se requerirá después.[6]

XP desalienta la importancia del diseño, opinión con la que no todos están de acuerdo. En realidad, hay veces en las que debe hacerse énfasis en el diseño.

XP estimula el uso de las tarjetas CRC (véase el capítulo 7) como un mecanismo eficaz para pensar en el software en un contexto orientado a objetos. Las tarjetas CRC (clase-responsabilidad-colaborador) identifican y organizan las clases orientadas a objetos[7] que son relevantes para el incremento actual de software. El equipo XP dirige el ejercicio de diseño con el uso de un proceso similar al que se describe en el capítulo 8. Las tarjetas CRC son el único producto del trabajo de diseño que se genera como parte del proceso XP.

Si en el diseño de una historia se encuentra un problema de diseño difícil, XP recomienda la creación inmediata de un prototipo operativo de esa porción del diseño. Entonces, se implementa y evalúa el prototipo del diseño, llamado *solución en punta*. El objetivo es disminuir el riesgo cuando comience la implementación verdadera y validar las estimaciones originales para la historia que contiene el problema de diseño.

WebRef

En la dirección **www.refactoring.com** se encuentran técnicas y herramientas de rediseño.

En la sección anterior se dijo que XP estimula el *rediseño*, técnica de construcción que también es un método para la optimización del diseño. Fowler [Fow00] describe el rediseño del modo siguiente:

> *Rediseño* es el proceso mediante el cual se cambia un sistema de software en forma tal que no altere el comportamiento externo del código, pero sí mejore la estructura interna. Es una manera disciplinada de limpiar el código [y modificar o simplificar el diseño interno] que minimiza la probabilidad de introducir errores. En esencia, cuando se rediseña, se mejora el diseño del código después de haber sido escrito.

6 Estos lineamientos de diseño deben seguirse en todo método de ingeniería de software, aunque hay ocasiones en los que la notación y terminología sofisticadas del diseño son un camino hacia la simplicidad.

7 Las clases orientadas a objetos se estudian en el apéndice 2, en el capítulo 8 y en toda la parte 2 de este libro.

PUNTO CLAVE

El rediseño mejora la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externo.

Como el diseño XP virtualmente no utiliza notación y genera pocos, si alguno, productos del trabajo que no sean tarjetas CRC y soluciones en punta, el diseño es visto como un artefacto en transición que puede y debe modificarse continuamente a medida que avanza la construcción. El objetivo del rediseño es controlar dichas modificaciones, sugiriendo pequeños cambios en el diseño que "son capaces de mejorarlo en forma radical" [Fow00]. Sin embargo, debe notarse que el esfuerzo que requiere el rediseño aumenta en forma notable con el tamaño de la aplicación.

Un concepto central en XP es que el diseño ocurre tanto antes *como después* de que comienza la codificación. Rediseñar significa que el diseño se hace de manera continua conforme se construye el sistema. En realidad, la actividad de construcción en sí misma dará al equipo XP una guía para mejorar el diseño.

Codificación. Después de que las historias han sido desarrolladas y de que se ha hecho el trabajo de diseño preliminar, el equipo *no* inicia la codificación, sino que desarrolla una serie de pruebas unitarias a cada una de las historias que se van a incluir en la entrega en curso (incremento de software).[8] Una vez creada la prueba unitaria,[9] el desarrollador está mejor capacitado para centrarse en lo que debe implementarse para pasar la prueba. No se agrega nada extraño (MS). Una vez que el código está terminado, se le aplica de inmediato una prueba unitaria, con lo que se obtiene retroalimentación instantánea para los desarrolladores.

WebRef

Hay información útil acerca de XP en la dirección **www.xprogramming.com.**

¿Qué es la programación por parejas?

Un concepto clave durante la actividad de codificación (y uno de los aspectos del que más se habla en la XP) es la *programación por parejas*. XP recomienda que dos personas trabajen juntas en una estación de trabajo con el objeto de crear código para una historia. Esto da un mecanismo para la solución de problemas en tiempo real (es frecuente que dos cabezas piensen más que una) y para el aseguramiento de la calidad también en tiempo real (el código se revisa conforme se crea). También mantiene a los desarrolladores centrados en el problema de que se trate. En la práctica, cada persona adopta un papel un poco diferente. Por ejemplo, una de ellas tal vez piense en los detalles del código de una porción particular del diseño, mientras la otra se asegura de que se siguen los estándares de codificación (parte necesaria de XP) o de que el código para la historia satisfará la prueba unitaria desarrollada a fin de validar el código confrontándolo con la historia.

CONSEJO

Muchos equipos de software están llenos de individualistas. Si la programación por parejas ha de funcionar con eficacia, tendrá que trabajar para cambiar esa cultura.

A medida que las parejas de programadores terminan su trabajo, el código que desarrollan se integra con el trabajo de los demás. En ciertos casos, esto lo lleva a cabo a diario un equipo de integración. En otros, las parejas de programadores tienen la responsabilidad de la integración. Esta estrategia de "integración continua" ayuda a evitar los problemas de compatibilidad e interfaces y brinda un ambiente de "prueba de humo" (véase el capítulo 17) que ayuda a descubrir a tiempo los errores.

¿Cómo se usan las pruebas unitarias en XP?

Pruebas. Ya se dijo que la creación de pruebas unitarias antes de que comience la codificación es un elemento clave del enfoque de XP. Las pruebas unitarias que se crean deben implementarse con el uso de una estructura que permita automatizarlas (de modo que puedan ejecutarse en repetidas veces y con facilidad). Esto estimula una estrategia de pruebas de regresión (véase el capítulo 17) siempre que se modifique el código (lo que ocurre con frecuencia, dada la filosofía del rediseño en XP).

A medida que se organizan las pruebas unitarias individuales en un "grupo de prueba universal" [Wel99], las pruebas de la integración y validación del sistema pueden efectuarse a diario. Esto da al equipo XP una indicación continua del avance y también lanza señales de alerta si las

8 Este enfoque es equivalente a saber las preguntas del examen antes de comenzar a estudiar. Vuelve mucho más fácil el estudio porque centra la atención sólo en las preguntas que se van a responder.

9 La prueba unitaria, que se estudia en detalle en el capítulo 17, se centra en un componente de software individual sobre interfaz, estructuras de datos y funcionalidad del componente, en un esfuerzo por descubrir errores locales del componente.

cosas marchan mal. Wells [Wel99] dice: "Corregir pequeños problemas cada cierto número de horas toma menos tiempo que resolver problemas enormes justo antes del plazo final."

Las *pruebas de aceptación* XP, también llamadas *pruebas del cliente*, son especificadas por el cliente y se centran en las características y funcionalidad generales del sistema que son visibles y revisables por parte del cliente. Las pruebas de aceptación se derivan de las historias de los usuarios que se han implementado como parte de la liberación del software.

PUNTO CLAVE
Las pruebas de aceptación se derivan de las historias de los usuarios.

3.4.3 XP industrial

Joshua Kerievsky [Ker05] describe la *programación extrema industrial* [IXP, por sus siglas en inglés] en la forma siguiente: "IXP es la evolución orgánica de XP. Está imbuida del espíritu minimalista, centrado en el cliente y orientado a las pruebas que tiene XP. IXP difiere sobre todo de la XP original en su mayor inclusión de la gerencia, el papel más amplio de los clientes y en sus prácticas técnicas actualizadas". IXP incorpora seis prácticas nuevas diseñadas para ayudar a garantizar que un proyecto XP funciona con éxito para proyectos significativos dentro de una organización grande.

? ¿Qué nuevas prácticas se agregan a XP para crear IXP?

Evaluación de la factibilidad. Antes de iniciar un proyecto IXP, la organización debe efectuar una *evaluación de la factibilidad*. Ésta deja en claro si: 1) existe un ambiente apropiado de desarrollo que acepte IXP, 2) el equipo estará constituido por los participantes adecuados, 3) la organización tiene un programa de calidad distintivo y apoya la mejora continua, 4) la cultura organizacional apoyará los nuevos valores de un equipo ágil, y 5) la comunidad extendida del proyecto estará constituida de modo apropiado.

Cita:

"Habilidad es aquello que eres capaz de hacer. La motivación determina lo que haces. La actitud determina cuán bien lo haces."

Lou Holtz

Comunidad del proyecto. La XP clásica sugiere que se utilice personal apropiado para formar el equipo ágil a fin de asegurar el éxito. La implicación es que las personas en el equipo deben estar bien capacitadas, ser adaptables y hábiles, y tener el temperamento apropiado para contribuir al equipo con organización propia. Cuando se aplica XP a un proyecto significativo en una organización grande, el concepto de "equipo" debe adoptar la forma de *comunidad*. Una comunidad puede tener un tecnólogo y clientes que son fundamentales para el éxito del proyecto, así como muchos otros participantes (equipo jurídico; auditores de calidad, de tipos de manufactura o de ventas, etc.) que "con frecuencia se encuentran en la periferia en un proyecto IXP, pero que desempeñan en éste papeles importantes" [Ker05]. En IXP, los miembros de la comunidad y sus papeles deben definirse de modo explícito, así como establecer los mecanismos para la comunicación y coordinación entre los integrantes de la comunidad.

Calificación del proyecto. El equipo de IXP evalúa el proyecto para determinar si existe una justificación apropiada de negocios y si el proyecto cumplirá las metas y objetivos generales de la organización. La calificación también analiza el contexto del proyecto a fin de determinar cómo complementa, extiende o reemplaza sistemas o procesos existentes.

Administración orientada a pruebas. Un proyecto IXP requiere criterios medibles para evaluar el estado del proyecto y el avance realizado. La administración orientada a pruebas establece una serie de "destinos" medibles [Ker05] y luego define los mecanismos para determinar si se han alcanzado o no éstos.

Retrospectivas. Después de entregar un incremento de software, el equipo XP realiza una revisión técnica especializada que se llama *retrospectiva* y que examina "los temas, eventos y lecciones aprendidas" [Ker05] a lo largo del incremento de software y/o de la liberación de todo el software. El objetivo es mejorar el proceso IXP.

Aprendizaje continuo. Como el aprendizaje es una parte vital del proceso de mejora continua, los miembros del equipo XP son invitados (y tal vez incentivados) a aprender nuevos métodos y técnicas que conduzcan a una calidad más alta del producto.

Además de las seis nuevas prácticas analizadas, IXP modifica algunas de las existentes en XP. El *desarrollo impulsado por la historia* (DIH) insiste en que las historias de las pruebas de aceptación se escriban antes de generar una sola línea de código. El *diseño impulsado por el dominio* (DID) es una mejora sobre el concepto de la "metáfora del sistema" usado en XP. El DID [Eva03] sugiere la creación evolutiva de un modelo de dominio que "represente con exactitud cómo piensan los expertos del dominio en su materia" [Ker05]. La *formación de parejas* amplía el concepto de programación en pareja para que incluya a los gerentes y a otros participantes. El objetivo es mejorar la manera de compartir conocimientos entre los integrantes del equipo XP que no estén directamente involucrados en el desarrollo técnico. La *usabilidad iterativa* desalienta el diseño de una interfaz cargada al frente y estimula un diseño que evoluciona a medida que se liberan los incrementos de software y que se estudia la interacción de los usuarios con el software.

La IXP hace modificaciones más pequeñas a otras prácticas XP y redefine ciertos roles y responsabilidades para hacerlos más asequibles a proyectos significativos de las organizaciones grandes. Para mayores detalles de IXP, visite el sitio **http://industrialxp.org**.

3.4.4 El debate XP

Los nuevos modelos y métodos de proceso han motivado análisis provechosos y en ciertas instancias debates acalorados. La programación extrema desencadena ambos. En un libro interesante que examina la eficacia de XP, Stephens y Rosenberg [Ste03] afirman que muchas prácticas de XP son benéficas, pero que otras están sobreestimadas y unas más son problemáticas. Los autores sugieren que la naturaleza codependiente de las prácticas de XP constituye tanto su fortaleza como su debilidad. Debido a que muchas organizaciones adoptan sólo un subconjunto de prácticas XP, debilitan la eficacia de todo el proceso. Los defensores contradicen esto al afirmar que la XP está en evolución continua y que muchas de las críticas que se le hacen han llevado a correcciones conforme maduran sus prácticas. Entre los aspectos que destacan algunos críticos de la XP están los siguientes:[10]

? ¿Cuáles son algunos de los aspectos que llevan al debate de XP?

- *Volatilidad de los requerimientos.* Como el cliente es un miembro activo del equipo XP, los cambios a los requerimientos se solicitan de manera informal. En consecuencia, el alcance del proyecto cambia y el trabajo inicial tiene que modificarse para dar acomodo a las nuevas necesidades. Los defensores afirman que esto pasa sin importar el proceso que se aplique y que la XP proporciona mecanismos para controlar los vaivenes del alcance.
- *Necesidades conflictivas del cliente.* Muchos proyectos tienen clientes múltiples, cada uno con sus propias necesidades. En XP, el equipo mismo tiene la tarea de asimilar las necesidades de distintos clientes, trabajo que tal vez esté más allá del alcance de su autoridad.
- *Los requerimientos se expresan informalmente.* Las historias de usuario y las pruebas de aceptación son la única manifestación explícita de los requerimientos en XP. Los críticos afirman que es frecuente que se necesite un modelo o especificación más formal para garantizar que se descubran las omisiones, inconsistencias y errores antes de que se construya el sistema. Los defensores contraatacan diciendo que la naturaleza cambiante de los requerimientos vuelve obsoletos esos modelos y especificaciones casi tan pronto como se desarrollan.
- *Falta de un diseño formal.* XP desalienta la necesidad del diseño de la arquitectura y, en muchas instancias, sugiere que el diseño de todas las clases debe ser relativamente informal. Los críticos argumentan que cuando se construyen sistemas complejos, debe ponerse el énfasis en el diseño con el objeto de garantizar que la estructura general del software tendrá calidad y que será susceptible de recibir mantenimiento. Los defensores

10 Para un estudio más detallado de ciertas críticas profundas hechas a XP, visite www.softwarereality.com/ExtremeProgramming.jsp.

de XP sugieren que la naturaleza incremental del proceso XP limita la complejidad (la sencillez es un valor fundamental), lo que reduce la necesidad de un diseño extenso.

El lector debe observar que todo proceso del software tiene sus desventajas, y que muchas organizaciones de software han utilizado con éxito la XP. La clave es identificar dónde tiene sus debilidades un proceso y adaptarlo a las necesidades de la organización.

3.5 Otros modelos ágiles de proceso

Cita:

"Nuestra profesión pasa por las metodologías como un chico de 14 años pasa por la ropa."

Stephen Hawrysh y Jim Ruprecht

La historia de la ingeniería de software está salpicada de decenas de descripciones y metodologías de proceso, métodos de modelado y notaciones, herramientas y tecnología, todos ellos obsoletos. Cada uno tuvo notoriedad y luego fue eclipsado por algo nuevo y (supuestamente) mejor. Con la introducción de una amplia variedad de modelos ágiles del proceso —cada uno en lucha por la aceptación de la comunidad de desarrollo de software— el movimiento ágil está siguiendo la misma ruta histórica.[11]

CasaSegura

Consideración del desarrollo ágil de software

La escena: Oficina de Doug Miller.

Participantes: Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software; Vinod Raman, integrante del equipo de software.

La conversación:

(Tocan a la puerta; Jamie y Vinod entran a la oficina de Doug.)

Jamie: Doug, ¿tienes un minuto?

Doug: Seguro, Jamie, ¿qué pasa?

Jamie: Hemos estado pensando en nuestra conversación de ayer sobre el proceso… ya sabes, el que vamos a elegir para este nuevo proyecto de *CasaSegura*.

Doug: ¿Y?

Vinod: Hablé con un amigo de otra compañía, y me contó sobre la programación extrema. Es un modelo de proceso ágil… ¿has oído de él?

Doug: Sí, algunas cosas buenas y otras malas.

Jamie: Bueno, a nosotros nos pareció bien. Permite el desarrollo de software realmente rápido, usa algo llamado programación en parejas para revisar la calidad en tiempo real… Pienso que es muy bueno.

Doug: Tiene muchas ideas realmente buenas. Por ejemplo, me gusta el concepto de programación en parejas y la idea de que los participantes deben formar parte del equipo.

Jamie: ¿Qué? ¿Quieren decir que mercadotecnia trabajará con nosotros en el proyecto?

Doug (afirmando con la cabeza): Ellos son uno de los participantes, ¿o no?

Jamie: Sí… Pedirán cambios cada cinco minutos.

Vinod: No necesariamente. Mi amigo dijo que hay formas de "adoptar" los cambios durante un proyecto de XP.

Doug: Entonces, chicos, ¿piensan que debemos usar XP?

Jamie: Definitivamente, sería bueno considerarlo.

Doug: Estoy de acuerdo. E incluso si elegimos un modelo incremental como nuestro enfoque, no hay razón para no incorporar mucho de lo que XP tiene que ofrecer.

Vinod: Doug, dijiste hace un rato "cosas buenas y malas". ¿Cuáles son las malas?

Doug: Lo que no me gusta es la forma en la que XP desprecia el análisis y el diseño… algo así como decir que la escritura del código está donde hay acción…

(Los miembros del equipo se miran entre sí y sonríen.)

Doug: Entonces, ¿están de acuerdo con el enfoque XP?

Jamie (habla por ambos): ¡Escribir código es lo que hacemos, jefe!

Doug (ríe): Es cierto, pero me gustaría ver que dediquen un poco menos de tiempo a escribir código y luego a repetirlo, y que pasen algo más de tiempo en el análisis de lo que debe hacerse para diseñar una solución que funcione.

Vinod: Tal vez tengamos las dos cosas, agilidad con un poco de disciplina.

Doug: Creo que podemos, Vinod. En realidad, estoy seguro de que se puede.

11 Esto no es algo malo. Antes de que uno o varios modelos se acepten como el estándar *de facto*, todos deben luchar por conquistar las mentes y corazones de los ingenieros de software. Los "ganadores" evolucionan hacia las mejores prácticas, mientras que los "perdedores" desaparecen o se funden con los modelos ganadores.

Como se dijo en la sección anterior, el más usado de todos los modelos ágiles de proceso es la programación extrema (XP). Pero se han propuesto muchos otros y están en uso en toda la industria. Entre ellos se encuentran los siguientes:

- Desarrollo adaptativo de software (DAS)
- Scrum
- Método de desarrollo de sistemas dinámicos (MDSD)
- Cristal
- Desarrollo impulsado por las características (DIC)
- Desarrollo esbelto de software (DES)
- Modelado ágil (MA)
- Proceso unificado ágil (PUA)

En las secciones que siguen se presenta un panorama muy breve de cada uno de estos modelos ágiles del proceso. Es importante notar que *todos* los modelos de proceso ágil se apegan (en mayor o menor grado) al *Manifiesto para el desarrollo ágil de software* y a los principios descritos en la sección 3.3.1. Para mayores detalles, consulte las referencias mencionadas en cada subsección o ingrese en la entrada "desarrollo de software ágil" de Wikipedia.[12]

3.5.1 Desarrollo adaptativo de software (DAS)

WebRef

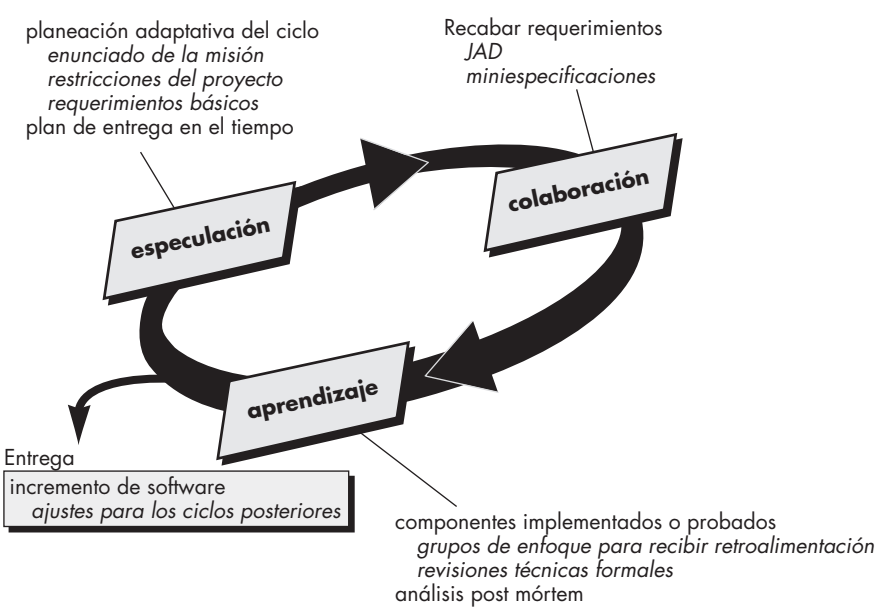
En la dirección **www.adaptivesd.com** hay referencias útiles sobre el DAS.

El *desarrollo adaptativo de software* (DAS) fue propuesto por Jim Highsmith [Hig00] como una técnica para elaborar software y sistemas complejos. Los fundamentos filosóficos del DAS se centran en la colaboración humana y en la organización propia del equipo.

Highsmith argumenta que un enfoque de desarrollo adaptativo basado en la colaboración es "tanto una fuente de *orden* en nuestras complejas interacciones, como de disciplina e ingeniería". Él define un "ciclo de vida" del DAS (véase la figura 3.3) que incorpora tres fases: especulación, colaboración y aprendizaje.

FIGURA 3.3

Desarrollo adaptativo de software



12 Consulte http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

Durante la *especulación*, se inicia el proyecto y se lleva a cabo la *planeación adaptativa del ciclo*. La especulación emplea la información de inicio del proyecto —enunciado de misión de los clientes, restricciones del proyecto (por ejemplo, fechas de entrega o descripciones de usuario) y requerimientos básicos— para definir el conjunto de ciclos de entrega (incrementos de software) que se requerirán para el proyecto.

La colaboración eficaz con el cliente sólo ocurrirá si evita cualquier actitud del tipo "nosotros y ellos".

No importa lo completo y previsor que sea el plan del ciclo, será inevitable que cambie. Con base en la información obtenida al terminar el primer ciclo, el plan se revisa y se ajusta, de modo que el trabajo planeado se acomode mejor a la realidad en la que trabaja el equipo DAS.

Las personas motivadas usan la *colaboración* de manera que multiplica su talento y producción creativa más allá de sus números absolutos. Este enfoque es un tema recurrente en todos los métodos ágiles. Sin embargo, la colaboración no es fácil. Incluye la comunicación y el trabajo en equipo, pero también resalta el individualismo porque la creatividad individual desempeña un papel importante en el pensamiento colaborativo. Es cuestión, sobre todo, de confianza. Las personas que trabajan juntas deben confiar una en otra a fin de: 1) criticarse sin enojo, 2) ayudarse sin resentimiento, 3) trabajar tan duro, o más, que como de costumbre, 4) tener el conjunto de aptitudes para contribuir al trabajo, y 5) comunicar los problemas o preocupaciones de manera que conduzcan a la acción efectiva.

El DAS pone el énfasis en el aprendizaje como elemento clave para lograr un equipo con "organización propia".

Conforme los miembros de un equipo DAS comienzan a desarrollar los componentes que forman parte de un ciclo adaptativo, el énfasis se traslada al "aprendizaje" de todo lo que hay en el avance hacia la terminación del ciclo. En realidad, Highsmith [Hig00] afirma que los desarrolladores de software sobreestiman con frecuencia su propia comprensión (de la tecnología, del proceso y del proyecto) y que el aprendizaje los ayudará a mejorar su nivel de entendimiento real. Los equipos DAS aprenden de tres maneras: grupos de enfoque (véase el capítulo 5), revisiones técnicas (véase el capítulo 14) y análisis post mórtem del proyecto.

La filosofía DAS tiene un mérito, sin importar el modelo de proceso que se use. El énfasis general que hace el DAS en la dinámica de los equipos con organización propia, la colaboración interpersonal y el aprendizaje individual y del equipo generan equipos para proyectos de software que tienen una probabilidad de éxito mucho mayor.

3.5.2 Scrum

Scrum (nombre que proviene de cierta jugada que tiene lugar durante un partido de rugby)[13] es un método de desarrollo ágil de software concebido por Jeff Sutherland y su equipo de desarrollo a principios de la década de 1990. En años recientes, Schwaber y Beedle [Sch01a] han desarrollado más los métodos Scrum.

WebRef

En la dirección **www.controlchaos.com** hay información útil sobre Scrum.

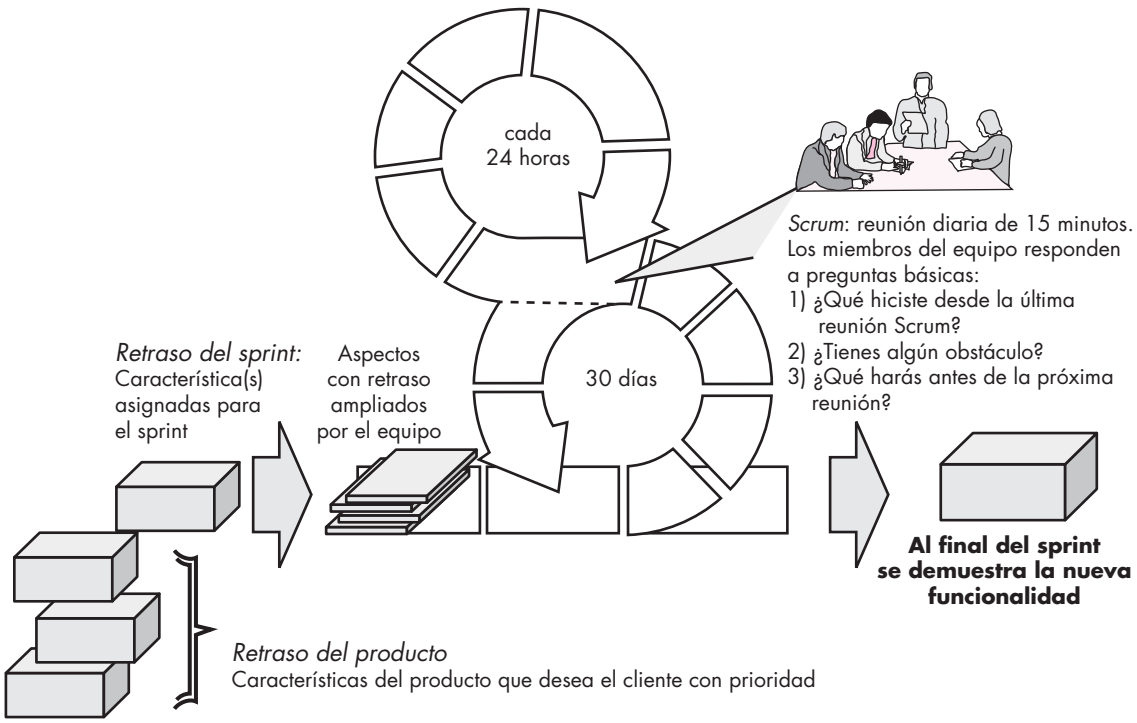
Los principios Scrum son congruentes con el manifiesto ágil y se utilizan para guiar actividades de desarrollo dentro de un proceso de análisis que incorpora las siguientes actividades estructurales: requerimientos, análisis, diseño, evolución y entrega. Dentro de cada actividad estructural, las tareas del trabajo ocurren con un patrón del proceso (que se estudia en el párrafo siguiente) llamado *sprint*. El trabajo realizado dentro de un sprint (el número de éstos que requiere cada actividad estructural variará en función de la complejidad y tamaño del producto) se adapta al problema en cuestión y se define —y con frecuencia se modifica— en tiempo real por parte del equipo Scrum. El flujo general del proceso Scrum se ilustra en la figura 3.4.

Scrum incorpora un conjunto de patrones del proceso que ponen el énfasis en las prioridades del proyecto, las unidades de trabajo agrupadas, la comunicación y la retroalimentación frecuente con el cliente.

Scrum acentúa el uso de un conjunto de patrones de proceso del software [Noy02] que han demostrado ser eficaces para proyectos con plazos de entrega muy apretados, requerimientos cambiantes y negocios críticos. Cada uno de estos patrones de proceso define un grupo de acciones de desarrollo:

13 Se forma un grupo de jugadores alrededor del balón y todos trabajan juntos (a veces con violencia) para moverlo a través del campo.

FIGURA 3.4
Flujo del proceso Scrum



Retraso: lista de prioridades de los requerimientos o características del proyecto que dan al cliente un valor del negocio. Es posible agregar en cualquier momento otros aspectos al retraso (ésta es la forma en la que se introducen los cambios). El gerente del proyecto evalúa el retraso y actualiza las prioridades según se requiera.

Sprints: consiste en unidades de trabajo que se necesitan para alcanzar un requerimiento definido en el retraso que debe ajustarse en una caja de tiempo[14] predefinida (lo común son 30 días). Durante el sprint no se introducen cambios (por ejemplo, aspectos del trabajo retrasado). Así, el sprint permite a los miembros del equipo trabajar en un ambiente de corto plazo pero estable.

Reuniones Scrum: son reuniones breves (de 15 minutos, por lo general) que el equipo Scrum efectúa a diario. Hay tres preguntas clave que se pide que respondan todos los miembros del equipo [Noy02]:

- ¿Qué hiciste desde la última reunión del equipo?
- ¿Qué obstáculos estás encontrando?
- ¿Qué planeas hacer mientras llega la siguiente reunión del equipo?

Un líder del equipo, llamado *maestro Scrum*, dirige la junta y evalúa las respuestas de cada persona. La junta Scrum ayuda al equipo a descubrir los problemas potenciales tan pronto como sea posible. Asimismo, estas juntas diarias llevan a la "socialización del conocimiento" [Bee99], con lo que se promueve una estructura de equipo con organización propia.

Demostraciones preliminares: entregar el incremento de software al cliente de modo que la funcionalidad que se haya implementado pueda demostrarse al cliente y éste pueda evaluarla.

14 Una *caja de tiempo* es un término de la administración de proyectos (véase la parte 4 de este libro) que indica el tiempo que se ha asignado para cumplir alguna tarea.

Es importante notar que las demostraciones preliminares no contienen toda la funcionalidad planeada, sino que éstas se entregarán dentro de la caja de tiempo establecida.

Beedle y sus colegas [Bee99] presentan un análisis exhaustivo de estos patrones en el que dicen: "Scrum supone de entrada la existencia de caos…" Los patrones de proceso Scrum permiten que un equipo de software trabaje con éxito en un mundo en el que es imposible eliminar la incertidumbre.

3.5.3 Método de desarrollo de sistemas dinámicos (MDSD)

WebRef

En la dirección **www.dsdm.org** hay recursos útiles para el MDSD.

El *método de desarrollo de sistemas dinámicos* (MDSD) [Sta97] es un enfoque de desarrollo ágil de software que "proporciona una estructura para construir y dar mantenimiento a sistemas que cumplan restricciones apretadas de tiempo mediante la realización de prototipos incrementales en un ambiente controlado de proyectos" [CCS02]. La filosofía MDSD está tomada de una versión modificada de la regla de Pareto: 80 por ciento de una aplicación puede entregarse en 20 por ciento del tiempo que tomaría entregarla completa (100 por ciento).

El MDSD es un proceso iterativo de software en el que cada iteración sigue la regla de 80 por ciento. Es decir, se requiere sólo suficiente trabajo para cada incremento con objeto de facilitar el paso al siguiente. Los detalles restantes se terminan más tarde, cuando se conocen los requerimientos del negocio y se han pedido y efectuado cambios.

El grupo DSDM Consortium (**www.dsdm.org**) es un conglomerado mundial de compañías que adoptan colectivamente el papel de "custodios" del método. El consorcio ha definido un modelo de proceso ágil, llamado *ciclo de vida MDSD*, que define tres ciclos iterativos distintos, precedidos de dos actividades adicionales al ciclo de vida:

Estudio de factibilidad: establece los requerimientos y restricciones básicas del negocio, asociados con la aplicación que se va a construir, para luego evaluar si la aplicación es un candidato viable para aplicarle el proceso MDSD.

PUNTO CLAVE

El MDSD es una estructura de proceso que adopta las tácticas de otro enfoque ágil, como XP.

Estudio del negocio: establece los requerimientos e información funcionales que permitirán la aplicación para dar valor al negocio; asimismo, define la arquitectura básica de la aplicación e identifica los requerimientos para darle mantenimiento.

Iteración del modelo funcional: produce un conjunto de prototipos incrementales que demuestran al cliente la funcionalidad. (Nota: todos los prototipos de MDSD están pensados para que evolucionen hacia la aplicación que se entrega.) El objetivo de este ciclo iterativo es recabar requerimientos adicionales por medio de la obtención de retroalimentación de los usuarios cuando practican con el prototipo.

Diseño e iteración de la construcción: revisita los prototipos construidos durante la *iteración del modelo funcional* a fin de garantizar que en cada iteración se ha hecho ingeniería en forma que permita dar valor operativo del negocio a los usuarios finales; la *iteración del modelo funcional* y el *diseño e iteración de la construcción* ocurren de manera concurrente.

Implementación: coloca el incremento más reciente del software (un prototipo "operacional") en el ambiente de operación. Debe notarse que: 1) el incremento tal vez no sea el de 100% final, o 2) quizá se pidan cambios cuando el incremento se ponga en su lugar. En cualquier caso, el trabajo de desarrollo MDSD continúa y vuelve a la actividad de iteración del modelo funcional.

El MDSD se combina con XP (véase la sección 3.4) para dar un enfoque de combinación que define un modelo sólido del proceso (ciclo de vida MDSD) con las prácticas detalladas (XP) que se requieren para elaborar incrementos de software. Además, los conceptos DAS se adaptan a un modelo combinado del proceso.

3.5.4 Cristal

Alistar Cockburn [Coc05] creó la *familia Cristal de métodos ágiles*[15] a fin de obtener un enfoque de desarrollo de software que premia la "maniobrabilidad" durante lo que Cockburn caracteriza como "un juego cooperativo con recursos limitados, de invención y comunicación, con el objetivo primario de entregar software útil que funcione y con la meta secundaria de plantear el siguiente juego" [Coc02].

PUNTO CLAVE

Cristal es una familia de modelos de proceso con el mismo "código genético" pero diferentes métodos para adaptarse a las características del proyecto.

Para lograr la maniobrabilidad, Cockburn y Highsmith definieron un conjunto de metodologías, cada una con elementos fundamentales comunes a todos, y roles, patrones de proceso, producto del trabajo y prácticas que son únicas para cada uno. La familia Cristal en realidad es un conjunto de ejemplos de procesos ágiles que han demostrado ser efectivos para diferentes tipos de proyectos. El objetivo es permitir que equipos ágiles seleccionen al miembro de la familia Cristal más apropiado para su proyecto y ambiente.

3.5.5 Desarrollo impulsado por las características (DIC)

El *desarrollo impulsado por las características* (DIC) lo concibió originalmente Peter Coad y sus colegas [Coa99] como modelo práctico de proceso para la ingeniería de software orientada a objetos. Stephen Palmer y John Felsing [Pal02] ampliaron y mejoraron el trabajo de Coad con la descripción de un proceso adaptativo y ágil aplicable a proyectos de software de tamaño moderado y grande.

WebRef

En la dirección **www.featuredrivendevelopment.com/** se encuentra una amplia variedad de artículos y presentaciones sobre el DIC.

Igual que otros proyectos ágiles, DIC adopta una filosofía que: 1) pone el énfasis en la colaboración entre los integrantes de un equipo DIC; 2) administra la complejidad de los problemas y del proyecto con el uso de la descomposición basada en las características, seguida de la integración de incrementos de software, y 3) comunica los detalles técnicos en forma verbal, gráfica y con medios basados en texto. El DIC pone el énfasis en las actividades de aseguramiento de la calidad del software mediante el estímulo de la estrategia de desarrollo incremental, el uso de inspecciones del diseño y del código, la aplicación de auditorías de aseguramiento de la calidad del software (véase el capítulo 16), el conjunto de mediciones y el uso de patrones (para el análisis, diseño y construcción).

En el contexto del DIC, una *característica* "es una función valiosa para el cliente que puede implementarse en dos semanas o menos" [Coa99]. El énfasis en la definición de características proporciona los beneficios siguientes:

- Debido a que las características son bloques pequeños de funcionalidad que se entrega, los usuarios las describen con más facilidad, entienden cómo se relacionan entre sí y las revisan mejor en busca de ambigüedades, errores u omisiones.
- Las características se organizan por jerarquía de grupos relacionados con el negocio.
- Como una característica es el incremento de software DIC que se entrega, el equipo desarrolla características operativas cada dos semanas.
- El diseño y representación en código de las características son más fáciles de inspeccionar con eficacia porque éstas son pequeñas.
- La planeación, programación de actividades y seguimiento son determinadas por la jerarquía de características, y no por un conjunto de tareas de ingeniería de software adoptadas en forma arbitraria.

Coad y sus colegas [Coa99] sugieren el esquema siguiente para definir una característica:

<acción> el **<resultado>** **<a|por|de|para>** un **<objeto>**

15 El nombre "cristal" se deriva de los cristales de minerales, cada uno de los cuales tiene propiedades específicas de color, forma y dureza.

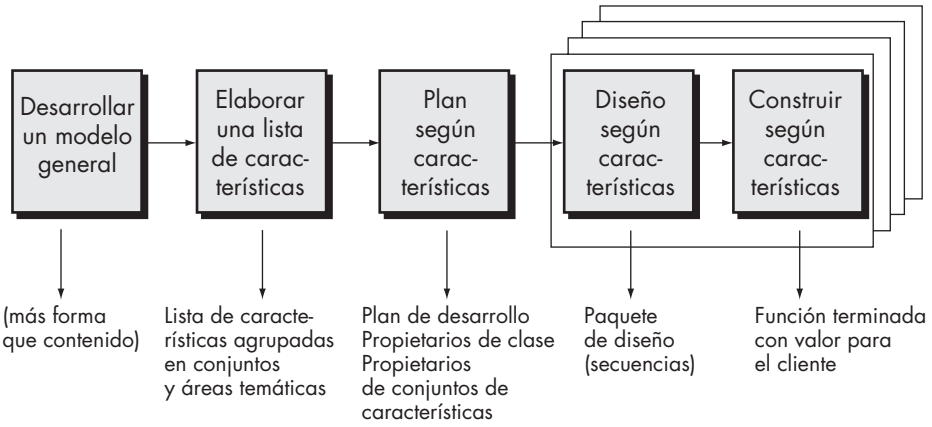


FIGURA 3.5
Desarrollo impulsado por las características [Coa99] (con permiso)

donde **<objeto>** es "una persona, lugar o cosa (incluso roles, momentos del tiempo o intervalos temporales, o descripciones parecidas a las entradas de un catálogo)". Algunos ejemplos de características para una aplicación de comercio electrónico son los siguientes:

Agregar el producto al carrito de compras

Mostrar las especificaciones técnicas del producto

Guardar la información de envío para el cliente

Un conjunto de características agrupa las que son similares en categorías relacionadas con el negocio y se define así:

<acción><ndo> un **<objeto>**

Por ejemplo: *Haciendo una venta del producto* es un conjunto de características que agruparía las que ya se mencionaron y otras más.

El enfoque DIC define cinco actividades estructurales "colaborativas" [Coa99] (en el enfoque DIC se llaman "procesos"), como se muestra en la figura 3.5.

El DIC pone más énfasis que otros métodos ágiles en los lineamientos y técnicas para la administración de proyectos. A medida que éstos aumentan su tamaño y complejidad, no es raro que la administración de proyectos *ad hoc* sea inadecuada. Para los desarrolladores, sus gerentes y otros participantes, es esencial entender el estado del proyecto, es decir, los avances realizados y los problemas que han surgido. Si la presión por cumplir el plazo de entrega es mucha, tiene importancia crítica determinar si la entrega de los incrementos del software está programada en forma adecuada. Para lograr esto, el DIC define seis puntos de referencia durante el diseño e implementación de una característica: "recorrido por el diseño, diseño, inspección del diseño, código, inspección del código, decisión de construir" [Coa99].

3.5.6 Desarrollo esbelto de software (DES)

El *desarrollo esbelto de software* (DES) adapta los principios de la manufactura esbelta al mundo de la ingeniería de software. Los principios de esbeltez que inspiran al proceso DES se resumen como sigue ([Pop03], [Pop06a]): *eliminar el desperdicio, generar calidad, crear conocimiento, aplazar el compromiso, entregar rápido, respetar a las personas* y *optimizar al todo*.

Es posible adaptar cada uno de estos principios al proceso del software. Por ejemplo, *eliminar el desperdicio* en el contexto de un proyecto de software ágil significa [Das05]: 1) no agregar características o funciones extrañas, 2) evaluar el costo y el efecto que tendrá en la programación de actividades cualquier nuevo requerimiento solicitado, 3) eliminar cualesquiera etapas superfluas del proceso, 4) establecer mecanismos para mejorar la forma en la que los miembros

del equipo obtienen información, 5) asegurar que las pruebas detecten tantos errores como sea posible, 6) reducir el tiempo requerido para pedir y obtener una decisión que afecta al software o al proceso que se aplica para crearlo, y 7) simplificar la manera en la que se transmite la información a todos los participantes involucrados en el proceso.

Para un análisis detallado del DES y para conocer lineamientos prácticos a fin de implementar el proceso, debe consultarse [Pop06a] y [Pop06b].

3.5.7 Modelado ágil (MA)

WebRef

En la dirección **www.agilemodeling.com** hay información amplia sobre el modelado ágil.

Hay muchas situaciones en las que los ingenieros de software deben construir sistemas grandes de importancia crítica para el negocio. El alcance y complejidad de tales sistemas debe modelarse de modo que: 1) todos los actores entiendan mejor cuáles son las necesidades que deben satisfacerse, 2) el problema pueda dividirse con eficacia entre las personas que deben resolverlo, y 3) se asegure la calidad a medida que se hace la ingeniería y se construye el sistema.

En los últimos 30 años se ha propuesto una gran variedad de métodos de modelado y notación para la ingeniería de software con objeto de hacer el análisis y el diseño (tanto en la arquitectura como en los componentes). Estos métodos tienen su mérito, pero se ha demostrado que son difíciles de aplicar y sostener (en muchos proyectos). Parte del problema es el "peso" de dichos métodos de modelación. Con esto se hace referencia al volumen de la notación que se requiere, al grado de formalismo sugerido, al tamaño absoluto de los modelos para proyectos grandes y a la dificultad de mantener el(los) modelo(s) conforme suceden los cambios. Sin embargo, el análisis y el modelado del diseño tienen muchos beneficios para los proyectos grandes, aunque no fuera más que porque hacen a esos proyectos intelectualmente más manejables. ¿Hay algún enfoque ágil para el modelado de la ingeniería de software que brinde una alternativa?

Cita:

"El otro día fui a la farmacia por una medicina para el resfriado… no fue fácil. Había toda una pared cubierta de productos. Al recorrerla vi uno que era de acción rápida, pero otro que era de larga duración… ¿Qué es más importante, el presente o el futuro?"

Jerry Seinfeld

En el "sitio oficial de modelado ágil", Scott Ambler [Amb02a] describe el *modelado ágil* (MA) del modo siguiente:

> El modelado ágil (MA) es una metodología basada en la práctica para modelar y documentar con eficacia los sistemas basados en software. En pocas palabras, es un conjunto de valores, principios y prácticas para hacer modelos de software aplicables de manera eficaz y ligera a un proyecto de desarrollo de software. Los modelos ágiles son más eficaces que los tradicionales porque son sólo buenos, sin pretender ser perfectos.

El modelado ágil adopta todos los valores del manifiesto ágil. La filosofía de modelado ágil afirma que un equipo ágil debe tener la valentía para tomar decisiones que impliquen rechazar un diseño y reconstruirlo. El equipo también debe tener la humildad de reconocer que los tecnólogos no tienen todas las respuestas y que los expertos en el negocio y otros participantes deben ser respetados e incluidos.

Aunque el MA sugiere una amplia variedad de principios de modelado "fundamentales" y "suplementarios", aquellos que son exclusivos del MA son los siguientes [Amb02a]:

Modelo con un propósito. Un desarrollador que use el MA debe tener en mente una meta específica (por ejemplo, comunicar información al cliente o ayudarlo a entender mejor algún aspecto del software) antes de crear el modelo. Una vez identificada la meta para el modelo, el tipo y nivel de detalle de la notación por usar serán más obvios.

Uso de modelos múltiples. Hay muchos modelos y notaciones diferentes que pueden usarse para describir el software. Para la mayoría de proyectos sólo es esencial un pequeño subconjunto. El MA sugiere que para dar la perspectiva necesaria, cada modelo debe presentar un diferente aspecto del sistema y que sólo deben utilizarse aquellos modelos que den valor al público al que se dirigen.

"Viajar ligero" es una filosofía apropiada para todo el trabajo de ingeniería de software. Construir sólo aquellos modelos que agreguen valor… ni más ni menos.

Viajar ligero. Conforme avanza el trabajo de ingeniería de software, conserve sólo aquellos modelos que agreguen valor a largo plazo y elimine los demás. Todo producto del trabajo que se conserve debe recibir mantenimiento cuando haya cambios. Esto representa una labor que hace lento al equipo. Ambler [Amb02a] afirma que "cada vez que se decide conservar un modelo, se pierde agilidad en nombre de la conveniencia de tener disponible esa información en forma abstracta para el equipo (y de ese modo mejorar potencialmente la comunicación dentro del equipo, así como con los participantes)".

El contenido es más importante que la representación. El modelado debe transmitir información al público al que se dirige. Un modelo con sintaxis perfecta que transmita poco contenido útil no es tan valioso como otro que tenga notación defectuosa, pero que, no obstante, provea contenido de valor para los usuarios.

Conocer los modelos y herramientas que se utilizan en su creación. Entender las fortalezas y debilidades de cada modelo y las herramientas que se emplean para crearlos.

Adaptación local. El enfoque de modelado debe adaptarse a las necesidades del equipo ágil.

Un segmento importante de la comunidad de ingeniería de software ha adoptado el lenguaje de unificado de modelado (UML, por sus siglas en inglés)[16] como el método preferido para representar modelos del análisis y del diseño. El proceso unificado (véase el capítulo 2) fue desarrollado para proveer una estructura para la aplicación del UML. Scott Ambler [Amb06] desarrolló una versión simplificada del PU que integra su filosofía de modelado ágil.

3.5.8 El proceso unificado ágil (PUA)

El *proceso unificado ágil* (PUA) adopta una filosofía "en serie para lo grande" e "iterativa para lo pequeño" [Amb06] a fin de construir sistemas basados en computadora. Al adoptar las actividades en fase clásicas del PU —*concepción, elaboración, construcción* y *transición*—, el PUA brinda un revestimiento en serie (por ejemplo, una secuencia lineal de actividades de ingeniería de software) que permite que el equipo visualice el flujo general del proceso de un proyecto de software. Sin embargo, dentro de cada actividad, el equipo repite con objeto de alcanzar la agilidad y entregar tan rápido como sea posible incrementos de software significativos a los usuarios finales. Cada iteración del PUA aborda las actividades siguientes [Amb06]:

- *Modelado.* Se crean representaciones de UML de los dominios del negocio y el problema. No obstante, para conservar la agilidad, estos modelos deben ser "sólo suficientemente buenos" [Amb06] para permitir que el equipo avance.
- *Implementación.* Los modelos se traducen a código fuente.
- *Pruebas*. Igual que con la XP, el equipo diseña y ejecuta una serie de pruebas para detectar errores y garantizar que el código fuente cumple sus requerimientos.
- *Despliegue.* Como en la actividad general del proceso que se estudió en los capítulos 1 y 2, el despliegue en este contexto se centra en la entrega de un incremento de software y en la obtención de retroalimentación de los usuarios finales.
- *Configuración y administración del proyecto*. En el contexto del PUA, la administración de la configuración (véase el capítulo 22) incluye la administración del cambio y el riesgo, y el control de cualesquiera productos del trabajo persistentes[17] que produzca el equipo.

16 En el apéndice 1 se presenta un método breve para aprender UML.

17 Un *producto del trabajo persistente* es un modelo o documento o caso de prueba producido por el equipo y que se conservará durante un periodo indeterminado. *No* se eliminará una vez entregado el incremento de software.

La administración del proyecto da seguimiento y controla el avance del equipo y coordina sus actividades.

- *Administración del ambiente.* La administración del ambiente coordina una infraestructura del proceso que incluye estándares, herramientas y otra tecnología de apoyo de la que dispone el equipo.

Aunque el PUA tiene conexiones históricas y técnicas con el lenguaje unificado de modelado, es importante observar que el modelado UML puede usarse junto con cualesquiera de los modelos de proceso ágil descritos en la sección 3.5.

HERRAMIENTAS DE SOFTWARE

Desarrollo ágil

Objetivo: El objetivo de las herramientas de desarrollo ágil es ayudar en uno o más aspectos de éste, con énfasis en facilitar la elaboración rápida de software funcional. Estas herramientas también pueden emplearse cuando se aplican modelos de proceso prescriptivo (véase el capítulo 2).

Mecánica: Las herramientas de mecánica varían. En general, las herramientas ágiles incluyen el apoyo automatizado para la planeación del proyecto, el desarrollo de casos y la obtención de requerimientos, el diseño rápido, la generación de código y la realización de pruebas.

Herramientas representativas:[18]

Nota: Debido a que el desarrollo ágil es un tema de moda, la mayoría de los vendedores de herramientas de software tratan de colocar herramientas que lo apoyan. Las que se mencionan a continuación tienen características que las hacen particularmente útiles para los proyectos ágiles.

OnTime, desarrollada por Axosoft (**www.axosoft.com**), presta apoyo a la administración de un proceso ágil para distintas actividades técnicas dentro del proceso.

Ideogramic UML, desarrollada por Ideogramic (**www.ideogramic.com**), es un conjunto de herramientas UML desarrolladas específicamente para usarlas dentro de un proceso ágil.

Together Tool Set, distribuido por Borland (**www.borland.com**), proporciona un grupo de herramientas para apoyar muchas actividades técnicas dentro de XP y otros procesos ágiles.

3.6 CONJUNTO DE HERRAMIENTAS PARA EL PROCESO ÁGIL

El "conjunto de herramientas" que da apoyo a los procesos ágiles se centra más en aspectos de la persona que en los de la tecnología.

Algunos defensores de la filosofía ágil afirman que las herramientas automatizadas de software (por ejemplo, las de diseño) deben verse como un complemento menor de las actividades del equipo, y no como algo fundamental para el éxito. Sin embargo, Alistair Cockburn [Coc04] sugiere que las herramientas tienen un beneficio y que "los equipos ágiles favorecen el uso de herramientas que permiten el flujo rápido de entendimiento. Algunas de estas herramientas son sociales y comienzan incluso en la etapa de reclutamiento. Otras son tecnológicas y ayudan a que los equipos distribuidos simulen su presencia física. Muchas herramientas son físicas y permiten que las personas las manipulen en talleres".

Prácticamente todos los modelos de proceso ágil son elementos clave en la contratación del personal adecuado (reclutamiento), la colaboración en equipo, la comunicación con los participantes y la administración indirecta; por eso, Cockburn afirma que las "herramientas" que se abocan a dichos aspectos son factores críticos para el éxito de la agilidad. Por ejemplo, una "herramienta" de reclutamiento tal vez sea el requerimiento de que un prospecto a miembro del equipo pase algunas horas programando en pareja con alguien que ya es integrante del equipo. El "ajuste" se evalúa de inmediato.

Las "herramientas" de colaboración y comunicación por lo general son de baja tecnología e incorporan cualquier mecanismo ("proximidad física, pizarrones, tableros, tarjetas y notas ad-

18 Las herramientas mencionadas aquí no son obligatorias, sólo son una muestra en esta categoría. En la mayoría de casos, sus nombres son marcas registradas por sus respectivos desarrolladores.

heribles" [Coc04] que provea información y coordinación entre los desarrolladores ágiles. La comunicación activa se logra por medio de la dinámica del equipo (por ejemplo, la programación en parejas), mientras que la comunicación pasiva se consigue con "radiadores de información" (un tablero que muestre el estado general de de los distintos componentes de un incremento). Las herramientas de administración de proyectos no ponen el énfasis en la gráfica de Gantt y la sustituyen con otras de valor agregado o "gráficas de pruebas creadas *versus* pasadas; otras herramientas ágiles se utilizan para optimizar el ambiente en el que trabaja el equipo ágil (por ejemplo, áreas más eficientes para reunirse), mejoran la cultura del equipo por medio de cultivar las interacciones sociales (equipos con algo en común), dispositivos físicos (pizarrones electrónicos) y el mejoramiento del proceso (por ejemplo, la programación por parejas o la caja de tiempo)" [Coc04].

¿Algunas de las mencionadas son en verdad herramientas? Sí, lo son, si facilitan el trabajo efectuado por un miembro del equipo ágil y mejoran la calidad del producto final.

3.7 Resumen

En una economía moderna, las condiciones del mercado cambian con rapidez, los clientes y usuarios finales necesitan evolucionar y surgen nuevas amenazas competitivas sin aviso previo. Los profesionales deben enfocar la ingeniería de software en forma que les permita mantenerse ágiles para definir procesos maniobrables, adaptativos y esbeltos que satisfagan las necesidades de los negocios modernos.

Una filosofía ágil para la ingeniería de software pone el énfasis en cuatro aspectos clave: la importancia de los equipos con organización propia que tienen el control sobre el trabajo que realizan, la comunicación y colaboración entre los miembros del equipo y entre los profesionales y sus clientes, el reconocimiento de que el cambio representa una oportunidad y la insistencia en la entrega rápida de software que satisfaga al consumidor. Los modelos de proceso ágil han sido diseñados para abordar cada uno de estos aspectos.

La programación extrema (XP) es el proceso ágil de más uso. Organizada con cuatro actividades estructurales: planeación, diseño, codificación y pruebas, la XP sugiere cierto número de técnicas innovadoras y poderosas que permiten a un equipo ágil generar entregas frecuentes de software que posee características y funcionalidad que han sido descritas y clasificadas según su prioridad por los participantes.

Otros modelos de proceso ágil también insisten en la colaboración humana y en la organización propia del equipo, pero definen sus actividades estructurales y seleccionan diferentes puntos de importancia. Por ejemplo, el DAS utiliza un proceso iterativo que incluye un ciclo de planeación adaptativa, métodos relativamente rigurosos para recabar requerimientos, y un ciclo de desarrollo iterativo que incorpora grupos de consumidores y revisiones técnicas formales como mecanismos de retroalimentación en tiempo real. El Scrum pone el énfasis en el uso de un conjunto de patrones de software que han demostrado ser eficaces para proyectos que tienen plazos de entrega apretados, requerimientos cambiantes o que se emplean en negocios críticos. Cada patrón de proceso define un conjunto de tareas de desarrollo y permite al equipo Scrum construir un proceso que se adapte a las necesidades del proyecto. El método de desarrollo de sistemas dinámicos (MDSD) resalta el uso de la programación con caja de tiempo y sugiere que en cada incremento de software sólo se requiere el trabajo suficiente que facilite el paso al incremento que sigue. Cristal es una familia de modelos de proceso ágil que se adaptan a las características específicas del proyecto.

El desarrollo impulsado por las características (DIC) es algo más "formal" que otros métodos ágiles, pero conserva su agilidad al centrar al equipo del proyecto en el desarrollo de características, funciones valiosas para el cliente que pueden implementarse en dos semanas o menos. El

desarrollo esbelto de software (DES) ha adaptado los principios de la manufactura esbelta al mundo de la ingeniería de software. El modelado ágil (MA) sugiere que el modelado es esencial para todos los sistemas, pero que la complejidad, tipo y tamaño del modelo deben ajustarse al software que se va a elaborar. El proceso unificado ágil (PUA) adopta una filosofía "serial en lo grande" e "iterativo en lo pequeño" para la elaboración de software.

Problemas y puntos por evaluar

3.1. Vuelva a leer el "Manifiesto para el desarrollo ágil de software" al principio de este capítulo. ¿Puede pensar en una situación en la que uno o más de los cuatro "valores" pudieran causar problemas al equipo de software?

3.2. Describa con sus propias palabras la *agilidad* (para proyectos de software).

3.3. ¿Por qué un proceso iterativo hace más fácil administrar el cambio? ¿Es iterativo todo proceso ágil analizado en este capítulo? ¿Es posible terminar un proyecto en sólo una iteración y aún así conseguir que sea ágil? Explique sus respuestas.

3.4. ¿Podría describirse cada uno de los procesos ágiles con el uso de las actividades estructurales generales mencionadas en el capítulo 2? Construya una tabla que mapee las actividades generales en las actividades definidas para cada proceso ágil.

3.5. Proponga un "principio de agilidad" más que ayudaría al equipo de ingeniería de software a ser aún más maniobrable.

3.6. Seleccione un principio de agilidad mencionado en la sección 3.3.1 y trate de determinar si está incluido en cada uno de los modelos de proceso presentados en este capítulo. [Nota: sólo se presentó el panorama general de estos modelos de proceso, por lo que tal vez no fuera posible determinar si un principio está incluido en uno o más de ellos, a menos que el lector hiciera una investigación (lo que no se requiere para este problema)].

3.7. ¿Por qué cambian tanto los requerimientos? Después de todo, ¿la gente no sabe lo que quiere?

3.8. La mayoría de modelos de proceso ágil recomiendan la comunicación cara a cara. No obstante, los miembros del equipo de software y sus clientes tal vez estén alejados geográficamente. ¿Piensa usted que esto implica que debe evitarse la separación geográfica? ¿Se le ocurren formas de resolver este problema?

3.9. Escriba una historia de usuario XP que describa la característica de "lugares favoritos" o "marcadores" disponible en la mayoría de navegadores web.

3.10. ¿Qué es una solución en punta en XP?

3.11. Describa con sus propias palabras los conceptos de rediseño y programación en parejas de XP.

3.12. Haga otras lecturas y describa lo que es una caja de tiempo. ¿Cómo ayuda a un equipo DAS para que entregue incrementos de software en un corto periodo?

3.13. ¿Se logra el mismo resultado con la regla de 80% del MDSD y con el enfoque de la caja de tiempo del DAS?

3.14. Con el formato de patrón de proceso presentado en el capítulo 2, desarrolle uno para cualquiera de los patrones Scrum presentados en la sección 3.5.2.

3.15. ¿Por qué se le llama a Cristal familia de métodos ágiles?

3.16. Con el formato de característica DIC descrito en la sección 3.5.5, defina un conjunto de características para un navegador web. Luego desarrolle un conjunto de características para el primer conjunto.

3.17. Visite el sitio oficial de modelación ágil y elabore la lista completa de todos los principios fundamentales y secundarios del MA.

3.18. El conjunto de herramientas propuestas en la sección 3.6 da apoyo a muchos de los aspectos "suaves" de los métodos ágiles. Debido a que la comunicación es tan importante, recomiende un conjunto de herramientas reales que podría utilizarse para que los participantes de un equipo ágil se comuniquen mejor.

Lecturas adicionales y fuentes de información

La filosofía general y principios que subyacen al desarrollo de software ágil se estudian a profundidad en muchos de los libros mencionados a lo largo de este capítulo. Además, los textos de Shaw y Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Construction*, Springer, 2005) y Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) presentan análisis útiles del tema. Aguanno (*Managing Agile Projects*, Multi-Media Publications, 2005), Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004) y Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) presentan el punto de vista de la administración y consideran ciertos aspectos de la administración de proyectos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) expone una encuesta acerca de los principios, procesos y prácticas ágiles. Booch y sus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004) hacen un análisis fructífero del delicado equilibrio entre agilidad y disciplina.

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) explica los principios, patrones y prácticas que se requieren para desarrollar "código limpio" en un ambiente de ingeniería de software ágil. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) estudia estrategias para ampliar las prácticas ágiles en proyectos grandes. Lippert y Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Succesfully*, Wiley, 2006) analizan el uso del rediseño cuando se aplica a sistemas grandes y complejos. Stamelos y Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) analizan las técnicas SQA que forman la filosofía ágil.

En la última década se han escrito decenas de libros sobre programación extrema. Beck (*Extreme Programming Explained: Embrace Change*, 2a. ed., Addison-Wesley, 2004) sigue siendo la referencia definitiva al respecto. Además, Jeffries y sus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi y Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk y Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001) y Auer y sus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) hacen un análisis detallado de XP y dan una guía para aplicarla de la mejor forma. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) adopta un enfoque crítico sobre XP, y define cuándo y dónde es apropiada. Un estudio profundo de la programación por parejas se presenta en McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Highsmith [Hig00] analiza con detalle el DAS. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) estudia el empleo de Scrum para proyectos que tienen un efecto grande en los negocios. Los detalles de Scrum los estudian Schwaber y Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Algunos tratamientos útiles del MDSD han sido escritos por DSDM Consortium (*DSDM: Business Focused Development*, 2a. ed., Pearson Education, 2003) y Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystal Clear*, Addison-Wesley, 2005) presenta un panorama excelente de la familia de procesos Cristal. Palmer y Felsing [Pal02] dan un tratamiento detallado del DIC. Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) proporcionan otro análisis útil del DIC, que incluye un recorrido, paso a paso, por la mecánica del proceso. Poppendieck y Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) dan lineamientos para la administración y el control de proyectos ágiles. Ambler y Jeffries (*Agile Modeling*, Wiley, 2002) estudian el MA con cierta profundidad.

En internet existe una amplia variedad de fuentes de información sobre el desarrollo de software ágil. En el sitio web del libro hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso ágil, en la dirección: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PARTE

Dos

Modelado

En esta parte de la obra, aprenderá sobre los principios, conceptos y métodos que se usan para crear requerimientos de alta calidad y para diseñar modelos. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Qué conceptos y principios guían la práctica de la ingeniería de software?
- ¿Qué son los requerimientos de ingeniería y cuáles son los conceptos subyacentes que llevan a un buen análisis de requerimientos?
- ¿Cómo se crean los requerimientos del modelo y cuáles son sus elementos?
- ¿Cuáles son los elementos de un buen diseño?
- ¿Cómo establece el diseño de la arquitectura una estructura para todas las demás acciones de diseño y qué modelos se utilizan?
- ¿Cómo se diseñan componentes de software de alta calidad?
- ¿Qué conceptos, modelos y métodos se aplican al diseñar una interfaz de usuario?
- ¿Qué es el diseño basado en patrones?
- ¿Qué estrategias y métodos especializados se emplean para diseñar *webapps*?

Una vez que se respondan estas preguntas, el lector estará mejor preparado para aplicar en la práctica la ingeniería de software.

CAPÍTULO

4 Principios que guían la práctica

Conceptos clave

En un libro que explora las vidas y pensamientos de los ingenieros de software, Ellen Ullman [Ull97] ilustra una parte de su vida con el relato de lo que piensa un profesional del software cuando está bajo presión:

> No tengo idea de la hora que es. En esta oficina no hay ventanas ni reloj, sólo la pantalla de un horno de microondas que parpadea su LED de color rojo: 12:00, 12:00, 12:00. Joel y yo hemos estado programando durante varios días. Tenemos una falla, endemoniada y testaruda. Así que nos sentimos bien con el pulso rojo sin tiempo, como si fuera un pasmo de nuestros cerebros, de algún modo sincronizados al mismo ritmo del parpadeo...
>
> ¿En qué estamos trabajando? Los detalles se me escapan. Tal vez ayudamos a personas pobres y enfermas o mejoramos un conjunto de rutinas de bajo nivel de un protocolo de base de datos distribuida, no me importa. Debería importarme; en otra parte de mi ser —más tarde, quizá cuando salga de este cuarto lleno de computadoras— me preocuparé mucho de por qué y para quién y con qué propósito estoy escribiendo software. Pero ahora, no. He cruzado una membrana tras la que el mundo real y sus asuntos ya no importan. Soy ingeniera de software.

La anterior es una imagen tenebrosa de la práctica de la ingeniería de software, pero si se detienen un poco a pensarlo, muchos de los lectores de este libro se verán reflejados en ella.

Las personas que elaboran software de cómputo practican el arte, artesanía o disciplina[1] conocida como ingeniería de software. Pero, ¿qué es la "práctica" de la ingeniería de software? En un sentido general, es un conjunto de conceptos, principios, métodos y herramientas a los que un ingeniero de software recurre en forma cotidiana. La práctica permite que los gerentes

Una mirada rápida

¿Qué es? La práctica de la ingeniería de software es un conjunto amplio de principios, conceptos, métodos y herramientas que deben considerarse al planear y desarrollar software.

¿Quién lo hace? Los profesionales (ingenieros de software) y sus gerentes realizan varias tareas de ingeniería de software.

¿Por qué es importante? El proceso de software proporciona a todos los involucrados en la creación de un sistema o producto basado en computadora un mapa para llegar con éxito al destino. La práctica proporciona los detalles que se necesitarán para circular por la carretera. Indica dónde se localizan los puentes, los caminos cerrados y las bifurcaciones. Ayuda a entender los conceptos y principios que deben entenderse y seguirse a fin de llegar con seguridad y rapidez. Enseña a manejar, dónde disminuir la velocidad y en qué lugares acelerar. En el contexto de la ingeniería de software, la práctica es lo que se hace día tras día conforme el software evoluciona de idea a realidad.

¿Cuáles son los pasos? Son tres los elementos de la práctica que se aplican sin importar el modelo de proceso que se elija. Se trata de: principios, conceptos y métodos. Un cuarto elemento de la práctica —las herramientas— da apoyo a la aplicación de los métodos.

¿Cuál es el producto final? La práctica incluye las actividades técnicas que generan todos los productos del trabajo definidos por el modelo del proceso de software que se haya escogido.

¿Cómo me aseguro de que lo hice bien? En primer lugar, hay que tener una comprensión sólida de los principios que se aplican al trabajo (por ejemplo, el diseño) en cuestión. Después, asegúrese de que se escogió el método apropiado para el trabajo, use herramientas automatizadas cuando sean adecuadas para la tarea y sea firme respecto de la necesidad de técnicas de aseguramiento de la calidad de los productos finales que se generen.

1 Algunos escritores afirman que cualquiera de estos términos excluye a los otros. En realidad, la ingeniería de software es las tres cosas.

administren proyectos de software y que los ingenieros de software elaboren programas de cómputo. La práctica da al modelo del proceso de software el saber técnico y administrativo para realizar el trabajo. La práctica transforma un enfoque caprichoso y disperso en algo más organizado, más eficaz y con mayor probabilidad de alcanzar el éxito.

A lo largo de lo que resta del libro se estudiarán distintos aspectos de la práctica de la ingeniería de software. En este capítulo, la atención se pone en los principios y conceptos que la guían en lo general.

4.1 Conocimiento de la ingeniería de software

En un editorial publicado hace diez años en *IEEE Software*, Steve McConnell [McC99] hizo el siguiente comentario:

> Muchos trabajadores del software piensan que el conocimiento de la ingeniería de software casi exclusivamente consiste en tecnologías específicas: Java, Perl, html, C++, Linux, Windows NT, etc. Para programar computadoras es necesario conocer los detalles tecnológicos específicos. Si alguien pide al lector que escriba un programa en C++, tiene que saber algo sobre este lenguaje a fin de que el programa funcione.
>
> Es frecuente escuchar que el conocimiento del desarrollo de software tiene una vida media de tres años, lo que significa que la mitad de lo que es necesario saber hoy será obsoleto dentro de tres años. En el dominio del conocimiento relacionado con la tecnología es probable que eso se cumpla. Pero hay otra clase de conocimiento de desarrollo de software —algo que el autor considera como los "principios de la ingeniería de software"— que no tiene una vida media de tres años. Es factible que dichos principios sirvan al programador profesional durante toda su carrera.

McConnell continúa y plantea que el cuerpo de conocimientos de la ingeniería de software (alrededor del año 2000) ha evolucionado para convertirse en un "núcleo estable" que representa cerca de "75% del conocimiento necesario para desarrollar un sistema complejo". Pero, ¿qué es lo que hay dentro de ese núcleo estable?

Como dice McConnell, los principios fundamentales —ideas elementales que guían a los ingenieros de software en el trabajo que realizan— dan ahora un fundamento a partir del cual pueden aplicarse y evaluarse los modelos, métodos y herramientas de ingeniería.

4.2 Principios fundamentales

Cita:

"En teoría no hay diferencia entre la teoría y la práctica. Pero en la práctica sí la hay."

Jan van de Snepscheut

La práctica de la ingeniería de software está guiada por un conjunto de principios fundamentales que ayudan en la aplicación del proceso de software significativo y en la ejecución de métodos eficaces de ingeniería de software. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando realiza actividades estructurales y actividades sombrilla, cuando navega por el flujo del proceso y elabora un conjunto de productos del trabajo de la ingeniería de software. En el nivel de la práctica, los principios fundamentales definen un conjunto de valores y reglas que sirven como guía cuando se analiza un problema, se diseña una solución, se implementa y prueba ésta y cuando, al final, se entrega el software a la comunidad de usuarios.

En el capítulo 1 se identificó un conjunto de principios generales que amplían el proceso y práctica de la ingeniería de software: 1) agregar valor para los usuarios finales, 2) mantenerlo sencillo, 3) fijar la visión (del producto y el proyecto), 4) reconocer que otros consumen (y deben entender) lo que usted produce, 5) abrirse al futuro, 6) planear la reutilización y 7) ¡pensar! Aunque estos principios generales son importantes, se caracterizan en un nivel tan alto de abstracción que en ocasiones son difíciles de traducir en la práctica cotidiana de la ingeniería de

software. En las subsecciones que siguen se analizan con más detalle los principios fundamentales que guían el proceso y la práctica.

4.2.1 Principios que guían el proceso

En la parte 1 de este libro se estudia la importancia del proceso de software y se describen los abundantes modelos de proceso que se han propuesto para hacer el trabajo de ingeniería de software. Sin que importe que un modelo sea lineal o iterativo, prescriptivo o ágil, puede caracterizarse con el empleo de la estructura general del proceso aplicable a todos los modelos de proceso. Los siguientes principios fundamentales se aplican a la estructura y, por extensión, a todo proceso de software:

Todo proyecto y equipo son únicos. Esto significa que debe adaptar el proceso para que se ajuste mejor a sus necesidades.

Principio 1. ***Ser ágil.*** Ya sea que el modelo de proceso que se elija sea prescriptivo o ágil, son los principios básicos del desarrollo ágil los que deben gobernar el enfoque. Todo aspecto del trabajo que se haga debe poner el énfasis en la economía de acción: en mantener el enfoque técnico tan sencillo como sea posible, hacer los productos del trabajo que se generan tan concisos como se pueda y tomar las decisiones localmente, siempre que sea posible.

Principio 2. ***En cada etapa, centrarse en la calidad.*** La condición de salida para toda actividad, acción y tarea del proceso debe centrarse en la calidad del producto del trabajo que se ha generado.

Principio 3. ***Estar listo para adaptar.*** El proceso no es una experiencia religiosa, en él no hay lugar para el dogma. Cuando sea necesario, adapte su enfoque a las restricciones impuestas por el problema, la gente y el proyecto en sí.

Principio 4. ***Formar un equipo eficaz.*** El proceso y práctica de la ingeniería de software son importantes, pero el objetivo son las personas. Forme un equipo con organización propia en el que haya confianza y respeto mutuos.

Principio 5. ***Establecer mecanismos para la comunicación y coordinación.*** Los proyectos fallan porque la información importante cae en las grietas o porque los participantes no coordinan sus esfuerzos para crear un producto final exitoso. Éstos son aspectos de la administración que deben enfrentarse.

Cita:

"La verdad es que siempre se sabe lo que es correcto hacer. La parte difícil es hacerlo."

General H. Norman Schwarzkopf

Principio 6. ***Administrar el cambio.*** El enfoque puede ser formal o informal, pero deben establecerse mecanismos para administrar la forma en la que los cambios se solicitan, evalúan, aprueban e implementan.

Principio 7. ***Evaluar el riesgo.*** Son muchas las cosas que pueden salir mal cuando se desarrolla software. Es esencial establecer planes de contingencia.

Principio 8. ***Crear productos del trabajo que agreguen valor para otros.*** Sólo genere aquellos productos del trabajo que agreguen valor para otras actividades, acciones o tareas del proceso. Todo producto del trabajo que se genere como parte de la práctica de ingeniería de software pasará a alguien más. La lista de las funciones y características requeridas se dará a la persona (o personas) que desarrollará(n) un diseño, el diseño pasará a quienes generan código y así sucesivamente. Asegúrese de que el producto del trabajo imparte la información necesaria sin ambigüedades u omisiones.

La parte 4 de este libro se centra en aspectos de la administración del proyecto y del proceso, y analiza en detalle varios aspectos de cada uno de dichos principios.

4.2.2 Principios que guían la práctica

La práctica de la ingeniería de software tiene un solo objetivo general: entregar a tiempo software operativo de alta calidad que contenga funciones y características que satisfagan las ne-

cesidades de todos los participantes. Para lograrlo, debe adoptarse un conjunto de principios fundamentales que guíen el trabajo técnico. Estos principios tienen mérito sin que importen los métodos de análisis y diseño que se apliquen, ni las técnicas de construcción (por ejemplo, el lenguaje de programación o las herramientas automatizadas) que se usen o el enfoque de verificación y validación que se elija. Los siguientes principios fundamentales son vitales para la práctica de la ingeniería de software:

Los problemas son más fáciles de resolver cuando se subdividen en entidades separadas, distintas entre sí, solucionables individualmente y verificables.

Principio 1. ***Divide y vencerás.*** Dicho en forma más técnica, el análisis y el diseño siempre deben enfatizar la *separación de entidades* (SdE). Un problema grande es más fácil de resolver si se divide en un conjunto de elementos (o *entidades*). Lo ideal es que cada entidad entregue funcionalidad distinta que pueda desarrollarse, y en ciertos casos validarse, independientemente de otras entidades.

Principio 2. ***Entender el uso de la abstracción.*** En su parte medular, una abstracción es una simplificación de algún elemento complejo de un sistema usado para comunicar significado en una sola frase. Cuando se usa la abstracción *hoja de cálculo*, se supone que se comprende lo que es una hoja de cálculo, la estructura general de contenido que presenta y las funciones comunes que se aplican a ella. En la práctica de la ingeniería de software, se usan muchos niveles diferentes de abstracción, cada uno de los cuales imparte o implica significado que debe comunicarse. En el trabajo de análisis y diseño, un equipo de software normalmente comienza con modelos que representan niveles elevados de abstracción (por ejemplo, una hoja de cálculo) y poco a poco los refina en niveles más bajos de abstracción (como una *columna* o la función *SUM*).

Joel Spolsky [Spo02] sugiere que "todas las abstracciones no triviales hasta cierto punto son esquivas". El objetivo de una abstracción es eliminar la necesidad de comunicar detalles. Pero, en ocasiones, los efectos problemáticos precipitados por estos detalles se "filtran" por todas partes. Sin la comprensión de los detalles, no puede diagnosticarse con facilidad la causa de un problema.

Principio 3. ***Buscar la coherencia.*** Ya sea que se esté creando un modelo de los requerimientos, se desarrolle un diseño de software, se genere código fuente o se elaboren casos de prueba, el principio de coherencia sugiere que un contexto familiar hace que el software sea más fácil de usar. Como ejemplo, considere el diseño de una interfaz de usuario para una *webapp*. La colocación consistente de opciones de menú, el uso de un esquema coherencia de color y el uso coherencia de íconos reconocibles ayudan a hacer que la interfaz sea muy buena en el aspecto ergonómico.

Principio 4. ***Centrarse en la transferencia de información.*** El software tiene que ver con la transferencia de información: de una base de datos a un usuario final, de un sistema heredado a una *webapp*, de un usuario final a una interfaz gráfica de usuario (GUI, por sus siglas en inglés), de un sistema operativo a una aplicación, de un componente de software a otro... la lista es casi interminable. En todos los casos, la información fluye a través de una interfaz, y como consecuencia hay posibilidades de cometer errores, omisiones o ambigüedades. Este principio implica que debe ponerse atención especial al análisis, diseño, construcción y prueba de las interfaces.

Principio 5. ***Construir software que tenga modularidad eficaz.*** La separación de entidades (principio 1) establece una filosofía para el software. La *modularidad* proporciona un mecanismo para llevar a cabo dicha filosofía. Cualquier sistema complejo puede dividirse en módulos (componentes), pero la buena práctica de la ingeniería de software demanda más. La modularidad debe ser *eficaz*. Es decir, cada módulo debe centrarse exclusivamente en un aspecto bien delimitado del sistema: debe ser cohesivo en su función o restringido en el contenido que representa. Además, los módulos deben estar interconectados en forma

relativamente sencilla: cada módulo debe tener poco acoplamiento con otros módulos, fuentes de datos y otros aspectos ambientales.

CONSEJO

Use patrones (véase el capítulo 12) a fin de acumular conocimiento y experiencia para las futuras generaciones de ingenieros de software.

Principio 6. ***Buscar patrones.*** Brad Appleton [App00] sugiere que:

> El objetivo de los patrones dentro de la comunidad de software es crear un cúmulo de bibliografía que ayude a los desarrolladores de software a resolver problemas recurrentes que surgen a lo largo del desarrollo. Los patrones ayudan a crear un lenguaje compartido para comunicar perspectiva y experiencia acerca de dichos patrones y sus soluciones. La codificación formal de estas soluciones y sus relaciones permite acumular con éxito el cuerpo de conocimientos que define nuestra comprensión de las buenas arquitecturas que satisfacen las necesidades de sus usuarios.

Principio 7. ***Cuando sea posible, representar el problema y su solución desde varias perspectivas diferentes.*** Cuando un problema y su solución se estudian desde varias perspectivas distintas, es más probable que se tenga mayor visión y que se detecten los errores y omisiones. Por ejemplo, un modelo de requerimientos puede representarse con el empleo de un punto de vista orientado a los datos, a la función o al comportamiento (véanse los capítulos 6 y 7). Cada uno brinda un punto de vista diferente del problema y de sus requerimientos.

Principio 8. ***Tener en mente que alguien dará mantenimiento al software.*** El software será corregido en el largo plazo, cuando se descubran sus defectos, se adapte a los cambios de su ambiente y se mejore en el momento en el que los participantes pidan más capacidades. Estas actividades de mantenimiento resultan más fáciles si se aplica una práctica sólida de ingeniería de software a lo largo del proceso de software.

Estos principios no son todo lo que se necesita para elaborar software de alta calidad, pero establecen el fundamento para todos los métodos de ingeniería de software que se estudian en este libro.

4.3 Principios que guían toda actividad estructural

Cita:

"El ingeniero ideal es una mezcla… no es un científico, no es un matemático, no es un sociólogo ni un escritor; pero para resolver problemas de ingeniería utiliza conocimiento y técnicas de algunas o de todas esas disciplinas."

N. W. Dougherty

En las secciones que siguen se consideran los principios que tienen mucha relevancia para el éxito de cada actividad estructural genérica, definida como parte del proceso de software. En muchos casos, los principios que se estudian para cada una de las actividades estructurales son un refinamiento de los principios presentados en la sección 4.2. Tan sólo son principios fundamentales planteados en un nivel más bajo de abstracción.

4.3.1 Principios de comunicación

Antes de que los requerimientos del cliente se analicen, modelen o especifiquen, deben recabarse a través de la actividad de comunicación. Un cliente tiene un problema que parece abordable mediante una solución basada en computadora. Usted responde a la solicitud de ayuda del cliente. Ha comenzado la comunicación. Pero es frecuente que el camino que lleva de la comunicación a la comprensión esté lleno de agujeros.

La comunicación efectiva (entre colegas técnicos, con el cliente y otros participantes, y con los gerentes de proyecto) se encuentra entre las actividades más difíciles que deben enfrentarse. En este contexto, aquí se estudian principios de comunicación aplicados a la comunicación con el cliente. Sin embargo, muchos de ellos se aplican por igual en todas las formas de comunicación que ocurren dentro de un proyecto de software.

Principio 1. ***Escuchar.*** Trate de centrarse en las palabras del hablante en lugar de formular su respuesta a dichas palabras. Si algo no está claro, pregunte para aclararlo, pero evite las interrupciones constantes. Si una persona habla, *nunca* parezca usted beligerante en sus palabras o actos (por ejemplo, con giros de los ojos o movimientos de la cabeza).

Antes de comunicarse, asegúrese de que entiende el punto de vista de la otra parte, conozca un poco sus necesidades y después escuche.

Principio 2. ***Antes de comunicarse, prepararse.*** Dedique algún tiempo a entender el problema antes de reunirse con otras personas. Si es necesario, haga algunas investigaciones para entender el vocabulario propio del negocio. Si tiene la responsabilidad de conducir la reunión, prepare una agenda antes de que ésta tenga lugar.

Principio 3. ***Alguien debe facilitar la actividad.*** Toda reunión de comunicación debe tener un líder (facilitador) que: 1) mantenga la conversación en movimiento hacia una dirección positiva, 2) sea un mediador en cualquier conflicto que ocurra y 3) garantice que se sigan otros principios.

Cita:

"Las preguntas directas y las respuestas directas son el camino más corto hacia las mayores perplejidades."

Mark Twain

Principio 4. ***Es mejor la comunicación cara a cara.*** Pero por lo general funciona mejor cuando está presente alguna otra representación de la información relevante. Por ejemplo, un participante quizá genere un dibujo o documento en "borrador" que sirva como centro de la discusión.

Principio 5. ***Tomar notas y documentar las decisiones.*** Las cosas encuentran el modo de caer en las grietas. Alguien que participe en la comunicación debe servir como "secretario" y escribir todos los temas y decisiones importantes.

Principio 6. ***Perseguir la colaboración.*** La colaboración y el consenso ocurren cuando el conocimiento colectivo de los miembros del equipo se utiliza para describir funciones o características del producto o sistema. Cada pequeña colaboración sirve para generar confianza entre los miembros del equipo y crea un objetivo común para el grupo.

Principio 7. ***Permanecer centrado; hacer módulos con la discusión.*** Entre más personas participen en cualquier comunicación, más probable es que la conversación salte de un tema a otro. El facilitador debe formar módulos de conversación para abandonar un tema sólo después de que se haya resuelto (sin embargo, considere el principio 9).

Principio 8. ***Si algo no está claro, hacer un dibujo.*** La comunicación verbal tiene sus límites. Con frecuencia, un esquema o dibujo arroja claridad cuando las palabras no bastan para hacer el trabajo.

? ¿Qué pasa si no puede llegarse a un acuerdo con el cliente en algún aspecto relacionado con el proyecto?

Principio 9. ***a) Una vez que se acuerde algo, avanzar. b) Si no es posible ponerse de acuerdo en algo, avanzar. c) Si una característica o función no está clara o no puede aclararse en el momento, avanzar.*** La comunicación, como cualquier actividad de ingeniería de software, requiere tiempo. En vez de hacer iteraciones sin fin, las personas que participan deben reconocer que hay muchos temas que requieren análisis (véase el principio 2) y que "avanzar" es a veces la mejor forma de tener agilidad en la comunicación.

Principio 10. ***La negociación no es un concurso o un juego. Funciona mejor cuando las dos partes ganan.*** Hay muchas circunstancias en las que usted y otros participantes deben negociar funciones y características, prioridades y fechas de entrega. Si el equipo ha

INFORMACIÓN

La diferencia entre los clientes y los usuarios finales

Los ingenieros de software se comunican con muchos participantes diferentes, pero los clientes y los usuarios finales son quienes tienen el efecto más significativo en el trabajo técnico que se desarrollará. En ciertos casos, el cliente y el usuario final son la misma persona, pero para muchos proyectos son individuos distintos que trabajan para diferentes gerentes en distintas organizaciones de negocios.

Un *cliente* es la persona o grupo que 1) solicitó originalmente que se construyera el software, 2) define los objetivos generales del negocio para el software, 3) proporciona los requerimientos básicos del producto y 4) coordina la obtención de fondos para el proyecto. En un negocio de productos o sistema, es frecuente que el cliente sea el departamento de mercadotecnia. En un ambiente de tecnologías de la información (TI), el cliente tal vez sea un componente o departamento del negocio.

Un *usuario final* es la persona o grupo que 1) usará en realidad el software que se elabore para lograr algún propósito del negocio y 2) definirá los detalles de operación del software de modo que se alcance el propósito del negocio.

CasaSegura

Errores de comunicación

La escena: Lugar de trabajo del equipo de ingeniería de software.

Participantes: Jamie Lazar, Vinod Roman y Ed Robins, miembros del equipo de software.

La conversación:

Ed: ¿Qué has oído sobre el proyecto *CasaSegura*?

Vinod: La reunión de arranque está programada para la semana siguiente.

Jamie: Traté de investigar algo, pero no salió bien.

Ed: ¿Qué quieres decir?

Jamie: Bueno, llamé a Lisa Pérez. Ella es la encargada de mercadotecnia en esto.

Vinod: ¿Y…?

Jamie: Yo quería que me dijera las características y funciones de *CasaSegura*… esa clase de cosas. En lugar de ello, comenzó a hacerme preguntas sobre sistemas de seguridad, de vigilancia… No soy experto en eso.

Vinod: ¿Qué te dice eso?

(Jamie se encoge de hombros.)

Vinod: Será que mercadotecnia quiere que actuemos como consultores y mejor que hagamos alguna tarea sobre esta área de productos antes de nuestra junta de arranque. Doug dijo que quería que "colaboráramos" con nuestro cliente, así que será mejor que aprendamos cómo hacerlo.

Ed: Tal vez hubiera sido mejor ir a su oficina. Las llamadas por teléfono simplemente no sirven para esta clase de trabajos.

Jamie: Están en lo correcto. Tenemos que actuar juntos o nuestras primeras comunicaciones serán una batalla.

Vinod: Yo vi a Doug leyendo un libro acerca de "requerimientos de ingeniería". Apuesto a que enlista algunos principios de buena comunicación. Voy a pedírselo prestado.

Jamie: Buena idea… luego nos enseñas.

Vinod (sonríe): Sí, de acuerdo.

colaborado bien, todas las partes tendrán un objetivo común. Aun así, la negociación demandará el compromiso de todas las partes.

4.3.2 Principios de planeación

La actividad de comunicación ayuda a definir las metas y objetivos generales (por supuesto, sujetos al cambio conforme pasa el tiempo). Sin embargo, la comprensión de estas metas y objetivos no es lo mismo que definir un plan para lograrlo. La actividad de planeación incluye un conjunto de prácticas administrativas y técnicas que permiten que el equipo de software defina un mapa mientras avanza hacia su meta estratégica y sus objetivos tácticos.

Cita:

"Al prepararme para una batalla siempre descubro que los planes son inútiles, pero que la planeación es indispensable."

General Dwight D. Eisenhower

Créalo, es imposible predecir con exactitud cómo se desarrollará un proyecto de software. No existe una forma fácil de determinar qué problemas técnicos se encontrarán, qué información importante permanecerá oculta hasta que el proyecto esté muy avanzado, qué malos entendidos habrá o qué aspectos del negocio cambiarán. No obstante, un buen equipo de software debe planear con este enfoque.

Hay muchas filosofías de planeación.[2] Algunas personas son "minimalistas" y afirman que es frecuente que el cambio elimine la necesidad de hacer un plan detallado. Otras son "tradicionalistas" y dicen que el plan da un mapa eficaz y que entre más detalles tenga menos probable será que el equipo se pierda. Otros más son "agilistas" y plantean que tal vez sea necesario un "juego de planeación" rápido, pero que el mapa surgirá a medida que comience el "trabajo real" con el software.

WebRef

En la dirección **www.4pm.com/repository.htm**, hay excelentes materiales informativos sobre la planeación y administración de proyectos.

¿Qué hacer? En muchos proyectos, planear en exceso consume tiempo y es estéril (porque son demasiadas las cosas que cambian), pero planear poco es una receta para el caos. Igual que la mayoría de cosas de la vida, la planeación debe ser tomada con moderación, suficiente para que dé una guía útil al equipo, ni más ni menos. Sin importar el rigor con el que se haga la planeación, siempre se aplican los principios siguientes:

2 En la parte 4 de este libro hay un análisis detallado de la planeación y administración de proyectos de software.

Principio 1. ***Entender el alcance del proyecto.*** Es imposible usar el mapa si no se sabe a dónde se va. El alcance da un destino al equipo de software.

Principio 2. ***Involucrar en la actividad de planeación a los participantes del software.*** Los participantes definen las prioridades y establecen las restricciones del proyecto. Para incluir estas realidades, es frecuente que los ingenieros de software deban negociar la orden de entrega, los plazos y otros asuntos relacionados con el proyecto.

Principio 3. ***Reconocer que la planeación es iterativa.*** Un plan para el proyecto nunca está grabado en piedra. Para cuando el trabajo comience, es muy probable que las cosas hayan cambiado. En consecuencia, el plan deberá ajustarse para incluir dichos cambios. Además, los modelos de proceso iterativo incrementales dictan que debe repetirse la planeación después de la entrega de cada incremento de software, con base en la retroalimentación recibida de los usuarios.

Cita:

"El éxito es más una función del sentido común coherente que del genio."

An Wang

Principio 4. ***Estimar con base en lo que se sabe.*** El objetivo de la estimación es obtener un índice del esfuerzo, costo y duración de las tareas, con base en la comprensión que tenga el equipo sobre el trabajo que va a realizar. Si la información es vaga o poco confiable, entonces las estimaciones tampoco serán confiables.

Principio 5. ***Al definir el plan, tomar en cuenta los riesgos.*** Si ha identificado riesgos que tendrían un efecto grande y es muy probable que ocurran, entonces es necesario elaborar planes de contingencia. Además, el plan del proyecto (incluso la programación de actividades) deberá ajustarse para que incluya la posibilidad de que ocurran uno o más de dichos riesgos.

Principio 6. ***Ser realista.*** Las personas no trabajan 100% todos los días. En cualquier comunicación humana hay ruido. Las omisiones y ambigüedad son fenómenos de la vida. Los cambios ocurren. Aun los mejores ingenieros de software cometen errores. Éstas y otras realidades deben considerarse al establecer un proyecto.

PUNTO CLAVE

El término *granularidad* se refiere al detalle con el que se representan o efectúan algunos elementos de la planeación.

Principio 7. ***Ajustar la granularidad cuando se defina el plan.*** La *granularidad* se refiere al nivel de detalle que se adopta cuando se desarrolla un plan. Un plan con "mucha granularidad" proporciona detalles significativos en las tareas para el trabajo que se planea, en incrementos durante un periodo relativamente corto (por lo que el seguimiento y control suceden con frecuencia). Un plan con "poca granularidad" da tareas más amplias para el trabajo que se planea, para plazos más largos. En general, la granularidad va de poca a mucha conforme el tiempo avanza. En las siguientes semanas o meses, el proyecto se planea con detalles significativos. Las actividades que no ocurrirán en muchos meses no requieren mucha granularidad (hay demasiadas cosas que pueden cambiar).

Principio 8. ***Definir cómo se trata de asegurar la calidad.*** El plan debe identificar la forma en la que el equipo de software busca asegurar la calidad. Si se realizan revisiones técnicas,[3] deben programarse. Si durante la construcción va a utilizarse programación por parejas (véase el capítulo 3), debe definirse en forma explícita en el plan.

Principio 9. ***Describir cómo se busca manejar el cambio.*** Aun la mejor planeación puede ser anulada por el cambio sin control. Debe identificarse la forma en la que van a recibirse los cambios a medida que avanza el trabajo de la ingeniería de software. Por ejemplo, ¿el cliente tiene la posibilidad de solicitar un cambio en cualquier momento? Si se solicita uno, ¿está obligado el equipo a implementarlo de inmediato? ¿Cómo se evalúan el efecto y el costo del cambio?

3 Las revisiones técnicas se estudian en el capítulo 15.

Principio 10. ***Dar seguimiento al plan con frecuencia y hacer los ajustes que se requieran.*** Los proyectos de software se atrasan respecto de su programación. Por tanto, tiene sentido evaluar diariamente el avance, en busca de áreas y situaciones problemáticas en las que las actividades programadas no se apeguen al avance real. Cuando se detecten desviaciones, hay que ajustar el plan en consecuencia.

Para ser más eficaz, cada integrante del equipo de software debe participar en la actividad de planeación. Sólo entonces sus miembros "firmarán" el plan.

4.3.3 Principios de modelado

Se crean modelos para entender mejor la entidad real que se va a construir. Cuando ésta es física (por ejemplo, un edificio, un avión, una máquina, etc.), se construye un modelo de forma idéntica pero a escala. Sin embargo, cuando la entidad que se va a construir es software, el modelo debe adoptar una forma distinta. Debe ser capaz de representar la información que el software transforma, la arquitectura y las funciones que permiten que esto ocurra, las características que desean los usuarios y el comportamiento del sistema mientras la transformación tiene lugar. Los modelos deben cumplir estos objetivos en diferentes niveles de abstracción, en primer lugar con la ilustración del software desde el punto de vista del cliente y después con su representación en un nivel más técnico.

Los modelos de requerimientos representan los requerimientos del cliente. Los modelos del diseño dan una especificación concreta para la construcción del software.

En el trabajo de ingeniería de software se crean dos clases de modelos: de requerimientos y de diseño. Los *modelos de requerimientos* (también conocidos como *modelos de análisis*) representan los requerimientos del cliente mediante la ilustración del software en tres dominios diferentes: el de la información, el funcional y el de comportamiento. Los *modelos de diseño* representan características del software que ayudan a los profesionales a elaborarlo con eficacia: arquitectura, interfaz de usuario y detalle en el nivel de componente.

En su libro sobre modelado ágil, Scott Ambler y Ron Jeffries [Amb02b] definen un conjunto de principios de modelado[4] dirigidos a todos aquellos que usan el modelo de proceso ágil (véase el capítulo 3), pero que son apropiados para todos los ingenieros de software que efectúan acciones y tareas de modelado:

Principio 1. ***El equipo de software tiene como objetivo principal elaborar software, no crear modelos.*** Agilidad significa entregar software al cliente de la manera más rápida posible. Los modelos que contribuyan a esto son benéficos, pero deben evitarse aquellos que hagan lento el proceso o que den poca perspectiva.

Principio 2. ***Viajar ligero, no crear más modelos de los necesarios.*** Todo modelo que se cree debe actualizarse si ocurren cambios. Más importante aún es que todo modelo nuevo exige tiempo, que de otra manera se destinaría a la construcción (codificación y pruebas). Entonces, cree sólo aquellos modelos que hagan más fácil y rápido construir el software.

El objetivo de cualquier modelo es comunicar información. Para lograr esto, use un formato consistente. Suponga que usted no estará para explicar el modelo. Por eso, el modelo debe describirse por sí solo.

Principio 3. ***Tratar de producir el modelo más sencillo que describa al problema o al software.*** No construya software en demasía [Amb02b]. Al mantener sencillos los modelos, el software resultante también lo será. El resultado es que se tendrá un software fácil de integrar, de probar y de mantener (para que cambie). Además, los modelos sencillos son más fáciles de entender y criticar por parte de los miembros del equipo, lo que da como resultado un formato funcional de retroalimentación que optimiza el resultado final.

Principio 4. ***Construir modelos susceptibles al cambio.*** Suponga que sus modelos cambiarán, pero vigile que esta suposición no lo haga descuidado. Por ejemplo, como los

4 Para fines de este libro, se han abreviado y reescrito los principios mencionados en esta sección.

requerimientos se modificarán, hay una tendencia a ignorar los modelos. ¿Por qué? Porque se sabe que de todos modos cambiarán. El problema con esta actitud es que sin un modelo razonablemente completo de los requerimientos, se creará un diseño (modelo de diseño) que de manera invariable carecerá de funciones y características importantes.

Principio 5. ***Ser capaz de enunciar un propósito explícito para cada modelo que se cree.*** Cada vez que cree un modelo, pregúntese por qué lo hace. Si no encuentra una razón sólida para la existencia del modelo, no pierda tiempo en él.

Principio 6. ***Adaptar los modelos que se desarrollan al sistema en cuestión.*** Tal vez sea necesario adaptar a la aplicación la notación del modelo o las reglas; por ejemplo, una aplicación de juego de video quizá requiera una técnica de modelado distinta que el software incrustado que controla el motor de un automóvil en tiempo real.

Principio 7. ***Tratar de construir modelos útiles, pero olvidarse de elaborar modelos perfectos.*** Cuando un ingeniero de software construye modelos de requerimientos y diseño, alcanza un punto de rendimientos decrecientes. Es decir, el esfuerzo requerido para terminar por completo el modelo y hacerlo internamente consistente deja de beneficiarse por tener dichas propiedades. ¿Se sugiere que el modelado debe ser pobre o de baja calidad? La respuesta es "no". Pero el modelado debe hacerse con la mirada puesta en las siguientes etapas de la ingeniería de software. Las iteraciones sin fin para obtener un modelo "perfecto" no cumplen la necesidad de agilidad.

Principio 8. ***No ser dogmático respecto de la sintaxis del modelo. Si se tiene éxito para comunicar contenido, la representación es secundaria.*** Aunque cada miembro del equipo de software debe tratar de usar una notación consistente durante el modelado, la característica más importante del modelo es comunicar información que permita la realización de la siguiente tarea de ingeniería. Si un modelo tiene éxito en hacer esto, es perdonable la sintaxis incorrecta.

Principio 9. ***Si su instinto dice que un modelo no es el correcto a pesar de que se vea bien en el papel, hay razones para estar preocupado.*** Si usted es un ingeniero de software experimentado, confíe en su instinto. El trabajo de software enseña muchas lecciones, algunas en el nivel del inconsciente. Si algo le dice que un modelo de diseño está destinado a fracasar (aun cuando esto no pueda demostrarse en forma explícita), hay razones para dedicar más tiempo a su estudio o a desarrollar otro distinto.

Principio 10. ***Obtener retroalimentación tan pronto como sea posible.*** Todo modelo debe ser revisado por los miembros del equipo. El objetivo de estas revisiones es obtener retroalimentación para utilizarla a fin de corregir los errores de modelado, cambiar las interpretaciones equivocadas y agregar las características o funciones omitidas inadvertidamente.

Requerimientos de los principios de modelado. En las últimas tres décadas se han desarrollado numerosos métodos de modelado de requerimientos. Los investigadores han identificado los problemas del análisis de requerimientos y sus causas, y han desarrollado varias notaciones de modelado y los conjuntos heurísticos correspondientes para resolver aquéllos. Cada método de análisis tiene un punto de vista único. Sin embargo, todos están relacionados por ciertos principios operacionales:

Principio 1. ***Debe representarse y entenderse el dominio de información de un problema.*** El *dominio de información* incluye los datos que fluyen hacia el sistema (usuarios finales, otros sistemas o dispositivos externos), los datos que fluyen fuera del sistema (por la interfaz de usuario, interfaces de red, reportes, gráficas y otros medios) y los almacenamientos de datos que recaban y organizan objetos persistentes de datos (por ejemplo, aquellos que se conservan en forma permanente).

PUNTO CLAVE

El modelado del análisis se centra en tres atributos del software: la información que se va a procesar, la función que se va a entregar y el comportamiento que va a suceder.

Principio 2. ***Deben definirse las funciones que realizará el software.*** Las funciones del software dan un beneficio directo a los usuarios finales y también brindan apoyo interno para las características que son visibles para aquéllos. Algunas funciones transforman los datos que fluyen hacia el sistema. En otros casos, las funciones activan algún nivel de control sobre el procesamiento interno del software o sobre los elementos externos del sistema. Las funciones se describen en muchos y distintos niveles de abstracción, que van de un enunciado de propósito general a la descripción detallada de los elementos del procesamiento que deben invocarse.

Cita:

"En cualquier trabajo de diseño, el primer problema del ingeniero es descubrir cuál es realmente el problema."

Autor desconocido

Principio 3. ***Debe representarse el comportamiento del software (como consecuencia de eventos externos).*** El comportamiento del software de computadora está determinado por su interacción con el ambiente externo. Las entradas que dan los usuarios finales, el control de los datos efectuado por un sistema externo o la vigilancia de datos reunidos en una red son el motivo por el que el software se comporta en una forma específica.

Principio 4. ***Los modelos que representen información, función y comportamiento deben dividirse de manera que revelen los detalles en forma estratificada (o jerárquica).*** El modelado de los requerimientos es el primer paso para resolver un problema de ingeniería de software. Eso permite entender mejor el problema y proporciona una base para la solución (diseño). Los problemas complejos son difíciles de resolver por completo. Por esta razón, debe usarse la estrategia de *divide y vencerás*. Un problema grande y complejo se divide en subproblemas hasta que cada uno de éstos sea relativamente fácil de entender. Este concepto se llama *partición* o *separación de entidades*, y es una estrategia clave en el modelado de requerimientos.

Principio 5. ***El trabajo de análisis debe avanzar de la información esencial hacia la implementación en detalle.*** El modelado de requerimientos comienza con la descripción del problema desde la perspectiva del usuario final. Se describe la "esencia" del problema sin considerar la forma en la que se implementará la solución. Por ejemplo, un juego de video requiere que la jugadora "enseñe" a su protagonista en qué dirección avanzar cuando se mueve hacia un laberinto peligroso. Ésa es la esencia del problema. La implementación detallada (normalmente descrita como parte del modelo del diseño) indica cómo se desarrollará la esencia. Para el juego de video, quizá se use una entrada de voz, o se escriba un comando en un teclado, o tal vez un *joystick* (o *mouse*) apunte en una dirección específica, o quizá se mueva en el aire un dispositivo sensible al movimiento.

Con la aplicación de estos principios, un ingeniero de software aborda el problema en forma sistemática. Pero, ¿cómo se aplican estos principios en la práctica? Esta pregunta se responderá en los capítulos 5 a 7.

Principios del modelado del diseño. El modelo del diseño del software es análogo a los planos arquitectónicos de una casa. Se comienza por representar la totalidad de lo que se va a construir (por ejemplo, un croquis tridimensional de la casa) que se refina poco a poco para que guíe la construcción de cada detalle (por ejemplo, la distribución de la plomería). De manera similar, el modelo del diseño que se crea para el software da varios puntos de vista distintos del sistema.

Cita:

"Vea primero que el diseño es sabio y justo: eso comprobado, siga resueltamente; no para uno renunciar a rechazar el propósito de que ha resuelto llevar a cabo."

William Shakespeare

No escasean los métodos para obtener los distintos elementos de un diseño de software. Algunos son activados por datos, lo que hace que sea la estructura de éstos la que determine la arquitectura del programa y los componentes de procesamiento resultantes. Otros están motivados por el patrón, y usan información sobre el dominio del problema (el modelo de requerimientos) para desarrollar estilos de arquitectura y patrones de procesamiento. Otros más están orientados a objetos, y utilizan objetos del dominio del problema como impulsores de la creación de estructuras de datos y métodos que los manipulan. No obstante la variedad, todos ellos se apegan a principios de diseño que se aplican sin importar el método empleado.

Principio 1. ***El diseño debe poderse rastrear hasta el modelo de requerimientos.*** El modelo de requerimientos describe el dominio de información del problema, las funciones visibles para el usuario, el comportamiento del sistema y un conjunto de clases de requerimientos que agrupa los objetos del negocio con los métodos que les dan servicio. El modelo de diseño traduce esta información en una arquitectura, un conjunto de subsistemas que implementan las funciones principales y un conjunto de componentes que son la realización de las clases de requerimientos. Los elementos del modelo de diseño deben poder rastrearse en el modelo de requerimientos.

WebRef

En la dirección **cs.wwc.edu/~aabyan/Design/**, se encuentran comentarios profundos sobre el proceso de diseño, así como un análisis de la estética del diseño.

Principio 2. ***Siempre tomar en cuenta la arquitectura del sistema que se va a construir.*** La arquitectura del software (véase el capítulo 9) es el esqueleto del sistema que se va a construir. Afecta interfaces, estructuras de datos, flujo de control y comportamiento del programa, así como la manera en la que se realizarán las pruebas, la susceptibilidad del sistema resultante a recibir mantenimiento y mucho más. Por todas estas razones, el diseño debe comenzar con consideraciones de la arquitectura. Sólo después de establecida ésta deben considerarse los aspectos en el nivel de los componentes.

Principio 3. ***El diseño de los datos es tan importante como el de las funciones de procesamiento.*** El diseño de los datos es un elemento esencial del diseño de la arquitectura. La forma en la que los objetos de datos se elaboran dentro del diseño no puede dejarse al azar. Un diseño de datos bien estructurado ayuda a simplificar el flujo del programa, hace más fácil el diseño e implementación de componentes de software y más eficiente el procesamiento conjunto.

Cita:

"Las diferencias no son menores; por el contrario, son como las que había entre Salieri y Mozart. Un estudio tras otro muestran que los mejores diseñadores elaboran estructuras más rápidas, pequeñas, sencillas, claras y producidas con menos esfuerzo."

Frederick P. Brooks

Principio 4. ***Las interfaces (tanto internas como externas) deben diseñarse con cuidado.*** La manera en la que los datos fluyen entre los componentes de un sistema tiene mucho que ver con la eficiencia del procesamiento, la propagación del error y la simplicidad del diseño. Una interfaz bien diseñada hace que la integración sea más fácil y ayuda a quien la somete a prueba a validar las funciones componentes.

Principio 5. ***El diseño de la interfaz de usuario debe ajustarse a las necesidades del usuario final. Sin embargo, en todo caso debe resaltar la facilidad de uso.*** La interfaz de usuario es la manifestación visible del software. No importa cuán sofisticadas sean sus funciones internas, ni lo incluyentes que sean sus estructuras de datos, ni lo bien diseñada que esté su arquitectura, un mal diseño de la interfaz con frecuencia conduce a la percepción de que el software es "malo".

Principio 6. ***El diseño en el nivel de componentes debe tener independencia funcional.*** La independencia funcional es una medida de la "mentalidad única" de un componente de software. La funcionalidad que entrega un componente debe ser cohesiva, es decir, debe centrarse en una y sólo una función o subfunción.[5]

Principio 7. ***Los componentes deben estar acoplados con holgura entre sí y con el ambiente externo.*** El acoplamiento se logra de muchos modos: con una interfaz de componente, con mensajería, por medio de datos globales, etc. A medida que se incrementa el nivel de acoplamiento, también aumenta la probabilidad de propagación del error y disminuye la facilidad general de dar mantenimiento al software. Entonces, el acoplamiento de componentes debe mantenerse tan bajo como sea razonable.

Principio 8. ***Las representaciones del diseño (modelos) deben entenderse con facilidad.*** El propósito del diseño es comunicar información a los profesionales que generarán el código, a los que probarán el software y a otros que le darán mantenimiento en el futuro. Si el diseño es difícil de entender, no servirá como medio de comunicación eficaz.

5 En el capítulo 8 hay más análisis de la cohesión.

Principio 9. ***El diseño debe desarrollarse en forma iterativa. El diseñador debe buscar más sencillez en cada iteración.*** Igual que ocurre con casi todas las actividades creativas, el diseño ocurre de manera iterativa. Las primeras iteraciones sirven para mejorar el diseño y corregir errores, pero las posteriores deben buscar un diseño tan sencillo como sea posible.

Cuando se aplican en forma apropiada estos principios de diseño, se crea uno que exhibe factores de calidad tanto externos como internos [Mye78]. Los *factores de calidad externos* son aquellas propiedades del software fácilmente observables por los usuarios (por ejemplo, velocidad, confiabilidad, corrección y usabilidad). Los *factores de calidad internos* son de importancia para los ingenieros de software. Conducen a un diseño de alta calidad desde el punto de vista técnico. Para obtener factores de calidad internos, el diseñador debe entender los conceptos básicos del diseño (véase el capítulo 8).

4.3.4 Principios de construcción

Cita:

"Durante gran parte de mi vida he sido un mirón del software, y observo furtivamente el código sucio de otras personas. A veces encuentro una verdadera joya, un programa bien estructurado escrito en un estilo consistente, libre de errores, desarrollado de modo que cada componente es sencillo y organizado, y que está diseñado de modo que el producto es fácil de cambiar."

David Parnas

La actividad de construcción incluye un conjunto de tareas de codificación y pruebas que lleva a un software operativo listo para entregarse al cliente o usuario final. En el trabajo de ingeniería de software moderna, la codificación puede ser 1) la creación directa de lenguaje de programación en código fuente (por ejemplo, Java), 2) la generación automática de código fuente que usa una representación intermedia parecida al diseño del componente que se va a construir o 3) la generación automática de código ejecutable que utiliza un "lenguaje de programación de cuarta generación" (por ejemplo, Visual C++).

Las pruebas dirigen su atención inicial al componente, y con frecuencia se denomina *prueba unitaria*. Otros niveles de pruebas incluyen 1) *de integración* (realizadas mientras el sistema está en construcción), 2) *de validación*, que evalúan si los requerimientos se han satisfecho para todo el sistema (o incremento de software) y 3) *de aceptación*, que efectúa el cliente en un esfuerzo por utilizar todas las características y funciones requeridas. Los siguientes principios y conceptos son aplicables a la codificación y prueba:

Principios de codificación. Los principios que guían el trabajo de codificación se relacionan de cerca con el estilo, lenguajes y métodos de programación. Sin embargo, puede enunciarse cierto número de principios fundamentales:

Principios de preparación: ***Antes de escribir una sola línea de código, asegúrese de:***

- Entender el problema que se trata de resolver.
- Comprender los principios y conceptos básicos del diseño.
- Elegir un lenguaje de programación que satisfaga las necesidades del software que se va a elaborar y el ambiente en el que operará.
- Seleccionar un ambiente de programación que disponga de herramientas que hagan más fácil su trabajo.
- Crear un conjunto de pruebas unitarias que se aplicarán una vez que se haya terminado el componente a codificar.

Evite desarrollar un programa elegante que resuelva el problema equivocado. Ponga especial atención al primer principio de preparación.

Principios de programación: ***Cuando comience a escribir código, asegúrese de:***

- Restringir sus algoritmos por medio del uso de programación estructurada [Boh00].
- Tomar en consideración el uso de programación por parejas.
- Seleccionar estructuras de datos que satisfagan las necesidades del diseño.
- Entender la arquitectura del software y crear interfaces que son congruentes con ella.
- Mantener la lógica condicional tan sencilla como sea posible.

- Crear lazos anidados en forma tal que se puedan probar con facilidad.
- Seleccionar nombres significativos para las variables y seguir otros estándares locales de codificación.
- Escribir código que se documente a sí mismo.
- Crear una imagen visual (por ejemplo, líneas con sangría y en blanco) que ayude a entender.

Principios de validación: ***Una vez que haya terminado su primer intento de codificación, asegúrese de:***

- Realizar el recorrido del código cuando sea apropiado.
- Llevar a cabo pruebas unitarias y corregir los errores que se detecten.
- Rediseñar el código.

WebRef

En la dirección **www.literateprogramming.com/fpstyle.html**, hay una amplia variedad de vínculos a estándares de codificación.

Se han escrito más libros sobre programación (codificación) y sobre los principios y conceptos que la guían que sobre cualquier otro tema del proceso de software. Los libros sobre el tema incluyen obras tempranas sobre estilo de programación [Ker78], construcción de software práctico [McC04], perlas de programación [Ben99], el arte de programar [Knu98], temas pragmáticos de programación [Hun99] y muchísimos temas más. El análisis exhaustivo de estos principios y conceptos está más allá del alcance de este libro. Si tiene interés en profundizar, estudie una o varias de las referencias que se mencionan.

Principios de la prueba. En un libro clásico sobre las pruebas de software, Glen Myers [Mye79] enuncia algunas reglas que sirven bien como objetivos de prueba:

? ¿Cuáles son los objetivos de probar el software?

- La prueba es el proceso que ejecuta un programa con objeto de encontrar un error.
- Un buen caso de prueba es el que tiene alta probabilidad de encontrar un error que no se ha detectado hasta el momento.
- Una prueba exitosa es la que descubre un error no detectado hasta el momento.

CONSEJO

En un contexto amplio del diseño de software, recuerde que se comienza "por lo grande" y se centra en la arquitectura del software, y que se termina "en lo pequeño" y se atiende a los componentes. Para la prueba sólo se invierte el proceso.

Estos objetivos implican un cambio muy grande en el punto de vista de ciertos desarrolladores de software. Ellos avanzan contra la opinión común de que una prueba exitosa es aquella que no encuentra errores en el software. El objetivo es diseñar pruebas que detecten de manera sistemática diferentes clases de errores, y hacerlo con el mínimo tiempo y esfuerzo.

Si las pruebas se efectúan con éxito (de acuerdo con los objetivos ya mencionados), descubrirán errores en el software. Como beneficio secundario, la prueba demuestra que las funciones de software parecen funcionar de acuerdo con las especificaciones, y que los requerimientos de comportamiento y desempeño aparentemente se cumplen. Además, los datos obtenidos conforme se realiza la prueba dan una buena indicación de la confiabilidad del software y ciertas indicaciones de la calidad de éste como un todo. Pero las pruebas no pueden demostrar la inexistencia de errores y defectos; sólo demuestran que hay errores y defectos. Es importante recordar esto (que de otro modo parecería muy pesimista) cuando se efectúe una prueba.

Davis [Dav95b] sugiere algunos principios para las pruebas,[6] que se han adaptado para usarlos en este libro:

Principio 1. ***Todas las pruebas deben poder rastrearse hasta los requerimientos del cliente.***[7] El objetivo de las pruebas de software es descubrir errores. Entonces, los defec-

6 Aquí sólo se mencionan pocos de los principios de prueba de Davis. Para más información, consulte [Dav95b].

7 Este principio se refiere a las *pruebas funcionales*, por ejemplo, aquellas que se centran en los requerimientos. Las *pruebas estructurales* (las que se centran en los detalles de arquitectura o lógica) tal vez no aborden directamente los requerimientos específicos.

tos más severos (desde el punto de vista del cliente) son aquellos que hacen que el programa no cumpla sus requerimientos.

Principio 2. ***Las pruebas deben planearse mucho antes de que den comienzo.*** La planeación de las pruebas (véase el capítulo 17) comienza tan pronto como se termina el modelo de requerimientos. La definición detallada de casos de prueba principia apenas se ha concluido el modelo de diseño. Por tanto, todas las pruebas pueden planearse y diseñarse antes de generar cualquier código.

Principio 3. ***El principio de Pareto se aplica a las pruebas de software.*** En este contexto, el principio de Pareto implica que 80% de todos los errores no detectados durante las pruebas se relacionan con 20% de todos los componentes de programas. Por supuesto, el problema es aislar los componentes sospechosos y probarlos a fondo.

Principio 4. ***Las pruebas deben comenzar "en lo pequeño" y avanzar hacia "lo grande".*** Las primeras pruebas planeadas y ejecutadas por lo general se centran en componentes individuales. Conforme avanzan las pruebas, la atención cambia en un intento por encontrar errores en grupos integrados de componentes y, en última instancia, en todo el sistema.

Principio 5. ***No son posibles las pruebas exhaustivas.*** Hasta para un programa de tamaño moderado, el número de permutaciones de las rutas es demasiado grande. Por esta razón, durante una prueba es imposible ejecutar todas las combinaciones de rutas. Sin embargo, es posible cubrir en forma adecuada la lógica del programa y asegurar que se han probado todas las condiciones en el nivel de componentes.

4.3.5 Principios de despliegue

Como se dijo en la parte 1 del libro, la actividad del despliegue incluye tres acciones: entrega, apoyo y retroalimentación. Como la naturaleza de los modelos del proceso del software moderno es evolutiva o incremental, el despliegue ocurre no una vez sino varias, a medida que el software avanza hacia su conclusión. Cada ciclo de entrega pone a disposición de los clientes y usuarios finales un incremento de software operativo que brinda funciones y características utilizables. Cada ciclo de apoyo provee documentación y ayuda humana para todas las funciones y características introducidas durante los ciclos de despliegue realizados hasta ese momento. Cada ciclo de retroalimentación da al equipo de software una guía importante que da como resultado modificaciones de las funciones, de las características y del enfoque adoptado para el siguiente incremento.

Asegúrese de que su cliente sabe lo que puede esperar antes de que se entregue un incremento de software. De otra manera, puede apostar a que el cliente espera más de lo que usted le dará.

La entrega de un incremento de software representa un punto de referencia importante para cualquier proyecto de software. Cuando el equipo se prepara para entregar un incremento, deben seguirse ciertos principios clave:

Principio 1. ***Deben manejarse las expectativas de los clientes.*** Con demasiada frecuencia, el cliente espera más de lo que el equipo ha prometido entregar, y la desilusión llega de inmediato. Esto da como resultado que la retroalimentación no sea productiva y arruine la moral del equipo. En su libro sobre la administración de las expectativas, Naomi Karten [Kar94] afirma que "el punto de inicio de la administración de las expectativas es ser más consciente de lo que se comunica y de la forma en la que esto se hace". Ella sugiere que el ingeniero de software debe tener cuidado con el envío de mensajes conflictivos al cliente (por ejemplo, prometer más de lo que puede entregarse de manera razonable en el plazo previsto, o entregar más de lo que se prometió en un incremento de software y para el siguiente entregar menos).

Principio 2. ***Debe ensamblarse y probarse el paquete completo que se entregará.*** Debe ensamblarse en un CD-ROM u otro medio (incluso descargas desde web) todo el software ejecutable, archivos de datos de apoyo, documentos de ayuda y otra información rele-

vante, para después hacer una prueba beta exhaustiva con usuarios reales. Todos los *scripts* de instalación y otras características de operación deben ejecutarse por completo en tantas configuraciones diferentes de cómputo como sea posible (por ejemplo, hardware, sistemas operativos, equipos periféricos, configuraciones de red, etcétera).

Principio 3. ***Antes de entregar el software, debe establecerse un régimen de apoyo.*** Un usuario final espera respuesta e información exacta cuando surja una pregunta o problema. Si el apoyo es *ad hoc*, o, peor aún, no existe, el cliente quedará insatisfecho de inmediato. El apoyo debe planearse, los materiales respectivos deben prepararse y los mecanismos apropiados de registro deben establecerse a fin de que el equipo de software realice una evaluación categórica de las clases de apoyo solicitado.

Principio 4. Se ***deben proporcionar a los usuarios finales materiales de aprendizaje apropiados.*** El equipo de software entrega algo más que el software en sí. Deben desarrollarse materiales de capacitación apropiados (si se requirieran); es necesario proveer lineamientos para solución de problemas y, cuando sea necesario, debe publicarse "lo que es diferente en este incremento de software".[8]

Principio 5. ***El software defectuoso debe corregirse primero y después entregarse.*** Cuando el tiempo apremia, algunas organizaciones de software entregan incrementos de baja calidad con la advertencia de que los errores "se corregirán en la siguiente entrega". Esto es un error. Hay un adagio en el negocio del software que dice así: "Los clientes olvidarán pronto que entregaste un producto de alta calidad, pero nunca olvidarán los problemas que les causó un producto de mala calidad. El software se los recuerda cada día."

El software entregado brinda beneficios al usuario final, pero también da retroalimentación útil para el equipo que lo desarrolló. Cuando el incremento se libere, debe invitarse a los usuarios finales a que comenten acerca de características y funciones, facilidad de uso, confiabilidad y cualesquiera otras características.

4.4 Resumen

La práctica de la ingeniería de software incluye principios, conceptos, métodos y herramientas que los ingenieros de software aplican en todo el proceso de desarrollo. Todo proyecto de ingeniería de software es diferente. No obstante, existe un conjunto de principios generales que se aplican al proceso como un todo y a cada actividad estructural, sin importar cuál sea el proyecto o el producto.

Existe un conjunto de principios fundamentales que ayudan en la aplicación de un proceso de software significativo y en la ejecución de métodos de ingeniería de software eficaz. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando avanza por el proceso del software. En el nivel de la práctica, los principios fundamentales establecen un conjunto de valores y reglas que sirven como guía al analizar el diseño de un problema y su solución, al implementar ésta y al someterla a prueba para, finalmente, desplegar el software en la comunidad del usuario.

Los principios de comunicación se centran en la necesidad de reducir el ruido y mejorar el ancho de banda durante la conversación entre el desarrollador y el cliente. Ambas partes deben colaborar a fin de lograr la mejor comunicación.

Los principios de planeación establecen lineamientos para elaborar el mejor mapa del proceso hacia un sistema o producto terminado. El plan puede diseñarse sólo para un incremento

8 Durante la actividad de comunicación, el equipo de software debe determinar los tipos de materiales de ayuda que quiere el usuario.

del software, o para todo el proyecto. Sin que esto importe, debe definir lo que se hará, quién lo hará y cuándo se terminará el trabajo.

El modelado incluye tanto el análisis como el diseño, y describe representaciones cada vez más detalladas del software. El objetivo de los modelos es afirmar el entendimiento del trabajo que se va a hacer y dar una guía técnica a quienes implementarán el software. Los principios de modelado dan fundamento a los métodos y notación que se utilizan para crear representaciones del software.

La construcción incorpora un ciclo de codificación y pruebas en el que se genera código fuente para cierto componente y es sometido a pruebas. Los principios de codificación definen las acciones generales que deben tener lugar antes de que se escriba el código, mientras se escribe y una vez terminado. Aunque hay muchos principios para las pruebas, sólo uno predomina: la prueba es el proceso que lleva a ejecutar un programa con objeto de encontrar un error.

El despliegue ocurre cuando se presenta al cliente un incremento de software, e incluye la entrega, apoyo y retroalimentación. Los principios clave para la entrega consideran la administración de las expectativas del cliente y darle información de apoyo adecuada sobre el software. El apoyo demanda preparación anticipada. La retroalimentación permite al cliente sugerir cambios que tengan valor para el negocio y que brinden al desarrollador información para el ciclo iterativo siguiente de ingeniería de software.

Problemas y puntos por evaluar

4.1. Toda vez que la búsqueda de la calidad reclama recursos y tiempo, ¿es posible ser ágil y centrarse en ella?

4.2. De los ocho principios fundamentales que guían el proceso (lo que se estudió en la sección 4.2.1), ¿cuál cree que sea el más importante?

4.3. Describa con sus propias palabras el concepto de *separación de entidades.*

4.4. Un principio de comunicación importante establece que hay que "prepararse antes de comunicarse". ¿Cómo debe manifestarse esta preparación en los primeros trabajos que se hacen? ¿Qué productos del trabajo son resultado de la preparación temprana?

4.5. Haga algunas investigaciones acerca de cómo "facilitar" la actividad de comunicación (use las referencias que se dan u otras distintas) y prepare algunos lineamientos que se centren en la facilitación.

4.6. ¿En qué difiere la comunicación ágil de la comunicación tradicional de la ingeniería de software? ¿En qué se parecen?

4.7. ¿Por qué es necesario "avanzar"?

4.8. Investigue sobre la "negociación" para la actividad de comunicación y prepare algunos lineamientos que se centren sólo en ella.

4.9. Describa lo que significa *granularidad* en el contexto de la programación de actividades de un proyecto.

4.10. ¿Por qué son importantes los modelos en el trabajo de ingeniería de software? ¿Siempre son necesarios? ¿Hay calificadores para la respuesta que se dio sobre esta necesidad?

4.11. ¿Cuáles son los tres "dominios" considerados durante el modelado de requerimientos?

4.12. Trate de agregar un principio adicional a los que se mencionan en la sección 4.3.4 para la codificación.

4.13. ¿Qué es una prueba exitosa?

4.14. Diga si está de acuerdo o en desacuerdo con el enunciado siguiente: "Como entregamos incrementos múltiples al cliente, no debiéramos preocuparnos por la calidad en los primeros incrementos; en las iteraciones posteriores podemos corregir los problemas. Explique su respuesta.

4.15. ¿Por qué es importante la retroalimentación para el equipo de software?

Lecturas y fuentes de información adicionales

La comunicación con el cliente es una actividad de importancia crítica en la ingeniería de software, pero pocos de sus practicantes dedican tiempo a leer sobre ella. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) presenta varios patrones útiles que analizan problemas en la comunicación. Sutliff (*User-Centred Requirements Engineering*, Springer, 2002) se centra mucho en los retos relacionados con la comunicación. Los libros de Weigers (*Software Requierements*, 2a. ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) y Karten [Kar94] analizan a profundidad los métodos para tener una interacción eficaz con el cliente. Aunque su libro no se centra en el software, Hooks y Farry (*Customer Centered Products*, American Management Association, 2000) presentan lineamientos generales útiles para la comunicación con los clientes. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) pone el énfasis en un "equipo conjunto" de clientes y desarrolladores que recaben los requerimientos en colaboración. Somerville y Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) analizan el concepto de "provocación" y las técnicas y otros requerimientos de los principios de ingeniería.

Los conceptos y principios de la comunicación y planeación son estudiados en muchos libros de administración de proyectos. Entre los más útiles se encuentran los de Bechtold (*Essentials of Software Project Management*, 2a. ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4a. ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006) Hughes (*Software Project Management*, McGraw-Hill, 2005) y Stellman y Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] hizo una compilación excelente de referencias sobre principios de la ingeniería de software. Además, virtualmente todo libro al respecto contiene un análisis útil de los conceptos y principios para análisis, diseño y prueba. Entre los más utilizados (además de éste, claro) se encuentran los siguientes:

Abran, A., y J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Christensen, M., y R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice*, 3a. ed., Prentice-Hall, 2005.

Schach, S., *Object- Oriented and Classical Software Engineering*, McGraw-Hill, 7a. ed., 2006.

Sommerville, I., *Software Engineering*, 8a. ed., Addison-Wesley, 2006

Estos libros también presentan análisis detallados sobre los principios de modelado y construcción.

Los principios de modelado se estudian en muchos libros dedicados al análisis de requerimientos o diseño de software. Los libros de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice,* Wiley, 2004) y Penker y Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) analizan los principios y métodos de modelado.

Todo ingeniero de software que trate de hacer diseño está obligado a leer el texto de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990). Winograd y sus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaron una excelente colección de ensayos sobre aspectos prácticos del diseño de software. Constantine y Lockwood (*Software for Use,* Addison-Wesley, 1999) presenta los conceptos asociados con el "diseño centrado en el usuario". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) presenta una reflexión filosófica útil sobre la naturaleza del diseño y el efecto que tienen las decisiones sobre la calidad y la capacidad del equipo para producir software que agregue mucho valor para su cliente. Stahl y sus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) estudian los principios del desarrollo determinado por el modelo.

Son cientos los libros que abordan uno o más elementos de la actividad de construcción. Kernighan y Plauger [Ker78] escribieron un texto clásico sobre el estilo de programación, McConell [McC93] presenta lineamientos prácticos para la construcción de software, Bentley [Ben99] sugiere una amplia variedad de perlas de la programación, Knuth [Knu99] escribió una serie clásica de tres volúmenes acerca del arte de programar y Hunt [Hun99] sugiere lineamientos pragmáticos para la programación.

Myers y sus colegas (*The Art of Software Testing*, 2a. ed., Wiley, 2004) desarrollaron una revisión importante de su texto clásico y muchos principios importantes para la realización de pruebas. Los libros de Perry (*Effective Methods for Software Testing*, 3a. ed., Wiley 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner y sus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) y Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) presentan por separado conceptos y principios importantes para hacer pruebas, así como muchas guías prácticas.

En internet existe una amplia variedad de fuentes de información sobre la práctica de ingeniería de software. En el sitio web del libro se encuentra una lista actualizada de referencias en la Red Mundial que son relevantes para la ingeniería de software: **www.mhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

5

COMPRENSIÓN DE LOS REQUERIMIENTOS

CONCEPTOS CLAVE

Entender los requerimientos de un problema es una de las tareas más difíciles que enfrenta el ingeniero de software. Cuando se piensa por primera vez, no parece tan difícil desarrollar un entendimiento claro de los requerimientos. Después de todo, ¿acaso no sabe el cliente lo que se necesita? ¿No deberían tener los usuarios finales una buena comprensión de las características y funciones que le darán un beneficio? Sorprendentemente, en muchas instancias la respuesta a estas preguntas es "no". E incluso si los clientes y los usuarios finales explican sus necesidades, éstas cambiarán mientras se desarrolla el proyecto.

En el prólogo a un libro escrito por Ralph Young [You01] sobre las prácticas eficaces respecto de los requerimientos, escribí lo siguiente:

> Es la peor de las pesadillas. Un cliente entra a la oficina, toma asiento, lo mira a uno fijamente a los ojos y dice: "Sé que cree que entiende lo que digo, pero lo que usted no entiende es que lo que digo no es lo que quiero decir." Invariablemente, esto pasa cuando ya está avanzado el proyecto, después de que se han hecho compromisos con los plazos de entrega, que hay reputaciones en juego y mucho dinero invertido.
>
> Todos los que hemos trabajado en el negocio de los sistemas y del software durante algunos años hemos vivido la pesadilla descrita, pero pocos hemos aprendido a escapar. Batallamos cuando tratamos de obtener los requerimientos de nuestros clientes. Tenemos problemas para entender la información que obtenemos. Es frecuente que registremos los requerimientos de manera desorganizada y que dediquemos muy poco tiempo a verificar lo que registramos. Dejamos que el cambio nos controle en lugar de establecer mecanismos para controlarlo a él. En pocas palabras, fallamos en establecer un fundamento sólido para el sistema o software. Cada uno de los problemas es difícil. Cuando se combinan, el panorama es atemorizador aun para los gerentes y profesionales más experimentados. Pero hay solución.

UNA MIRADA RÁPIDA

¿Qué es? Antes de comenzar cualquier trabajo técnico es una buena idea aplicar un conjunto de tareas de ingeniería a los requerimientos. Éstas llevarán a la comprensión de cuál será el efecto que tendrá el software en el negocio, qué es lo que quiere el cliente y cómo interactuarán los usuarios finales con el software.

¿Quién lo hace? Los ingenieros de software (que en el mundo de las tecnologías de información a veces son llamados *ingenieros de sistemas* o *analistas*) y todos los demás participantes del proyecto (gerentes, clientes y usuarios) intervienen en la ingeniería de requerimientos.

¿Por qué es importante? Diseñar y construir un elegante programa de cómputo que resuelva el problema equivocado no satisface las necesidades de nadie. Por eso es importante entender lo que el cliente desea antes de comenzar a diseñar y a construir un sistema basado en computadora.

¿Cuáles son los pasos? La ingeniería de requerimientos comienza con la concepción, tarea que define el alcance y la naturaleza del problema que se va a resolver. Va seguida de la indagación, labor que ayuda a los participantes a definir lo que se requiere. Después sigue la elaboración, donde se refinan y modifican los requerimientos básicos. Cuando los participantes definen el problema, tiene lugar una negociación: ¿cuáles son las prioridades, qué es lo esencial, cuándo se requiere? Por último, se especifica el problema de algún modo y luego se revisa o valida para garantizar que hay coincidencia entre la comprensión que usted tiene del problema y la que tienen los participantes.

¿Cuál es el producto final? El objetivo de los requerimientos de ingeniería es proporcionar a todas las partes un entendimiento escrito del problema. Esto se logra por medio de varios productos del trabajo: escenarios de uso, listas de funciones y de características, modelos de requerimientos o especificaciones.

¿Cómo me aseguro de que lo hice bien? Se revisan con los participantes los productos del trabajo de la ingeniería de requerimientos a fin de asegurar que lo que se aprendió es lo que ellos quieren decir en realidad. Aquí cabe una advertencia: las cosas cambiarán aun después de que todas las partes estén de acuerdo, y seguirán cambiando durante todo el proyecto.

Es razonable afirmar que las técnicas que se estudiarán en este capítulo no son una "solución" verdadera para los retos que se mencionaron, pero sí proveen de un enfoque sólido para enfrentarlos.

5.1 Ingeniería de requerimientos

Cita:

"La parte más difícil al construir un sistema de software es decidir qué construir. Ninguna parte del trabajo invalida tanto al sistema resultante si ésta se hace mal. Nada es más difícil de corregir después."

Fred Brooks

El diseño y construcción de software de computadora es difícil, creativo y sencillamente divertido. En realidad, elaborar software es tan atractivo que muchos desarrolladores de software quieren ir directo a él antes de haber tenido el entendimiento claro de lo que se necesita. Argumentan que las cosas se aclararán a medida que lo elaboren, que los participantes en el proyecto podrán comprender sus necesidades sólo después de estudiar las primeras iteraciones del software, que las cosas cambian tan rápido que cualquier intento de entender los requerimientos en detalle es una pérdida de tiempo, que las utilidades salen de la producción de un programa que funcione y que todo lo demás es secundario. Lo que hace que estos argumentos sean tan seductores es que tienen algunos elementos de verdad.[1] Pero todos son erróneos y pueden llevar un proyecto de software al fracaso.

El espectro amplio de tareas y técnicas que llevan a entender los requerimientos se denomina *ingeniería de requerimientos*. Desde la perspectiva del proceso del software, la ingeniería de requerimientos es una de las acciones importantes de la ingeniería de software que comienza durante la actividad de comunicación y continúa en la de modelado. Debe adaptarse a las necesidades del proceso, del proyecto, del producto y de las personas que hacen el trabajo.

PUNTO CLAVE

La ingeniería de requerimientos establece una base sólida para el diseño y la construcción. Sin ésta, el software resultante tiene alta probabilidad de no satisfacer las necesidades del cliente.

La ingeniería de requerimientos tiende un puente para el diseño y la construcción. Pero, ¿dónde se origina el puente? Podría argumentarse que principia en los pies de los participantes en el proyecto (por ejemplo, gerentes, clientes y usuarios), donde se definen las necesidades del negocio, se describen los escenarios de uso, se delinean las funciones y características y se identifican las restricciones del proyecto. Otros tal vez sugieran que empieza con una definición más amplia del sistema, donde el software no es más que un componente del dominio del sistema mayor. Pero sin importar el punto de arranque, el recorrido por el puente lo lleva a uno muy alto sobre el proyecto, lo que le permite examinar el contexto del trabajo de software que debe realizarse; las necesidades específicas que deben abordar el diseño y la construcción; las prioridades que guían el orden en el que se efectúa el trabajo, y la información, las funciones y los comportamientos que tendrán un profundo efecto en el diseño resultante.

CONSEJO

Espere hacer un poco de diseño al recabar los requerimientos, y un poco de requerimientos durante el trabajo de diseño.

La ingeniería de requerimientos proporciona el mecanismo apropiado para entender lo que desea el cliente, analizar las necesidades, evaluar la factibilidad, negociar una solución razonable, especificar la solución sin ambigüedades, validar la especificación y administrar los requerimientos a medida de que se transforman en un sistema funcional [Tha97]. Incluye siete tareas diferentes: concepción, indagación, elaboración, negociación, especificación, validación y administración. Es importante notar que algunas de estas tareas ocurren en paralelo y que todas se adaptan a las necesidades del proyecto.

Cita:

"Las semillas de los desastres enormes del software por lo general se vislumbran en los tres primeros meses del inicio del proyecto."

Coper Jones

Concepción. ¿Cómo inicia un proyecto de software? ¿Existe un solo evento que se convierte en el catalizador de un nuevo sistema o producto basado en computadora o la necesidad evoluciona en el tiempo? No hay respuestas definitivas a estas preguntas. En ciertos casos, una conversación casual es todo lo que se necesita para desencadenar un trabajo grande de ingeniería de software. Pero en general, la mayor parte de proyectos comienzan cuando se identifica una necesidad del negocio o se descubre un nuevo mercado o servicio potencial. Los partici-

1 Esto es cierto en particular para los proyectos pequeños (menos de un mes) y muy pequeños, que requieren relativamente poco esfuerzo de software sencillo. A medida que el software crece en tamaño y complejidad, estos argumentos comienzan a ser falsos.

pantes de la comunidad del negocio (por ejemplo, los directivos, personal de mercadotecnia, gerentes de producto, etc.) definen un caso de negocios para la idea, tratan de identificar el ritmo y profundidad del mercado, hacen un análisis de gran visión de la factibilidad e identifican una descripción funcional del alcance del proyecto. Toda esta información está sujeta a cambio, pero es suficiente para desencadenar análisis con la organización de ingeniería de software.[2]

En la concepción del proyecto,[3] se establece el entendimiento básico del problema, las personas que quieren una solución, la naturaleza de la solución que se desea, así como la eficacia de la comunicación y colaboración preliminares entre los otros participantes y el equipo de software.

Indagación. En verdad que parece muy simple: preguntar al cliente, a los usuarios y a otras personas cuáles son los objetivos para el sistema o producto, qué es lo que va a lograrse, cómo se ajusta el sistema o producto a las necesidades del negocio y, finalmente, cómo va a usarse el sistema o producto en las operaciones cotidianas. Pero no es simple: es muy difícil.

Christel y Kang [Cri92] identificaron cierto número de problemas que se encuentran cuando ocurre la indagación:

¿Por qué es difícil llegar al entendimiento claro de lo que quiere el cliente?

- **Problemas de alcance.** La frontera de los sistemas está mal definida o los clientes o usuarios finales especifican detalles técnicos innecesarios que confunden, más que clarifican, los objetivos generales del sistema.
- **Problemas de entendimiento.** Los clientes o usuarios no están completamente seguros de lo que se necesita, comprenden mal las capacidades y limitaciones de su ambiente de computación, no entienden todo el dominio del problema, tienen problemas para comunicar las necesidades al ingeniero de sistemas, omiten información que creen que es "obvia", especifican requerimientos que están en conflicto con las necesidades de otros clientes o usuarios, o solicitan requerimientos ambiguos o que no pueden someterse a prueba.
- **Problemas de volatilidad.** Los requerimientos cambian con el tiempo.

Para superar estos problemas, debe enfocarse la obtención de requerimientos en forma organizada.

CONSEJO

La elaboración es algo bueno, pero hay que saber cuándo detenerse. La clave es describir el problema en forma que establezca una base firme para el diseño. Si se trabaja más allá de este punto, se está haciendo diseño.

Elaboración. La información obtenida del cliente durante la concepción e indagación se expande y refina durante la elaboración. Esta tarea se centra en desarrollar un modelo refinado de los requerimientos (véanse los capítulos 6 y 7) que identifique distintos aspectos de la función del software, su comportamiento e información.

La elaboración está motivada por la creación y mejora de escenarios de usuario que describan cómo interactuará el usuario final (y otros actores) con el sistema. Cada escenario de usuario se enuncia con sintaxis apropiada para extraer clases de análisis, que son entidades del dominio del negocio visibles para el usuario final. Se definen los atributos de cada clase de análisis y se identifican los servicios[4] que requiere cada una de ellas. Se identifican las relaciones y colaboración entre clases, y se producen varios diagramas adicionales.

Negociación. No es raro que los clientes y usuarios pidan más de lo que puede lograrse dado lo limitado de los recursos del negocio. También es relativamente común que distintos clientes

2 Si va a desarrollarse un sistema basado en computadora, los análisis comienzan en el contexto de un proceso de ingeniería de sistemas. Para más detalles de la ingeniería de sistemas, visite el sitio web de esta obra.

3 Recuerde que el proceso unificado (véase el capítulo 2) define una "fase de concepción" más amplia que incluye las fases de concepción, indagación y elaboración, que son estudiadas en dicho capítulo.

4 Un *servicio* manipula los datos agrupados por clase. También se utilizan los términos *operación* y *método*. Si no está familiarizado con conceptos de la orientación a objetos, consulte el apéndice 2, en el que se presenta una introducción básica.

En una negociación eficaz no debe haber ganador ni perdedor. Ambos lados ganan porque un "trato" con el que ambas partes pueden vivir es algo sólido.

o usuarios propongan requerimientos conflictivos con el argumento de que su versión es "esencial para nuestras necesidades especiales".

Estos conflictos deben reconciliarse por medio de un proceso de negociación. Se pide a clientes, usuarios y otros participantes que ordenen sus requerimientos según su prioridad y que después analicen los conflictos. Con el empleo de un enfoque iterativo que da prioridad a los requerimientos, se evalúa su costo y riesgo, y se enfrentan los conflictos internos; algunos requerimientos se eliminan, se combinan o se modifican de modo que cada parte logre cierto grado de satisfacción.

Especificación. En el contexto de los sistemas basados en computadora (y software), el término *especificación* tiene diferentes significados para distintas personas. Una especificación puede ser un documento escrito, un conjunto de modelos gráficos, un modelo matemático formal, un conjunto de escenarios de uso, un prototipo o cualquier combinación de éstos.

La formalidad y el formato de una especificación varían con el tamaño y complejidad del software que se va a construir.

Algunos sugieren que para una especificación debe desarrollarse y utilizarse una "plantilla estándar" [Som97], con el argumento de que esto conduce a requerimientos presentados en forma consistente y por ello más comprensible. Sin embargo, en ocasiones es necesario ser flexible cuando se desarrolla una especificación. Para sistemas grandes, el mejor enfoque puede ser un documento escrito que combine descripciones en un lenguaje natural con modelos gráficos. No obstante, para productos o sistemas pequeños que residan en ambientes bien entendidos, quizá todo lo que se requiera sea escenarios de uso.

INFORMACIÓN

Formato de especificación de requerimientos de software

Una *especificación de requerimientos de software* (ERS) es un documento que se crea cuando debe especificarse una descripción detallada de todos los aspectos del software que se va a elaborar, antes de que el proyecto comience. Es importante notar que una ERS formal no siempre está en forma escrita. En realidad, hay muchas circunstancias en las que el esfuerzo dedicado a la ERS estaría mejor aprovechado en otras actividades de la ingeniería de software. Sin embargo, se justifica la ERS cuando el software va a ser desarrollado por una tercera parte, cuando la falta de una especificación crearía problemas severos al negocio, si un sistema es complejo en extremo o si se trata de un negocio de importancia crítica.

Karl Wiegers [Wie03], de la empresa Process Impact Inc., desarrolló un formato útil (disponible en **www.processimpact.com/process_assets/srs_template.doc**) que sirve como guía para aquellos que deben crear una ERS completa. Su contenido normal es el siguiente:

Tabla de contenido
Revisión de la historia

1. Introducción
1.1 Propósito
1.2 Convenciones del documento
1.3 Audiencia objetivo y sugerencias de lectura
1.4 Alcance del proyecto
1.5 Referencias

2. Descripción general
2.1 Perspectiva del producto
2.2 Características del producto
2.3 Clases y características del usuario
2.4 Ambiente de operación
2.5 Restricciones de diseño e implementación
2.6 Documentación para el usuario
2.7 Suposiciones y dependencias

3. Características del sistema
3.1 Característica 1 del sistema
3.2 Característica 2 del sistema (y así sucesivamente)

4. Requerimientos de la interfaz externa
4.1 Interfaces de usuario
4.2 Interfaces del hardware
4.3 Interfaces del software
4.4 Interfaces de las comunicaciones

5. Otros requerimientos no funcionales
5.1 Requerimientos de desempeño
5.2 Requerimientos de seguridad
5.3 Requerimientos de estabilidad
5.4 Atributos de calidad del software

6. Otros requerimientos

Apéndice A: Glosario
Apéndice B: Modelos de análisis
Apéndice C: Lista de conceptos

Puede obtenerse una descripción detallada de cada ERS si se descarga el formato desde la URL mencionada antes.

Validación. La calidad de los productos del trabajo que se generan como consecuencia de la ingeniería de los requerimientos se evalúa durante el paso de validación. La validación de los requerimientos analiza la especificación[5] a fin de garantizar que todos ellos han sido enunciados sin ambigüedades; que se detectaron y corrigieron las inconsistencias, las omisiones y los errores, y que los productos del trabajo se presentan conforme a los estándares establecidos para el proceso, el proyecto y el producto.

Un aspecto clave durante la validación de los requerimientos es la consistencia. Utilice el modelo de análisis para asegurar que los requerimientos se han enunciado de manera consistente.

El mecanismo principal de validación de los requerimientos es la revisión técnica (véase el capítulo 15). El equipo de revisión que los valida incluye ingenieros de software, clientes, usuarios y otros participantes, que analizan la especificación en busca de errores de contenido o de interpretación, de aspectos en los que tal vez se requiera hacer aclaraciones, falta de información, inconsistencias (problema notable cuando se hace la ingeniería de productos o sistemas grandes) y requerimientos en conflicto o irreales (no asequibles).

INFORMACIÓN

Lista de verificación para validar requerimientos

Con frecuencia es útil analizar cada requerimiento en comparación con preguntas de verificación. A continuación se presentan algunas:

- ¿Los requerimientos están enunciados con claridad? ¿Podrían interpretarse mal?
- ¿Está identificada la fuente del requerimiento (por ejemplo, una persona, reglamento o documento)? ¿Se ha estudiado el planteamiento final del requerimiento en comparación con la fuente original?
- ¿El requerimiento está acotado en términos cuantitativos?
- ¿Qué otros requerimientos se relacionan con éste? ¿Están comparados con claridad por medio de una matriz de referencia cruzada u otro mecanismo?
- ¿El requerimiento viola algunas restricciones del dominio?
- ¿Puede someterse a prueba el requerimiento? Si es así, ¿es posible especificar las pruebas (en ocasiones se denominan criterios de validación) para ensayar el requerimiento?
- ¿Puede rastrearse el requerimiento hasta cualquier modelo del sistema que se haya creado?
- ¿Es posible seguir el requerimiento hasta los objetivos del sistema o producto?
- ¿La especificación está estructurada en forma que lleva a entenderlo con facilidad, con referencias y traducción fáciles a productos del trabajo más técnicos?
- ¿Se ha creado un índice para la especificación?
- ¿Están enunciadas con claridad las asociaciones de los requerimientos con las características de rendimiento, comportamiento y operación? ¿Cuáles requerimientos parecen ser implícitos?

Administración de los requerimientos. Los requerimientos para sistemas basados en computadora cambian, y el deseo de modificarlos persiste durante toda la vida del sistema. La administración de los requerimientos es el conjunto de actividades que ayudan al equipo del proyecto a identificar, controlar y dar seguimiento a los requerimientos y a sus cambios en cualquier momento del desarrollo del proyecto.[6] Muchas de estas actividades son idénticas a las técnicas de administración de la configuración del software (TAS) que se estudian en el capítulo 22.

5 Recuerde que la naturaleza de la especificación variará con cada proyecto. En ciertos casos, la "especificación" no es más que un conjunto de escenarios de usuario. En otros, la especificación tal vez sea un documento que contiene escenarios, modelos y descripciones escritas.

6 La administración formal de los requerimientos sólo se practica para proyectos grandes que tienen cientos de requerimientos identificables. Para proyectos pequeños, esta actividad tiene considerablemente menos formalidad.

HERRAMIENTAS DE SOFTWARE

Ingeniería de requerimientos

Objetivo: Las herramientas de la ingeniería de los requerimientos ayudan a reunir éstos, a modelarlos, administrarlos y validarlos.

Mecánica: La mecánica de las herramientas varía. En general, éstas elaboran varios modelos gráficos (por ejemplo, UML) que ilustran los aspectos de información, función y comportamiento de un sistema. Estos modelos constituyen la base de todas las demás actividades del proceso de software.

Herramientas representativas:[7]

En el sitio de Volere Requirements, en **www.volere.co.uk/tools.htm**, se encuentra una lista razonablemente amplia (y actualizada) de herramientas para la ingeniería de requerimientos. En los capítulos 6 y 7 se estudian las herramientas que sirven para modelar aquéllos. Las que se mencionan a continuación se centran en su administración.

EasyRM, desarrollada por Cybernetic Intelligence GmbH (**www.easy-rm.com**), construye un diccionario/glosario especial para proyectos, que contiene descripciones y atributos detallados de los requerimientos.

Rational RequisitePro, elaborada por Rational Software (**www-306.ibm.com/software/awdtools/reqpro/**), permite a los usuarios construir una base de datos de requerimientos, representar relaciones entre ellos y organizarlos, indicar su prioridad y rastrearlos.

En el sitio de Volere ya mencionado, se encuentran muchas herramientas adicionales para administrar requerimientos, así como en la dirección **www.jiludwig.com/Requirements_Management_Tools.html**

5.2 ESTABLECER LAS BASES

En el caso ideal, los participantes e ingenieros de software trabajan juntos en el mismo equipo.[8] En esas condiciones, la ingeniería de requerimientos tan sólo consiste en sostener conversaciones significativas con colegas que sean miembros bien conocidos del equipo. Pero es frecuente que en la realidad esto sea muy diferente.

Los clientes o usuarios finales tal vez se encuentren en ciudades o países diferentes, quizá sólo tengan una idea vaga de lo que se requiere, puede ser que tengan opiniones en conflicto sobre el sistema que se va a elaborar, que posean un conocimiento técnico limitado o que dispongan de poco tiempo para interactuar con el ingeniero que recabará los requerimientos. Ninguna de estas posibilidades es deseable, pero todas son muy comunes y es frecuente verse forzado a trabajar con las restricciones impuestas por esta situación.

En las secciones que siguen se estudian las etapas requeridas para establecer las bases que permiten entender los requerimientos de software a fin de que el proyecto comience en forma tal que se mantenga avanzando hacia una solución exitosa.

PUNTO CLAVE

Un *participante* es cualquier persona que tenga interés directo o que se beneficie del sistema que se va a desarrollar.

5.2.1 Identificación de los participantes

Sommerville y Sawyer [Som97] definen *participante* como "cualquier persona que se beneficie en forma directa o indirecta del sistema en desarrollo". Ya se identificaron los candidatos habituales: gerentes de operaciones del negocio, gerentes de producto, personal de mercadotecnia, clientes internos y externos, usuarios finales, consultores, ingenieros de producto, ingenieros de software e ingenieros de apoyo y mantenimiento, entre otros. Cada participante tiene un punto de vista diferente respecto del sistema, obtiene distintos beneficios cuando éste se desarrolla con éxito y corre distintos riesgos si fracasa el esfuerzo de construcción.

7 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

8 Este enfoque es ampliamente recomendable para proyectos que adoptan la filosofía de desarrollo de software ágil.

Durante la concepción, debe hacerse la lista de personas que harán aportes cuando se recaben los requerimientos (véase la sección 5.3). La lista inicial crecerá cuando se haga contacto con los participantes porque a cada uno se le hará la pregunta: "¿A quién más piensa que debe consultarse?"

Cita:

"Ponga a tres participantes en un cuarto y pregúnteles qué clase de sistema quieren. Es probable que escuche cuatro o más opiniones diferentes."

Anónimo

5.2.2 Reconocer los múltiples puntos de vista

Debido a que existen muchos participantes distintos, los requerimientos del sistema se explorarán desde muchos puntos de vista diferentes. Por ejemplo, el grupo de mercadotecnia se interesa en funciones y características que estimularán el mercado potencial, lo que hará que el nuevo sistema sea fácil de vender. Los gerentes del negocio tienen interés en un conjunto de características para que se elabore dentro del presupuesto y que esté listo para ocupar nichos de mercado definidos. Los usuarios finales tal vez quieran características que les resulten familiares y que sean fáciles de aprender y usar. Los ingenieros de software quizá piensen en funciones invisibles para los participantes sin formación técnica, pero que permitan una infraestructura que dé apoyo a funciones y características más vendibles. Los ingenieros de apoyo tal vez se centren en la facilidad del software para recibir mantenimiento.

Cada uno de estos integrantes (y otros más) aportará información al proceso de ingeniería de los requerimientos. A medida que se recaba información procedente de múltiples puntos de vista, los requerimientos que surjan tal vez sean inconsistentes o estén en conflicto uno con otro. Debe clasificarse toda la información de los participantes (incluso los requerimientos inconsistentes y conflictivos) en forma que permita a quienes toman las decisiones escoger para el sistema un conjunto de requerimientos que tenga coherencia interna.

5.2.3 Trabajar hacia la colaboración

Si en un proyecto de software hay involucrados cinco participantes, tal vez se tengan cinco (o más) diferentes opiniones acerca del conjunto apropiado de requerimientos. En los primeros capítulos se mencionó que, para obtener un sistema exitoso, los clientes (y otros participantes) debían colaborar entre sí (sin pelear por insignificancias) y con los profesionales de la ingeniería de software. Pero, ¿cómo se llega a esta colaboración?

El trabajo del ingeniero de requerimientos es identificar las áreas de interés común (por ejemplo, requerimientos en los que todos los participantes estén de acuerdo) y las de conflicto o incongruencia (por ejemplo, requerimientos que desea un participante, pero que están en conflicto con las necesidades de otro). Es la última categoría la que, por supuesto, representa un reto.

INFORMACIÓN

Uso de "puntos de prioridad"

Una manera de resolver requerimientos conflictivos y, al mismo tiempo, mejorar la comprensión de la importancia relativa de todos, es usar un esquema de "votación" con base en *puntos de prioridad*. Se da a todos los participantes cierto número de puntos de prioridad que pueden "gastarse" en cualquier número de requerimientos. Se presenta una lista de éstos y cada participante indica la importancia relativa de cada uno (desde su punto de vista) con la asignación de uno o más puntos de prioridad. Los puntos gastados ya no pueden utilizarse otra vez. Cuando un participante agota sus puntos de prioridad, ya no tiene la posibilidad de hacer algo con los requerimientos. El total de puntos asignados a cada requerimiento por los participantes da una indicación de la importancia general de cada requerimiento.

La colaboración no significa necesariamente que todos los requerimientos los defina un comité. En muchos casos, los participantes colaboran con la aportación de su punto de vista respecto de los requerimientos, pero un influyente "campeón del proyecto" (por ejemplo, el director

del negocio o un tecnólogo experimentado) toma la decisión final sobre los requerimientos que lo integrarán.

5.2.4 Hacer las primeras preguntas

Las preguntas que se hacen en la concepción del proyecto deben estar "libres del contexto" [Gau89]. El primer conjunto de ellas se centran en el cliente y en otros participantes, en las metas y beneficios generales. Por ejemplo, tal vez se pregunte:

Cita:

"Es mejor conocer algunas preguntas que todas las respuestas."

James Thurber

- ¿Quién está detrás de la solicitud de este trabajo?
- ¿Quién usará la solución?
- ¿Cuál será el beneficio económico de una solución exitosa?
- ¿Hay otro origen para la solución que se necesita?

Estas preguntas ayudan a identificar a todos los participantes con interés en el software que se va a elaborar. Además, las preguntas identifican el beneficio mensurable de una implementación exitosa y las posibles alternativas para el desarrollo de software personalizado.

Las preguntas siguientes permiten entender mejor el problema y hacen que el cliente exprese sus percepciones respecto de la solución:

? ¿Cuáles preguntas ayudarían a tener un entendimiento preliminar del problema?

- ¿Cuál sería una "buena" salida generada por una solución exitosa?
- ¿Qué problemas resolvería esta solución?
- ¿Puede mostrar (o describir) el ambiente de negocios en el que se usaría la solución?
- ¿Hay aspectos especiales del desempeño o restricciones que afecten el modo en el que se enfoque la solución?

Las preguntas finales se centran en la eficacia de la actividad de comunicación en sí. Gause y Weinberg [Gau89] las llaman "metapreguntas" y proponen la siguiente lista (abreviada):

Cita:

"El que hace una pregunta es tonto durante cinco minutos; el que no la hace será tonto para siempre."

Proverbio chino

- ¿Es usted la persona indicada para responder estas preguntas? ¿Sus respuestas son "oficiales"?
- ¿Mis preguntas son relevantes para el problema que se tiene?
- ¿Estoy haciendo demasiadas preguntas?
- ¿Puede otra persona dar información adicional?
- ¿Debería yo preguntarle algo más?

Estas preguntas (y otras) ayudarán a "romper el hielo" y a iniciar la comunicación, que es esencial para una indagación exitosa. Pero una reunión de preguntas y respuestas no es un enfoque que haya tenido un éxito apabullante. En realidad, la sesión de preguntas y respuestas sólo debe usarse para el primer encuentro y luego ser reemplazada por un formato de indagación de requerimientos que combine elementos de solución de problemas, negociación y especificación. En la sección 5.3 se presenta un enfoque de este tipo.

5.3 Indagación de los requerimientos

La indagación de los requerimientos (actividad también llamada *recabación de los requerimientos*) combina elementos de la solución de problemas, elaboración, negociación y especificación. A fin de estimular un enfoque colaborativo y orientado al equipo, los participantes trabajan juntos para identificar el problema, proponer elementos de la solución, negociar distintas visiones y especificar un conjunto preliminar de requerimientos para la solución [Zah90].[9]

9 En ocasiones se denomina a este enfoque *técnica facilitada de especificación de la aplicación* (TFEA).

5.3.1 Recabación de los requerimientos en forma colaborativa

Se han propuesto muchos enfoques distintos para recabar los requerimientos en forma colaborativa. Cada uno utiliza un escenario un poco diferente, pero todos son variantes de los siguientes lineamientos básicos:

¿Cuáles son los lineamientos básicos para conducir una reunión a fin de recabar los requerimientos en forma colaborativa?

- Tanto ingenieros de software como otros participantes dirigen o intervienen en las reuniones.
- Se establecen reglas para la preparación y participación.
- Se sugiere una agenda con suficiente formalidad para cubrir todos los puntos importantes, pero con la suficiente informalidad para que estimule el libre flujo de ideas.
- Un "facilitador" (cliente, desarrollador o participante externo) controla la reunión.
- Se utiliza un "mecanismo de definición" (que pueden ser hojas de trabajo, tablas sueltas, etiquetas adhesivas, pizarrón electrónico, grupos de conversación o foro virtual).

Cita:

"Dedicamos mucho tiempo —la mayor parte de todo el esfuerzo del proyecto— no a implementar o hacer pruebas, sino a tratar de decidir qué construir."

Brian Lawrence

La meta es identificar el problema, proponer elementos de la solución, negociar distintos enfoques y especificar un conjunto preliminar de requerimientos de la solución en una atmósfera que favorezca el logro de la meta. Para entender mejor el flujo de eventos conforme ocurren, se presenta un escenario breve que bosqueja la secuencia de hechos que llevan a la reunión para obtener requerimientos, a lo que sucede durante ésta y a lo que sigue después de ella.

Durante la concepción (véase la sección 5.2), hay preguntas y respuestas básicas que establecen el alcance del problema y la percepción general de lo que constituye una solución. Fuera de estas reuniones iniciales, el desarrollador y los clientes escriben una o dos páginas de "solicitud de producto".

Se selecciona un lugar, fecha y hora para la reunión, se escoge un facilitador y se invita a asistir a integrantes del equipo de software y de otras organizaciones participantes. Antes de la fecha de la reunión, se distribuye la solicitud de producto a todos los asistentes.

WebRef

La *solicitud conjunta de desarrollo* (SCD) es una técnica popular para recabar requerimientos. En la dirección **www.carolla.com/wp-jad.htm** se encuentra una buena descripción de ella.

Por ejemplo,[10] considere un extracto de una solicitud de producto escrita por una persona de mercadotecnia involucrada en el proyecto *CasaSegura*. Esta persona escribe la siguiente narración sobre la función de seguridad en el hogar que va a ser parte de *CasaSegura*:

> Nuestras investigaciones indican que el mercado para los sistemas de administración del hogar crece a razón de 40% anual. La primera función de *CasaSegura* que llevemos al mercado deberá ser la de seguridad del hogar. La mayoría de la gente está familiarizada con "sistemas de alarma", por lo que ésta deberá ser fácil de vender.
>
> La función de seguridad del hogar protegería, o reconocería, varias "situaciones" indeseables, como acceso ilegal, incendio y niveles de monóxido de carbono, entre otros. Emplearía sensores inalámbricos para detectar cada situación. Sería programada por el propietario y telefonearía en forma automática a una agencia de vigilancia cuando detectara una situación como las descritas.

CONSEJO

Si un sistema o producto servirá a muchos usuarios, asegúrese de que los requerimientos se obtengan de una franja representativa de ellos. Si sólo uno define todos los requerimientos, el riesgo de no aceptación es elevado.

En realidad, durante la reunión para recabar los requerimientos, otros contribuirían a esta narración y se dispondría de mucha más información. Pero aun con ésta habría ambigüedad, sería probable que existieran omisiones y ocurrieran errores. Por ahora bastará la "descripción funcional" anterior.

Mientras se revisa la solicitud del producto antes de la reunión, se pide a cada asistente que elabore una lista de objetos que sean parte del ambiente que rodeará al sistema, los objetos

10 Este ejemplo (con extensiones y variantes) se usa para ilustrar métodos importantes de la ingeniería de software en muchos de los capítulos siguientes. Como ejercicio, sería provechoso que el lector realizara su propia reunión para recabar requerimientos y que desarrollara un conjunto de listas para ella.

que producirá éste y los que usará para realizar sus funciones. Además, se solicita a cada asistente que haga otra lista de servicios (procesos o funciones) que manipulen o interactúen con los objetos. Por último, también se desarrollan listas de restricciones (por ejemplo, costo, tamaño, reglas del negocio, etc.) y criterios de desempeño (como velocidad y exactitud). Se informa a los asistentes que no se espera que las listas sean exhaustivas, pero sí que reflejen la percepción que cada persona tiene del sistema.

Cita:

"Los hechos no dejan de existir porque se les ignore."

Aldous Huxley

Entre los objetos descritos por *CasaSegura* tal vez estén incluidos el panel de control, detectores de humo, sensores en ventanas y puertas, detectores de movimiento, alarma, un evento (activación de un sensor), una pantalla, una computadora, números telefónicos, una llamada telefónica, etc. La lista de servicios puede incluir *configurar* el sistema, *preparar* la alarma, *vigilar* los sensores, *marcar* el teléfono, *programar* el panel de control y *leer* la pantalla (observe que los servicios actúan sobre los objetos). En forma similar, cada asistente desarrollará una lista de restricciones (por ejemplo, el sistema debe reconocer cuando los sensores no estén operando, debe ser amistoso con el usuario, debe tener una interfaz directa con una línea telefónica estándar, etc.) y de criterios de desempeño (un evento en un sensor debe reconocerse antes de un segundo, debe implementarse un esquema de prioridad de eventos, etcétera).

CONSEJO

Evite el impulso de desechar alguna idea de un cliente con expresiones como "demasiado costosa" o "imprácticа". La intención aquí es negociar una lista aceptable para todos. Para lograrlo, debe tenerse la mente abierta.

Las listas de objetos pueden adherirse a las paredes del cuarto con el empleo de pliegos de papel grandes o con láminas adhesivas, o escribirse en un tablero. Alternativamente, las listas podrían plasmarse en un boletín electrónico, sitio web interno o en un ambiente de grupo de conversación para revisarlas antes de la reunión. Lo ideal es que cada entrada de las listas pueda manipularse por separado a fin de combinar las listas o modificar las entradas y agregar otras. En esta etapa, están estrictamente prohibidos las críticas y el debate.

Una vez que se presentan las listas individuales acerca de un área temática, el grupo crea una lista, eliminando las entradas redundantes o agregando ideas nuevas que surjan durante el análisis, pero no se elimina ninguna. Después de crear listas combinadas para todas las áreas temáticas, sigue el análisis, coordinado por el facilitador. La lista combinada se acorta, se alarga o se modifica su redacción para que refleje de manera apropiada al producto o sistema que se va a desarrollar. El objetivo es llegar a un consenso sobre la lista de objetos, servicios, restricciones y desempeño del sistema que se va a construir.

En muchos casos, un objeto o servicio descrito en la lista requerirá mayores explicaciones. Para lograr esto, los participantes desarrollan *miniespecificaciones* para las entradas en las listas.[11] Cada miniespecificación es una elaboración de un objeto o servicio. Por ejemplo, la correspondiente al objeto **Panel de control** de *CasaSegura* sería así:

> El panel de control es una unidad montada en un muro, sus dimensiones aproximadas son de 9 por 5 pulgadas. Tiene conectividad inalámbrica con los sensores y con una PC. La interacción con el usuario tiene lugar por medio de un tablero que contiene 12 teclas. Una pantalla de cristal líquido de 3 por 3 pulgadas brinda retroalimentación al usuario. El software hace anuncios interactivos, como eco y funciones similares.

Las miniespecificaciones se presentan a todos los participantes para que sean analizadas. Se hacen adiciones, eliminaciones y otras modificaciones. En ciertos casos, el desarrollo de las miniespecificaciones descubrirá nuevos objetos, servicios o restricciones, o requerimientos de desempeño que se agregarán a las listas originales. Durante todos los análisis, el equipo debe posponer los aspectos que no puedan resolverse en la reunión. Se conserva una *lista de aspectos* para volver después a dichas ideas.

11 En vez de crear una miniespecificación, muchos equipos de software eligen desarrollar escenarios del usuario llamados *casos de uso*. Éstos se estudian en detalle en la sección 5.4 y en el capítulo 6.

CASASEGURA

Conducción de una reunión para recabar los requerimientos

La escena: Sala de juntas. Está en marcha la primera reunión para recabar los requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, gerente de ingeniería de software; tres trabajadores de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador (apunta en un pizarrón): De modo que ésa es la lista actual de objetos y servicios para la función de seguridad del hogar.

Persona de mercadotecnia: Eso la cubre, desde nuestro punto de vista.

Vinod: ¿No dijo alguien que quería que toda la funcionalidad de *CasaSegura* fuera accesible desde internet? Eso incluiría la función de seguridad, ¿o no?

Persona de mercadotecnia: Sí, así es… tendremos que añadir esa funcionalidad y los objetos apropiados.

Facilitador: ¿Agrega eso algunas restricciones?

Jamie: Sí, tanto técnicas como legales.

Representante del producto: ¿Qué significa eso?

Jamie: Nos tendríamos que asegurar de que un extraño no pueda ingresar al sistema, desactivarlo y robar en el lugar o hacer algo peor. Mucha responsabilidad sobre nosotros.

Doug: Muy cierto.

Mercadotecnia: Pero lo necesitamos así… sólo asegúrense de impedir que ingrese un extraño.

Ed: Eso es más fácil de decir que de hacer.

Facilitador (interrumpe): No quiero que debatamos esto ahora. Anotémoslo como un aspecto y continuemos.

(Doug, que es el secretario de la reunión, toma debida nota.)

Facilitador: Tengo la sensación de que hay más por considerar aquí.

(El grupo dedica los siguientes 20 minutos a mejorar y aumentar los detalles de la función de seguridad del hogar.)

5.3.2 Despliegue de la función de calidad

El DFC define los requerimientos de forma que maximicen la satisfacción del cliente.

El *despliegue de la función de calidad* (DFC) es una técnica de administración de la calidad que traduce las necesidades del cliente en requerimientos técnicos para el software. El DFC "se concentra en maximizar la satisfacción del cliente a partir del proceso de ingeniería del software" [Zul92]. Para lograr esto, el DFC pone el énfasis en entender lo que resulta valioso para el cliente y luego despliega dichos valores en todo el proceso de ingeniería. El DFC identifica tres tipos de requerimientos [Zul92]:

Todos desean implementar muchos requerimientos emocionantes, pero hay que tener cuidado. Así es como empiezan a "quedar lisiados los requerimientos". Pero en contrapartida, los requerimientos emocionantes llevan a un avance enorme del producto…

Requerimientos normales. Objetivos y metas que se establecen para un producto o sistema durante las reuniones con el cliente. Si estos requerimientos están presentes, el cliente queda satisfecho. Ejemplos de requerimientos normales son los tipos de gráficos pedidos para aparecer en la pantalla, funciones específicas del sistema y niveles de rendimiento definidos.

Requerimientos esperados. Están implícitos en el producto o sistema y quizá sean tan importantes que el cliente no los mencione de manera explícita. Su ausencia causará mucha insatisfacción. Algunos ejemplos de requerimientos esperados son: fácil interacción humano/máquina, operación general correcta y confiable, y facilidad para instalar el software.

Requerimientos emocionantes. Estas características van más allá de las expectativas del cliente y son muy satisfactorias si están presentes. Por ejemplo, el software para un nuevo teléfono móvil viene con características estándar, pero si incluye capacidades inesperadas (como pantalla sensible al tacto, correo de voz visual, etc.) agrada a todos los usuarios del producto.

WebRef

En la dirección **www.qfdi.org** se encuentra información útil sobre el DFC.

Aunque los conceptos del DFC son aplicables en todo el proceso del software [Par96a], hay técnicas específicas de aquél que pueden aplicarse a la actividad de indagación de los requerimientos. El DFC utiliza entrevistas con los clientes, observación, encuestas y estudio de datos históricos (por ejemplo, reportes de problemas) como materia prima para la actividad de recabación

de los requerimientos. Después, estos datos se llevan a una tabla de requerimientos —llamada *tabla de la voz del cliente*— que se revisa con el cliente y con otros participantes. Luego se emplean varios diagramas, matrices y métodos de evaluación para extraer los requerimientos esperados y tratar de percibir requerimientos emocionantes [Aka04].

5.3.3 Escenarios de uso

A medida que se reúnen los requerimientos, comienza a materializarse la visión general de funciones y características del sistema. Sin embargo, es difícil avanzar hacia actividades más técnicas de la ingeniería de software hasta no entender cómo emplearán los usuarios finales dichas funciones y características. Para lograr esto, los desarrolladores y usuarios crean un conjunto de escenarios que identifican la naturaleza de los usos para el sistema que se va a construir. Los escenarios, que a menudo se llaman *casos de uso* [Jac92], proporcionan la descripción de la manera en la que se utilizará el sistema. Los casos de uso se estudian con más detalle en la sección 5.4.

CasaSegura

Desarrollo de un escenario preliminar de uso

La escena: Una sala de juntas, donde continúa la primera reunión para recabar los requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, gerente de ingeniería de software; tres personas de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Hemos estado hablando sobre la seguridad para el acceso a la funcionalidad de *CasaSegura* si ha de ser posible el ingreso por internet. Me gustaría probar algo. Desarrollemos un escenario de uso para entrar a la función de seguridad.

Jamie: ¿Cómo?

Facilitador: Podríamos hacerlo de dos maneras, pero de momento mantengamos las cosas informales. Díganos (señala a una persona de mercadotecnia), ¿cómo visualiza el acceso al sistema?

Persona de mercadotecnia: Um… bueno, es la clase de cosa que haría si estuviera fuera de casa y tuviera que dejar entrar a alguien a ella —por ejemplo, una trabajadora doméstica o un técnico de reparaciones— que no tuviera el código de seguridad.

Facilitador (sonríe): Ésa es la razón por la que lo hace… dígame, ¿cómo lo haría en realidad?

Persona de mercadotecnia: Bueno… lo primero que necesitaría sería una PC. Entraría a un sitio web que mantendríamos para todos los usuarios de *CasaSegura*. Daría mi identificación de usuario y…

Vinod (interrumpe): La página web tendría que ser segura, encriptada, para garantizar que estuviéramos seguros y…

Facilitador (interrumpe): Ésa es buena información, Vinod, pero es técnica. Centrémonos en cómo emplearía el usuario final esta capacidad, ¿está bien?

Vinod: No hay problema.

Persona de mercadotecnia: Decía que entraría a un sitio web y daría mi identificación de usuario y dos niveles de clave.

Jamie: ¿Qué pasa si olvido mi clave?

Facilitador (interrumpe): Buena observación, Jamie, pero no entraremos a ella por ahora. Lo anotaremos y la llamaremos una *excepción*. Estoy seguro de que habrá otras.

Persona de mercadotecnia: Después de que introdujera las claves, aparecería una pantalla que representaría todas las funciones de *CasaSegura*. Seleccionaría la función de seguridad del hogar. El sistema pediría que verificara quién soy, pidiendo mi dirección o número telefónico o algo así. Entonces aparecería un dibujo del panel de control del sistema de seguridad y la lista de funciones que puede realizar —activar el sistema, desactivar el sistema o desactivar uno o más sensores—. Supongo que también me permitiría reconfigurar las zonas de seguridad y otras cosas como ésa, pero no estoy seguro.

(Mientras la persona de mercadotecnia habla, Doug toma muchas notas; esto forma la base para el primer escenario informal de uso. Alternativamente, hubiera podido pedirse a la persona de mercadotecnia que escribiera el escenario, pero esto se hubiera hecho fuera de la reunión.)

5.3.4 Indagación de los productos del trabajo

Los productos del trabajo generados como consecuencia de la indagación de los requerimientos variarán en función del tamaño del sistema o producto que se va a construir. Para la mayoría de sistemas, los productos del trabajo incluyen los siguientes:

¿Qué información se produce como consecuencia de recabar los requerimientos?

- Un enunciado de la necesidad y su factibilidad.
- Un enunciado acotado del alcance del sistema o producto.
- Una lista de clientes, usuarios y otros participantes que intervienen en la indagación de los requerimientos.
- Una descripción del ambiente técnico del sistema.
- Una lista de requerimientos (de preferencia organizados por función) y las restricciones del dominio que se aplican a cada uno.
- Un conjunto de escenarios de uso que dan perspectiva al uso del sistema o producto en diferentes condiciones de operación.
- Cualesquiera prototipos desarrollados para definir requerimientos.

Cada uno de estos productos del trabajo es revisado por todas las personas que participan en la indagación de los requerimientos.

5.4 DESARROLLO DE CASOS DE USO

En un libro que analiza cómo escribir casos de uso eficaces, Alistair Cockburn [Coc01b] afirma que "un caso de uso capta un contrato [...] [que] describe el comportamiento del sistema en distintas condiciones en las que el sistema responde a una petición de alguno de sus participantes [...]". En esencia, un caso de uso narra una historia estilizada sobre cómo interactúa un usuario final (que tiene cierto número de roles posibles) con el sistema en circunstancias específicas. La historia puede ser un texto narrativo, un lineamiento de tareas o interacciones, una descripción basada en un formato o una representación diagramática. Sin importar su forma, un caso de uso ilustra el software o sistema desde el punto de vista del usuario final.

PUNTO CLAVE

Los casos de uso se definen desde el punto de vista de un actor. Un actor es un papel que desempeñan las personas (usuarios) o los dispositivos cuando interactúan con el software.

El primer paso para escribir un caso de uso es definir un conjunto de "actores" que estarán involucrados en la historia. Los *actores* son las distintas personas (o dispositivos) que usan el sistema o producto en el contexto de la función y comportamiento que va a describirse. Los actores representan los papeles que desempeñan las personas (o dispositivos) cuando opera el sistema. Con una definición más formal, un *actor* es cualquier cosa que se comunique con el sistema o producto y que sea externo a éste. Todo actor tiene uno o más objetivos cuando utiliza el sistema.

Es importante notar que un actor y un usuario final no necesariamente son lo mismo. Un usuario normal puede tener varios papeles diferentes cuando usa el sistema, mientras que un actor representa una clase de entidades externas (gente, con frecuencia pero no siempre) que sólo tiene un papel en el contexto del caso de uso. Por ejemplo, considere al operador de una máquina (un usuario) que interactúa con la computadora de control de una celda de manufactura que contiene varios robots y máquinas de control numérico. Después de una revisión cuidadosa de los requerimientos, el software para la computadora de control requiere cuatro diferentes modos (papeles) para la interacción: modo de programación, modo de prueba, modo de vigilancia y modo de solución de problemas. Por tanto, es posible definir cuatro actores: programador, probador, vigilante y solucionador de problemas. En ciertos casos, el operador de la máquina desempeñará todos los papeles. En otros, distintas personas tendrán el papel de cada actor.

WebRef

Un artículo excelente sobre casos de uso puede descargarse desde la dirección **www.ibm.com/developerworks/webservices/library/codesign7.html**

Debido a que la indagación de los requerimientos es una actividad evolutiva, no todos los actores son identificados en la primera iteración. En ésta es posible identificar a los actores principales [Jac92], y a los secundarios cuando se sabe más del sistema. Los *actores principales* interactúan para lograr la función requerida del sistema y obtienen el beneficio previsto de éste. Trabajan con el software en forma directa y con frecuencia. Los *actores secundarios* dan apoyo al sistema, de modo que los primarios puedan hacer su trabajo.

Una vez identificados los actores, es posible desarrollar casos de uso. Jacobson [Jac92] sugiere varias preguntas[12] que debe responder un caso de uso:

¿Qué se necesita saber a fin de desarrollar un caso de uso eficaz?

- ¿Quién es el actor principal y quién(es) el(los) secundario(s)?
- ¿Cuáles son los objetivos de los actores?
- ¿Qué precondiciones deben existir antes de comenzar la historia?
- ¿Qué tareas o funciones principales son realizadas por el actor?
- ¿Qué excepciones deben considerarse al describir la historia?
- ¿Cuáles variaciones son posibles en la interacción del actor?
- ¿Qué información del sistema adquiere, produce o cambia el actor?
- ¿Tendrá que informar el actor al sistema acerca de cambios en el ambiente externo?
- ¿Qué información desea obtener el actor del sistema?
- ¿Quiere el actor ser informado sobre cambios inesperados?

En relación con los requerimientos básicos de *CasaSegura*, se definen cuatro actores: **propietario de la casa** (usuario), **gerente de arranque** (tal vez la misma persona que el **propietario de la casa**, pero en un papel diferente), **sensores** (dispositivos adjuntos al sistema) y **subsistema de vigilancia y respuesta** (estación central que vigila la función de seguridad de la casa de *CasaSegura*). Para fines de este ejemplo, consideraremos sólo al actor llamado **propietario de la casa**. Éste interactúa con la función de seguridad de la casa en varias formas distintas con el empleo del panel de control de la alarma o con una PC:

- Introduce una clave que permita todas las demás interacciones.
- Pregunta sobre el estado de una zona de seguridad.
- Interroga acerca del estado de un sensor.
- En una emergencia, oprime el botón de pánico.
- Activa o desactiva el sistema de seguridad.

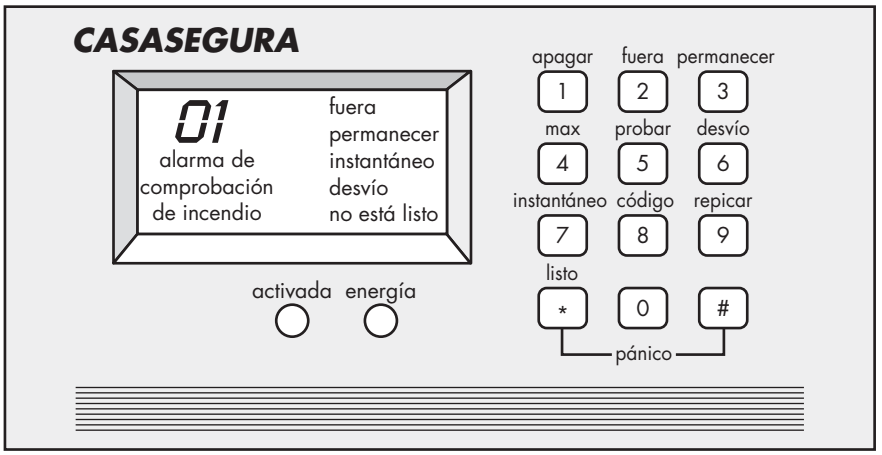
Considerando la situación en la que el propietario de la casa usa el panel de control, a continuación se plantea el caso de uso básico para la activación del sistema:[13]

1. El propietario observa el panel de control de *CasaSegura* (véase la figura 5.1) para determinar si el sistema está listo para recibir una entrada. Si el sistema no está listo, se muestra el mensaje *no está listo* en la pantalla de cristal líquido y el propietario debe cerrar físicamente ventanas o puertas de modo que desaparezca dicho mensaje [el mensaje *no está listo* implica que un sensor está abierto; por ejemplo, que una puerta o ventana está abierta].
2. El propietario usa el teclado para introducir una clave de cuatro dígitos. La clave se compara con la que guarda el sistema como válida. Si la clave es incorrecta, el panel de control emitirá un sonido una vez y se reiniciará para recibir una entrada adicional. Si la clave es correcta, el panel de control espera otras acciones.
3. El propietario selecciona y teclea *permanecer* o *fuera* (véase la figura 5.1) para activar el sistema. La entrada *permanecer* activa sólo sensores perimetrales (se desactivan los sensores de detección de movimiento interior). La entrada *fuera* activa todos los sensores.
4. Cuando ocurre una activación, el propietario observa una luz roja de alarma.

12 Las preguntas de Jacobson se han ampliado para que den una visión más completa del contenido del caso de uso.

13 Observe que este caso de uso difiere de la situación en la que se accede al sistema a través de internet. En este caso, la interacción es por medio del panel de control y no con la interfaz de usuario gráfica (GUI) que se da cuando se emplea una PC.

FIGURA 5.1
Panel de control de *CasaSegura*



Es frecuente que los casos de uso se escriban de manera informal. Sin embargo, utilice el formato que se presenta aquí para asegurar que se incluyen todos los aspectos clave.

El caso de uso básico presenta una historia de alto nivel que describe la interacción entre el actor y el sistema.

En muchas circunstancias, los casos de uso son más elaborados a fin de que brinden muchos más detalles sobre la interacción. Por ejemplo, Cockburn [Coc01b] sugiere el formato siguiente para hacer descripciones detalladas de casos de uso:

Caso de uso: *IniciarVigilancia*

Actor principal: Propietario.

Objetivo en contexto: Preparar el sistema para que vigile los sensores cuando el propietario salga de la casa o permanezca dentro.

Precondiciones: El sistema se ha programado para recibir una clave y reconocer distintos sensores.

Disparador: El propietario decide "preparar" el sistema, por ejemplo, para que encienda las funciones de alarma.

Escenario:

1. Propietario: observa el panel de control
2. Propietario: introduce una clave
3. Propietario: selecciona "permanecer" o "fuera"
4. Propietario: observa una luz roja de alarma que indica que *CasaSegura* ha sido activada.

Excepciones:

1. El panel de control *no está listo*: el propietario verifica todos los sensores para determinar cuáles están abiertos; los cierra.
2. La clave es incorrecta (el panel de control suena una vez): el propietario introduce la clave correcta.
3. La clave no es reconocida: debe contactarse el subsistema de vigilancia y respuesta para reprogramar la clave.
4. Se elige *permanecer*: el panel de control suena dos veces y se enciende un letrero luminoso que dice *permanecer*; se activan los sensores del perímetro.
5. Se selecciona *fuera*: el panel de control suena tres veces y se enciende un letrero luminoso que dice *fuera*; se activan todos los sensores.

Prioridad: Esencial, debe implementarse

Cuándo estará disponible: En el primer incremento

Frecuencia de uso: Muchas veces por día

Canal para el actor: A través de la interfaz del panel de control

Actores secundarios: Técnico de apoyo, sensores

Canales para los actores secundarios:

Técnico de apoyo: línea telefónica

Sensores: interfaces cableadas y frecuencia de radio

Aspectos pendientes:

1. ¿Debe haber una forma de activar el sistema sin usar clave o con una clave abreviada?
2. ¿El panel de control debe mostrar mensajes de texto adicionales?
3. ¿De cuánto tiempo dispone el propietario para introducir la clave a partir del momento en el que se oprime la primera tecla?
4. ¿Hay una forma de desactivar el sistema antes de que se active en realidad?

Los casos de uso para otras interacciones de **propietario** se desarrollarían en una forma similar. Es importante revisar con cuidado cada caso de uso. Si algún elemento de la interacción es ambiguo, es probable que la revisión del caso de uso lo detecte.

CasaSegura

Desarrollo de un diagrama de caso de uso de alto nivel

La escena: Sala de juntas, continúa la reunión para recabar los requerimientos.

Participantes: Jamie Lazar, miembro del equipo de software; Vinod Roman, integrante del equipo de software; Ed Robbins, integrante del equipo de software; Doug Miller, gerente de ingeniería de software; tres miembros de mercadotecnia; un representante de ingeniería del producto; un facilitador.

La conversación:

Facilitador: Hemos pasado un buen tiempo hablando de la función de seguridad del hogar de *CasaSegura*. Durante el receso hice un diagrama de caso de uso para resumir los escenarios importantes que forman parte de esta función. Veámoslo.

(Todos los asistentes observan la figura 5.2.)

Jamie: Estoy aprendiendo la notación UML.[14] Veo que la función de seguridad del hogar está representada por el rectángulo grande con óvalos en su interior, ¿verdad? ¿Y los óvalos representan los casos de uso que hemos escrito?

Facilitador: Sí. Y las figuras pegadas representan a los actores —personas o cosas que interactúan con el sistema según los describe el caso de uso…—; ¡ah! usé el cuadrado para representar un actor que no es persona… en este caso, sensores.

Doug: ¿Es válido eso en UML?

Facilitador: La legalidad no es lo importante. El objetivo es comunicar información. Veo que usar una figura humana para representar un equipo sería erróneo. Así que adapté las cosas un poco. No pienso que genere problemas.

Vinod: Está bien, entonces tenemos narraciones de casos de uso para cada óvalo. ¿Necesitamos desarrollarlas con base en los formatos sobre los que he leído?

Facilitador: Es probable, pero eso puede esperar hasta que hayamos considerado otras funciones de *CasaSegura*.

Persona de mercadotecnia: Esperen, he estado observando este diagrama y de pronto me doy cuenta de que hemos olvidado algo.

Facilitador: ¿De verdad? Dime, ¿qué hemos olvidado?

(La reunión continúa.)

14 En el apéndice 1 se presenta un breve método de aprendizaje de UML para aquellos lectores que no estén familiarizados con dicha notación.

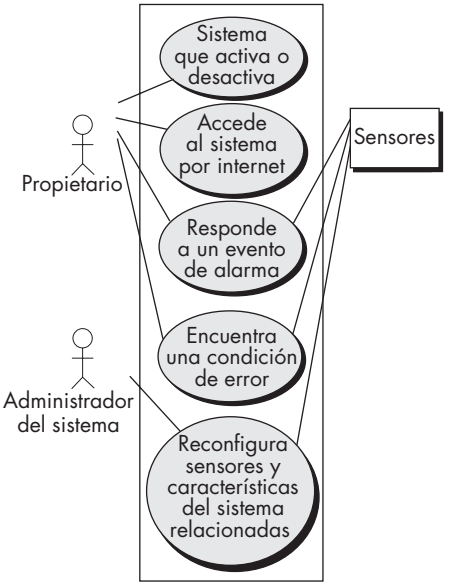


FIGURA 5.2
Diagrama de caso de uso de UML para la función de seguridad del hogar de *CasaSegura*

HERRAMIENTAS DE SOFTWARE

Desarrollo de un caso de uso

Objetivo: Ayudar a desarrollar casos de uso proporcionando formatos y mecanismos automatizados para evaluar la claridad y consistencia.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas para casos de uso dan formatos con espacios en blanco para ser llenados y crear así casos eficaces. La mayor parte de la funcionalidad de los casos de uso está incrustada en un conjunto más amplio de funciones de ingeniería de los requerimientos.

Herramientas representativas[15]

La gran mayoría de herramientas de análisis del modelado basadas en UML dan apoyo tanto de texto como gráfico para el desarrollo y modelado de casos de uso.

Objects by Design
(www.objectsbydesign.com/tools/umltools_byCompany.html) proporciona vínculos exhaustivos con herramientas de este tipo.

5.5 ELABORACIÓN DEL MODELO DE LOS REQUERIMIENTOS[16]

El objetivo del modelo del análisis es describir los dominios de información, función y comportamiento que se requieren para un sistema basado en computadora. El modelo cambia en forma dinámica a medida que se aprende más sobre el sistema por construir, y otros participantes comprenden más lo que en realidad requieren. Por esa razón, el modelo del análisis es una fotografía de los requerimientos en cualquier momento dado. Es de esperar que cambie.

A medida que evoluciona el modelo de requerimientos, ciertos elementos se vuelven relativamente estables, lo que da un fundamento sólido para diseñar las tareas que sigan. Sin embargo, otros elementos del modelo son más volátiles, lo que indica que los participantes todavía no entienden bien los requerimientos para el sistema. En los capítulos 6 y 7 se presentan en

15 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

16 En este libro se usan como sinónimos las expresiones *modelar el análisis* y *modelar los requerimientos*. Ambos se refieren a representaciones de los dominios de la información, funcional y de comportamiento que describen los requerimientos del problema.

detalle el modelo del análisis y los métodos que se usan para construirlo. En las secciones siguientes se da un panorama breve.

5.5.1 Elementos del modelo de requerimientos

Hay muchas formas diferentes de concebir los requerimientos para un sistema basado en computadora. Algunos profesionales del software afirman que es mejor seleccionar un modo de representación (por ejemplo, el caso de uso) y aplicarlo hasta excluir a todos los demás. Otros piensan que es más benéfico usar cierto número de modos de representación distintos para ilustrar el modelo de requerimientos. Los modos diferentes de representación fuerzan a considerar los requerimientos desde distintos puntos de vista, enfoque que tiene una probabilidad mayor de detectar omisiones, inconsistencia y ambigüedades.

Los elementos específicos del modelo de requerimientos están determinados por el método de análisis de modelado (véanse los capítulos 6 y 7) que se use. No obstante, la mayoría de modelos tiene en común un conjunto de elementos generales.

Siempre es buena idea involucrar a los participantes. Una de las mejores formas de lograrlo es hacer que cada uno escriba casos de uso que narren el modo en el que se utilizará el software.

Elementos basados en el escenario. El sistema se describe desde el punto de vista del usuario con el empleo de un enfoque basado en el escenario. Por ejemplo, los casos de uso básico (véase la sección 5.4) y sus diagramas correspondientes de casos de uso (véase la figura 5.2) evolucionan hacia otros más elaborados que se basan en formatos. Los elementos del modelo de requerimientos basados en el escenario con frecuencia son la primera parte del modelo en desarrollo. Como tales, sirven como entrada para la creación de otros elementos de modelado. La figura 5.3 ilustra un diagrama de actividades UML[17] para indagar los requerimientos y representarlos con el empleo de casos de uso. Se aprecian tres niveles de elaboración que culminan en una representación basada en el escenario.

Una forma de aislar las clases es buscar sustantivos descriptivos en un caso de usuario expresado con texto. Al menos algunos de ellos serán candidatos cercanos. Sobre esto se habla más en el capítulo 8.

Elementos basados en clases. Cada escenario de uso implica un conjunto de objetos que se manipulan cuando un actor interactúa con el sistema. Estos objetos se clasifican en clases: conjunto de objetos que tienen atributos similares y comportamientos comunes. Por ejemplo, para ilustrar la clase **Sensor** de la función de seguridad de *Casa Segura* (véase la figura 5.4), puede utilizarse un diagrama de clase UML. Observe que el diagrama enlista los atributos de los sensores (por ejemplo, nombre, tipo, etc.) y las operaciones (por ejemplo, *identificar* y *permitir*) que se aplican para modificarlos. Además de los diagramas de clase, otros elementos de modelado del análisis ilustran la manera en la que las clases colaboran una con otra y las relaciones e interacciones entre ellas. Esto se analiza con más detalle en el capítulo 7.

Elementos de comportamiento. El comportamiento de un sistema basado en computadora tiene un efecto profundo en el diseño que se elija y en el enfoque de implementación que se aplique. Por tanto, el modelo de requerimientos debe proveer elementos de modelado que ilustren el comportamiento.

Un estado es un modo de comportamiento observable desde el exterior. Los estímulos externos ocasionan transiciones entre los estados.

El *diagrama de estado* es un método de representación del comportamiento de un sistema que ilustra sus estados y los eventos que ocasionan que el sistema cambie de estado. Un *estado* es cualquier modo de comportamiento observable desde el exterior. Además, el diagrama de estado indica acciones (como la activación de un proceso, por ejemplo) tomadas como consecuencia de un evento en particular.

Para ilustrar el uso de un diagrama de estado, considere el software incrustado dentro del panel de control de *CasaSegura* que es responsable de leer las entradas que hace el usuario. En la figura 5.5 se presenta un diagrama de estado UML simplificado.

Además de las representaciones de comportamiento del sistema como un todo, también es posible modelar clases individuales. Sobre esto se presentan más análisis en el capítulo 7.

17 En el apéndice 1 se presenta un instructivo breve sobre UML, para aquellos lectores que no estén familiarizados con dicha notación.

FIGURA 5.3

Diagramas de actividad del UML para indagar los requerimientos

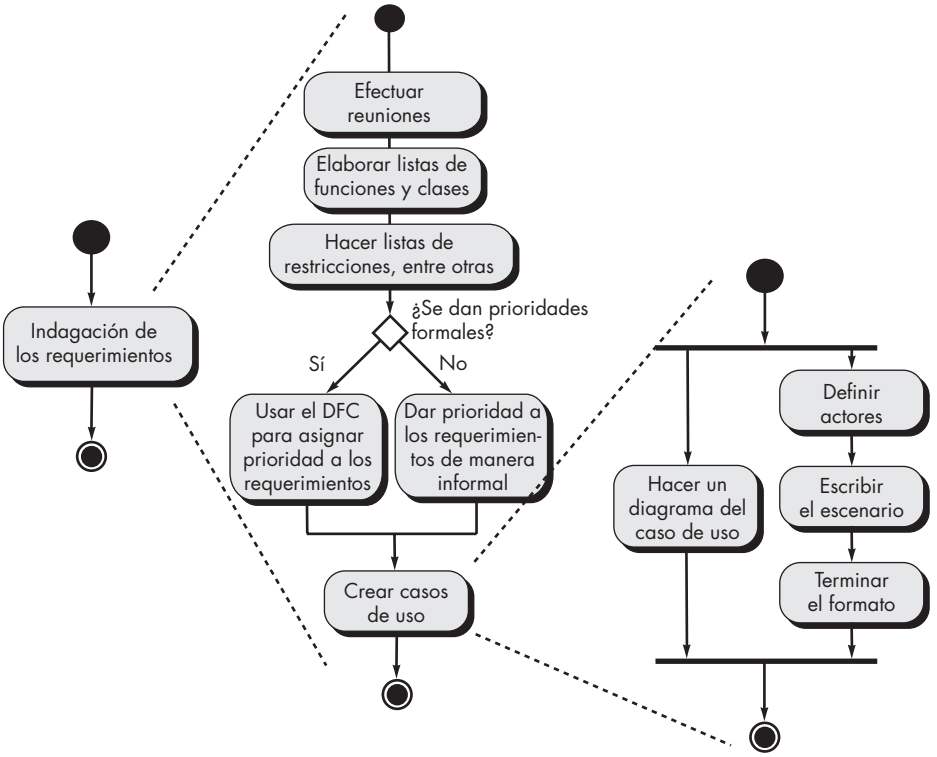


FIGURA 5.4

Diagrama de clase para un sensor

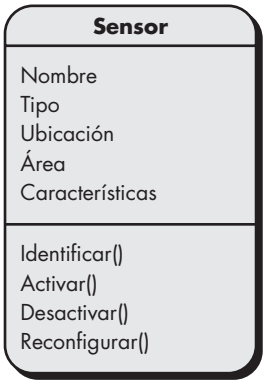
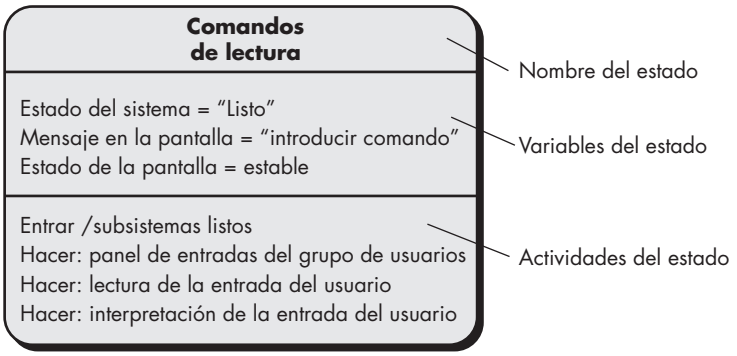


FIGURA 5.5

Notación UML del diagrama de estado



CasaSegura

Modelado preliminar del comportamiento

La escena: Sala de juntas, continúa la reunión de requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, integrante del equipo de software; Doug Miller, gerente de ingeniería de software; tres trabajadores de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Estamos por terminar de hablar sobre la funcionalidad de seguridad del hogar de *CasaSegura*. Pero antes, quisiera que analizáramos el comportamiento de la función.

Persona de mercadotecnia: No entiendo lo que quiere decir con *comportamiento*.

Ed (sonríe): Es cuando le das un "tiempo fuera" al producto si se porta mal.

Facilitador: No exactamente. Permítanme explicarlo.

(El facilitador explica al equipo encargado de recabar los requerimientos y los fundamentos de modelado del comportamiento.)

Persona de mercadotecnia: Esto parece un poco técnico. No estoy seguro de ser de ayuda aquí.

Facilitador: Seguro que lo serás. ¿Qué comportamiento se observa desde el punto de vista de un usuario?

Persona de mercadotecnia: Mmm... bueno, el sistema estará *vigilando* los sensores. *Leerá comandos* del propietario. *Mostrará* su estado.

Facilitador: ¿Ves?, lo puedes hacer.

Jamie: También estará *interrogando* a la PC para determinar si hay alguna entrada desde ella, por ejemplo, un acceso por internet o información sobre la configuración.

Vinod: Sí, en realidad, *configurar* el sistema es un estado por derecho propio.

Doug: Muchachos, lo hacen bien. Pensemos un poco más... ¿hay alguna forma de hacer un diagrama de todo esto?

Facilitador: Sí la hay, pero la dejaremos para la próxima reunión.

Elementos orientados al flujo. La información se transforma cuando fluye a través de un sistema basado en computadora. El sistema acepta entradas en varias formas, aplica funciones para transformarla y produce salidas en distintos modos. La entrada puede ser una señal de control transmitida por un transductor, una serie de números escritos con el teclado por un operador humano, un paquete de información enviado por un enlace de red o un archivo grande de datos recuperado de un almacenamiento secundario. La transformación quizá incluya una sola comparación lógica, un algoritmo numérico complicado o un enfoque de regla de inferencia para un sistema experto. La salida quizá encienda un diodo emisor de luz o genere un informe de 200 páginas. En efecto, es posible crear un modelo del flujo para cualquier sistema basado en computadora, sin importar su tamaño y complejidad. En el capítulo 7 se hace un análisis más detallado del modelado del flujo.

5.5.2 Patrones de análisis

Cualquiera que haya hecho la ingeniería de los requerimientos en varios proyectos de software ha observado que ciertos problemas son recurrentes en todos ellos dentro de un dominio de aplicación específico.[18] Estos *patrones de análisis* [Fow97] sugieren soluciones (por ejemplo, una clase, función o comportamiento) dentro del dominio de la aplicación que pueden volverse a utilizar cuando se modelen muchas aplicaciones.

Geyer-Schulz y Hahsler [Gey01] sugieren dos beneficios asociados con el uso de patrones de análisis:

> En primer lugar, los patrones de análisis aceleran el desarrollo de los modelos de análisis abstracto que capturan los principales requerimientos del problema concreto, debido a que proveen modelos de análisis reutilizables con ejemplos, así como una descripción de sus ventajas y limitaciones. En se-

18 En ciertos casos, los problemas vuelven a suceder sin importar el dominio de la aplicación. Por ejemplo, son comunes las características y funciones usadas para resolver problemas de la interfaz de usuario sin importar el dominio de la aplicación en consideración.

gundo lugar, los patrones de análisis facilitan la transformación del modelo de análisis en un modelo del diseño, sugiriendo patrones de diseño y soluciones confiables para problemas comunes.

Los patrones de análisis se integran en el modelo del análisis, haciendo referencia al nombre del patrón. También se guardan en un medio de almacenamiento de modo que los ingenieros de requerimientos usen herramientas de búsqueda para encontrarlos y aplicarlos. La información sobre el patrón de análisis (y otros tipos de patrones) se presenta en un formato estándar [Gey01][19] que se estudia con más detalle en el capítulo 12. En el capítulo 7 se dan ejemplos de patrones de análisis y más detalles de este tema.

5.6 Requerimientos de las negociaciones

Cita:

"Un compromiso es el arte de dividir un pastel en forma tal que todos crean que tienen el trozo mayor."

Ludwig Erhard

En un contexto ideal de la ingeniería de los requerimientos, las tareas de concepción, indagación y elaboración determinan los requerimientos del cliente con suficiente detalle como para avanzar hacia las siguientes actividades de la ingeniería de software. Desafortunadamente, esto rara vez ocurre. En realidad, se tiene que entrar en negociaciones con uno o varios participantes. En la mayoría de los casos, se pide a éstos que evalúen la funcionalidad, desempeño y otras características del producto o sistema, en contraste con el costo y el tiempo para entrar al mercado. El objetivo de esta negociación es desarrollar un plan del proyecto que satisfaga las necesidades del participante y que al mismo tiempo refleje las restricciones del mundo real (por ejemplo, tiempo, personas, presupuesto, etc.) que se hayan establecido al equipo del software.

Las mejores negociaciones buscan un resultado "ganar-ganar".[20] Es decir, los participantes ganan porque obtienen el sistema o producto que satisface la mayoría de sus necesidades y usted (como miembro del equipo de software) gana porque trabaja con presupuestos y plazos realistas y asequibles.

WebRef

Un artículo breve sobre la negociación para los requerimientos de software puede descargarse desde la dirección **www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html**

Boehm [Boe98] define un conjunto de actividades de negociación al principio de cada iteración del proceso de software. En lugar de una sola actividad de comunicación con el cliente, se definen las actividades siguientes:

1. Identificación de los participantes clave del sistema o subsistema.
2. Determinación de las "condiciones para ganar" de los participantes.

Información

El arte de la negociación

Aprender a negociar con eficacia le servirá en su vida personal y técnica. Es útil considerar los lineamientos que siguen:

1. *Reconocer que no es una competencia.* Para tener éxito, ambas partes tienen que sentir que han ganado o logrado algo. Las dos tienen que llegar a un compromiso.
2. *Mapear una estrategia.* Decidir qué es lo que le gustaría lograr; qué quiere obtener la otra parte y cómo hacer para que ocurran las dos cosas.
3. *Escuchar activamente.* No trabaje en la formulación de su respuesta mientras la otra parte esté hablando. Escúchela. Es probable que obtenga conocimientos que lo ayuden a negociar mejor su posición.
4. *Centrarse en los intereses de la otra parte.* Si quiere evitar conflictos, no adopte posiciones inamovibles.
5. *No lo tome en forma personal.* Céntrese en el problema que necesita resolverse.
6. *Sea creativo.* Si están empantanados, no tenga miedo de pensar fuera de los moldes.
7. *Esté listo para comprometerse.* Una vez que se llegue a un acuerdo, no titubee; comprométase con él y cúmplalo.

19 En la bibliografía existen varias propuestas de formatos para patrones. Si el lector tiene interés, consulte [Fow97], [Gam95], [Yac03] y [Bus07], entre muchas otras fuentes.

20 Se han escrito decenas de libros acerca de las aptitudes para negociar (por ejemplo [Lew06], [Rai06] y [Fis06]). Es una de las aptitudes más importantes que pueda aprender. Lea alguno.

3. Negociación de las condiciones para ganar de los participantes a fin de reconciliarlas en un conjunto de condiciones ganar-ganar para todos los que intervienen (incluso el equipo de software).

La realización exitosa de estos pasos iniciales lleva a un resultado ganar-ganar, que se convierte en el criterio clave para avanzar hacia las siguientes actividades de la ingeniería de software.

CASASEGURA

El principio de una negociación

La escena: Oficina de Lisa Pérez, después de la primera reunión para recabar los requerimientos.

Participantes: Doug Miller, gerente de ingeniería de software, y Lisa Pérez, gerente de mercadotecnia.

La conversación:

Lisa: Pues escuché que la primera reunión salió realmente bien.

Doug: En realidad, sí. Enviaste buenos representantes... contribuyeron de verdad.

Lisa (sonríe): Sí; en realidad me dijeron que habían entrado y que no había sido una "actividad que les despejara la cabeza".

Doug (ríe): La próxima vez me aseguraré de quitarme la vena tecnológica... Mira, Lisa, creo que tenemos un problema para llegar a toda esa funcionalidad del sistema de seguridad para el hogar en las fechas que propone tu dirección. Sé que aún es temprano, pero hice un poco de planeación sobre las rodillas y...

Lisa (con el ceño fruncido): Lo debemos tener para esa fecha, Doug. ¿De qué funcionalidad hablas?

Doug: Supongo que podemos tener la funcionalidad completa en la fecha establecida, pero tendríamos que retrasar el acceso por internet hasta el segundo incremento.

Lisa: Doug, es el acceso por internet lo que da a *CasaSegura* su "súper" atractivo. Toda nuestra campaña de publicidad va a girar alrededor de eso. Lo tenemos que tener…

Doug: Entiendo la situación, de verdad. El problema es que para dar acceso por internet tendríamos que tener un sitio web por completo seguro y en operación. Esto requiere tiempo y personal. También tenemos que elaborar mucha funcionalidad adicional en la primera entrega… no creo que podamos hacerlo con los recursos que tenemos.

Lisa (todavía frunce el ceño): Ya veo, pero tienes que imaginar una manera de hacerlo. Tiene importancia crítica para las funciones de seguridad del hogar y también para otras… éstas podrían esperar hasta las siguientes entregas… estoy de acuerdo con eso.

Lisa y Doug parecen estar en suspenso, pero todavía deben negociar una solución a este problema. ¿Pueden "ganar" los dos en este caso? Si usted fuera el mediador, ¿qué sugeriría?

5.7 VALIDACIÓN DE LOS REQUERIMIENTOS

A medida que se crea cada elemento del modelo de requerimientos, se estudia para detectar inconsistencias, omisiones y ambigüedades. Los participantes asignan prioridades a los requerimientos representados por el modelo y se agrupan en paquetes de requerimientos que se implementarán como incrementos del software. La revisión del modelo de requerimientos aborda las preguntas siguientes:

? Cuando se revisan los requerimientos, ¿qué preguntas deben plantearse?

- ¿Es coherente cada requerimiento con los objetivos generales del sistema o producto?
- ¿Se han especificado todos los requerimientos en el nivel apropiado de abstracción? Es decir, ¿algunos de ellos tienen un nivel de detalle técnico que resulta inapropiado en esta etapa?
- El requerimiento, ¿es realmente necesario o representa una característica agregada que tal vez no sea esencial para el objetivo del sistema?
- ¿Cada requerimiento está acotado y no es ambiguo?
- ¿Tiene atribución cada requerimiento? Es decir, ¿hay una fuente (por lo general una individual y específica) clara para cada requerimiento?
- ¿Hay requerimientos en conflicto con otros?

- ¿Cada requerimiento es asequible en el ambiente técnico que albergará el sistema o producto?
- Una vez implementado cada requerimiento, ¿puede someterse a prueba?
- El modelo de requerimientos, ¿refleja de manera apropiada la información, la función y el comportamiento del sistema que se va a construir?
- ¿Se ha "particionado" el modelo de requerimientos en forma que exponga información cada vez más detallada sobre el sistema?
- ¿Se ha usado el patrón de requerimientos para simplificar el modelo de éstos? ¿Se han validado todos los patrones de manera apropiada? ¿Son consistentes todos los patrones con los requerimientos del cliente?

Éstas y otras preguntas deben plantearse y responderse para garantizar que el modelo de requerimientos es una reflexión correcta sobre las necesidades del participante y que provee un fundamento sólido para el diseño.

5.8 Resumen

Las tareas de la ingeniería de requerimientos se realizan para establecer un fundamento sólido para el diseño y la construcción. La ingeniería de requerimientos ocurre durante las actividades de comunicación y modelado que se hayan definido para el proceso general del software. Los miembros del equipo de software llevan a cabo siete funciones de ingeniería de requerimientos: concepción, indagación, elaboración, negociación, especificación, validación y administración.

En la concepción del proyecto, los participantes establecen los requerimientos básicos del problema, definen las restricciones generales del proyecto, así como las características y funciones principales que debe presentar el sistema para cumplir sus objetivos. Esta información se mejora y amplía durante la indagación, actividad en la que se recaban los requerimientos y que hace uso de reuniones que lo facilitan, DFC y el desarrollo de escenarios de uso.

La elaboración amplía aún más los requerimientos en un modelo: una colección de elementos basados en escenarios, clases y comportamiento, y orientados al flujo. El modelo hace referencia a patrones de análisis: soluciones para problemas de análisis que se ha observado que son recurrentes en diferentes aplicaciones.

Conforme se identifican los requerimientos y se crea su modelo, el equipo de software y otros participantes negocian la prioridad, la disponibilidad y el costo relativo de cada requerimiento. Además, se valida cada requerimiento y su modelo como un todo comparado con las necesidades del cliente a fin de garantizar que va a construirse el sistema correcto.

Problemas y puntos por evaluar

5.1. ¿Por qué muchos desarrolladores de software no ponen atención suficiente a la ingeniería de requerimientos? ¿Existen algunas circunstancias que puedan ignorarse?

5.2. El lector tiene la responsabilidad de indagar los requerimientos de un cliente que dice estar demasiado ocupado para tener una reunión. ¿Qué debe hacer?

5.3. Analice algunos de los problemas que ocurren cuando los requerimientos deben indagarse para tres o cuatro clientes distintos.

5.4. ¿Por qué se dice que el modelo de requerimientos representa una fotografía instantánea del sistema en el tiempo?

5.5. Suponga que ha convencido al cliente (es usted muy buen vendedor) para que esté de acuerdo con todas las demandas que usted hace como desarrollador. ¿Eso lo convierte en un gran negociador? ¿Por qué?

5.6. Desarrolle al menos tres "preguntas libres de contexto" adicionales que podría plantear a un participante durante la concepción.

5.7. Desarrolle un "kit" para recabar requerimientos. Debe incluir un conjunto de lineamientos a fin de llevar a cabo la reunión para recabar requerimientos y los materiales que pueden emplearse para facilitar la creación de listas y otros objetos que ayuden a definir los requerimientos.

5.8. Su profesor formará grupos de cuatro a seis estudiantes. La mitad de ellos desempeñará el papel del departamento de mercadotecnia y la otra mitad adoptará el del equipo para la ingeniería de software. Su trabajo es definir los requerimientos para la función de seguridad de *CasaSegura* descrita en este capítulo. Efectúe una reunión para recabar los requerimientos con el uso de los lineamientos presentados en este capítulo.

5.9. Desarrolle un caso de uso completo para una de las actividades siguientes:

a) Hacer un retiro de efectivo en un cajero automático.
b) Usar su tarjeta de crédito para pagar una comida en un restaurante.
c) Comprar acciones en la cuenta en línea de una casa de bolsa.
d) Buscar libros (sobre un tema específico) en una librería en línea.
e) La actividad que especifique su profesor.

5.10. ¿Qué representan las "excepciones" en un caso de uso?

5.11. Describa con sus propias palabras lo que es un *patrón de análisis*.

5.12. Con el formato presentado en la sección 5.5.2, sugiera uno o varios patrones de análisis para los siguientes dominios de aplicación:

a) Software de contabilidad.
b) Software de correo electrónico.
c) Navegadores de internet.
d) Software de procesamiento de texto.
e) Software para crear un sitio web.
f) El dominio de aplicación que diga su profesor.

5.13. ¿Qué significa *ganar-ganar* en el contexto de una negociación durante la actividad de ingeniería de los requerimientos?

5.14. ¿Qué piensa que pasa cuando la validación de los requerimientos detecta un error? ¿Quién está involucrado en su corrección?

Lecturas adicionales y fuentes de información

La ingeniería de requerimientos se estudia en muchos libros debido a su importancia crítica para la creación exitosa de cualquier sistema basado en computadoras. Hood *et al.* (*Requirements Management*, Springer, 2007) analizan varios aspectos de la ingeniería de los requerimientos que incluyen tanto la ingeniería de sistemas como la de software. Young (*The Requirements Engineering Handbook*, Artech House Publishers, 2007) presenta un análisis profundo de las tareas de la ingeniería de requerimientos. Wiegers (*More About Software Requirements*, Microsoft Press, 2006) menciona muchas técnicas prácticas para recabar y administrar los requerimientos. Hull *et al.* (*Requirements Engineering*, 2a. ed., Springer-Verlag, 2004), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) y Sommerville y Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley, 1998) son sólo algunos de los muchos libros dedicados al tema. Gottesdiener (*Requirements by Collaboration: Workshops for Defining Needs*, Addison-Wesley, 2002) proporciona una guía útil para quienes deben generar un ambiente de colaboración a fin de recabar los requerimientos con los participantes.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) presenta una recopilación exhaustiva de los métodos y notación para el análisis de requerimientos. Weigers (*Software Requirements*, Microsoft Press, 1999) y Leffingwell *et al.* (*Managing Software Requirements: A Use Case Approach*, 2a. ed., Addison-Wesley, 2003) presentan una colección útil de las mejores prácticas respecto de los requerimientos y sugieren lineamientos prácticos para la mayoría de los aspectos del proceso de su ingeniería.

En Withall (*Software Requirement Patterns*, Microsoft Press, 2007) se describe la ingeniería de requerimientos desde un punto de vista basado en los patrones. Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) analiza técnicas avanzadas para desarrollar requerimientos de software. Windle y Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) estudian la ingeniería de los requerimientos en el contexto del proceso unificado y la notación UML. Alexander y Steven (*Writing Better Requirements*, Addison-Wesley, 2002) presentan un conjunto abreviado de lineamientos para escribir requerimientos claros, representarlos como escenarios y revisar el resultado final.

Es frecuente que el modelado de un caso de uso sea el detonante para crear todos los demás aspectos del modelo de análisis. El tema lo estudian mucho Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander y Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Leffingwell *et al.* (*Managing Software Requirements: A Use Case Approach*, 2a. ed., Addison-Wesley, 2003) presentan una colección útil de las mejores prácticas sobre los requerimientos. Bittner y Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01], Armour y Miller (*Advanced Use Cases Modeling: Software Systems*, Addison-Wesley, 2000) y Kulak *et al.* (*Use Cases: Requirements in Context*, Addison-Wesley, 2000) estudian la obtención de requerimientos con énfasis en el modelado del caso de uso.

En internet hay una variedad amplia de fuentes de información acerca de la ingeniería y análisis de los requerimientos. En el sitio web del libro, **www.mhhe.com/engcs/compsi/pressman/professional/olc/ser.htm**, se halla una lista actualizada de referencias en web que son relevantes para la ingeniería y análisis de los requerimientos.

CAPÍTULO

6 Modelado de los requerimientos: escenarios, información y clases de análisis

Conceptos clave

En el nivel técnico, la ingeniería de software comienza con una serie de tareas de modelado que conducen a la especificación de los requerimientos y a la representación de un diseño del software que se va a elaborar. El modelo de requerimientos[1] —un conjunto de modelos, en realidad— es la primera representación técnica de un sistema.

En un libro fundamental sobre métodos para modelar los requerimientos, Tom DeMarco [DeM79] describe el proceso de la manera siguiente:

> Al mirar retrospectivamente los problemas y las fallas detectados en la fase de análisis, concluyo que es necesario agregar lo siguiente al conjunto de objetivos de dicha fase. Debe ser muy fácil dar mantenimiento a los productos del análisis. Esto se aplica en particular al Documento de Objetivos [especificación de los requerimientos del software]. Los problemas grandes deben ser enfrentados con el empleo de un método eficaz para dividirlos. La especificación victoriana original resulta caduca. Deben usarse gráficas, siempre que sea posible. Es necesario diferenciar las consideraciones lógicas [esenciales] y las físicas [implementación]... Finalmente, se necesita... algo que ayude a dividir los requerimientos y a documentar dicha partición antes de elaborar la especificación... algunos medios para dar seguimiento a las interfaces y evaluarlas... nuevas herramientas para describir la lógica y la política, algo mejor que un texto narrativo.

Una mirada rápida

¿Qué es? La palabra escrita es un vehículo maravilloso para la comunicación, pero no necesariamente es la mejor forma de representar los requerimientos de software de computadora. El modelado de los requerimientos utiliza una combinación de texto y diagramas para ilustrarlos en forma que sea relativamente fácil de entender y, más importante, de revisar para corregir, completar y hacer congruente.

¿Quién lo hace? Un ingeniero de software (a veces llamado "analista") construye el modelo con el uso de los requerimientos recabados del cliente.

¿Por qué es importante? Para validar los requerimientos del software se necesita estudiarlos desde varios puntos de vista diferentes. En este capítulo se considerará el modelado de los requerimientos desde tres perspectivas distintas: modelos basados en el escenario, modelos de datos (información) y modelos basados en la clase. Cada una representa a los requerimientos en una "dimensión" diferente, con lo que aumenta la probabilidad de detectar errores, de que afloren las inconsistencias y de que se revelen las omisiones.

¿Cuáles son los pasos? El modelado basado en escenarios es una representación del sistema desde el punto de vista del usuario. El modelado basado en datos recrea el espacio de información e ilustra los objetos de datos que manipulará el software y las relaciones entre ellos. El modelado orientado a clases define objetos, atributos y relaciones. Una vez que se crean los modelos preliminares, se mejoran y analizan para evaluar si están claros y completos, y si son consistentes. En el capítulo 7 se amplían con representaciones adicionales las dimensiones del modelado descritas aquí, lo que da un punto de vista más sólido de los requerimientos.

¿Cuál es el producto final? Para construir el modelo de requerimientos, se escoge una amplia variedad de representaciones basadas en texto y en diagramas. Cada una de dichas representaciones da una perspectiva de uno o más de los elementos del modelo.

¿Cómo me aseguro de que lo hice bien? Los productos del trabajo para modelar los requerimientos deben revisarse para saber si son correctos, completos y consistentes. Deben reflejar las necesidades de todos los participantes y establecer el fundamento desde el que se realizará el diseño.

1 En ediciones anteriores de este libro, se usó el término *modelo de análisis*, en lugar de *modelo de requerimientos*. En esta edición, el autor decidió usar ambas expresiones para designar la actividad que define distintos aspectos del problema por resolver. *Análisis* es lo que ocurre cuando se obtienen los *requerimientos*.

Aunque DeMarco escribió hace más de un cuarto de siglo acerca de los atributos del modelado del análisis, sus comentarios aún son aplicables a los métodos y notación modernos del modelado de los requerimientos.

6.1 Análisis de los requerimientos

El análisis de los requerimientos da como resultado la especificación de las características operativas del software, indica la interfaz de éste y otros elementos del sistema, y establece las restricciones que limitan al software. El análisis de los requerimientos permite al profesional (sin importar si se llama *ingeniero de software*, *analista* o *modelista*) construir sobre los requerimientos básicos establecidos durante las tareas de concepción, indagación y negociación, que son parte de la ingeniería de los requerimientos (véase el capítulo 5).

Cita:

"Cualquier 'vista' de los requerimientos es insuficiente para entender o describir el comportamiento deseado de un sistema complejo."

Alan M. Davis

La acción de modelar los requerimientos da como resultado uno o más de los siguientes tipos de modelo:

- *Modelos basados en el escenario* de los requerimientos desde el punto de vista de distintos "actores" del sistema.
- *Modelos de datos*, que ilustran el dominio de información del problema.
- *Modelos orientados a clases*, que representan clases orientadas a objetos (atributos y operaciones) y la manera en la que las clases colaboran para cumplir con los requerimientos del sistema.
- *Modelos orientados al flujo*, que representan los elementos funcionales del sistema y la manera como transforman los datos a medida que se avanza a través del sistema.
- *Modelos de comportamiento*, que ilustran el modo en el que se comparte el software como consecuencia de "eventos" externos.

PUNTO CLAVE

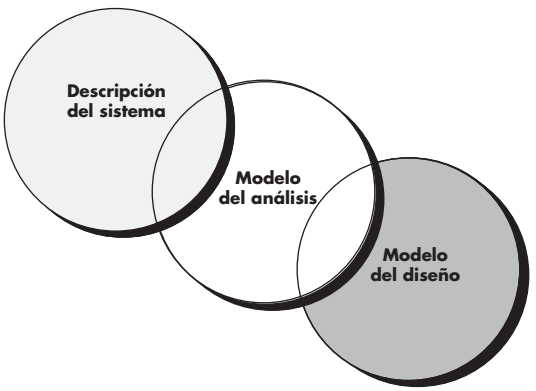
El modelo de análisis y la especificación de requerimientos proporcionan un medio para evaluar la calidad una vez construido el software.

Estos modelos dan al diseñador del software la información que se traduce en diseños de arquitectura, interfaz y componentes. Por último, el modelo de requerimientos (y la especificación de requerimientos de software) brinda al desarrollador y al cliente los medios para evaluar la calidad una vez construido el software.

Este capítulo se centra en el *modelado basado en escenarios*, técnica que cada vez es más popular entre la comunidad de la ingeniería de software; el *modelado basado en datos*, más especializado, apropiado en particular cuando debe crearse una aplicación o bien manipular un espacio complejo de información; y el *modelado orientado a clases*, representación de las clases orientada a objetos y a las colaboraciones resultantes que permiten que funcione el sistema. En

FIGURA 6.1

El modelo de requerimientos como puente entre la descripción del sistema y el modelo del diseño



el capítulo 7 se analizan los modelos orientados al flujo, al comportamiento, basados en el patrón y en *webapps*.

6.1.1 Objetivos y filosofía general

Cita:

"Los requerimientos no son arquitectura. No son diseño ni la interfaz de usuario. Los requerimientos son las necesidades."

Andrew Hunt y David Thomas

Durante el modelado de los requerimientos, la atención se centra en *qué*, no en *cómo*. ¿Qué interacción del usuario ocurre en una circunstancia particular?, ¿qué objetos manipula el sistema?, ¿qué funciones debe realizar el sistema?, ¿qué comportamientos tiene el sistema?, ¿qué interfaces se definen? y ¿qué restricciones son aplicables?[2]

En los capítulos anteriores se dijo que en esta etapa tal vez no fuera posible tener la especificación completa de los requerimientos. El cliente quizá no esté seguro de qué es lo que requiere con precisión para ciertos aspectos del sistema. Puede ser que el desarrollador esté inseguro de que algún enfoque específico cumpla de manera apropiada la función y el desempeño. Estas realidades hablan a favor de un enfoque iterativo para el análisis y el modelado de los requerimientos. El analista debe modelar lo que se sabe y usar el modelo como base para el diseño del incremento del software.[3]

PUNTO CLAVE

El modelo de análisis debe describir lo que quiere el cliente, establecer una base para el diseño y un objetivo para la validación.

El modelo de requerimientos debe lograr tres objetivos principales: 1) describir lo que requiere el cliente, 2) establecer una base para la creación de un diseño de software y 3) definir un conjunto de requerimientos que puedan validarse una vez construido el software. El modelo de análisis es un puente entre la descripción en el nivel del sistema que se centra en éste en lo general o en la funcionalidad del negocio que se logra con la aplicación de software, hardware, datos, personas y otros elementos del sistema y un diseño de software (véanse los capítulos 8 a 13) que describa la arquitectura de la aplicación del software, la interfaz del usuario y la estructura en el nivel del componente. Esta relación se ilustra en la figura 6.1.

Es importante observar que todos los elementos del modelo de requerimientos pueden rastrearse directamente hasta las partes del modelo del diseño. No siempre es posible la división clara entre las tareas del análisis y las del diseño en estas dos importantes actividades del modelado. Invariablemente, ocurre algo de diseño como parte del análisis y algo de análisis se lleva a cabo durante el diseño.

6.1.2 Reglas prácticas del análisis

Arlow y Neustadt [Arl02] sugieren cierto número de reglas prácticas útiles que deben seguirse cuando se crea el modelo del análisis:

? ¿Hay lineamientos básicos que nos ayuden a hacer el trabajo de análisis de los requerimientos?

- *El modelo debe centrarse en los requerimientos que sean visibles dentro del problema o dentro del dominio del negocio. El nivel de abstracción debe ser relativamente elevado.* "No se empantane en los detalles" [Arl02] que traten de explicar cómo funciona el sistema.
- *Cada elemento del modelo de requerimientos debe agregarse al entendimiento general de los requerimientos del software y dar una visión del dominio de la información, de la función y del comportamiento del sistema*.
- *Hay que retrasar las consideraciones de la infraestructura y otros modelos no funcionales hasta llegar a la etapa del diseño.* Es decir, quizá se requiera una base de datos, pero las clases necesarias para implementarla, las funciones requeridas para acceder a ella y el comportamiento que tendrá cuando se use sólo deben considerarse después de que se haya terminado el análisis del dominio del problema.

---

2 Debe notarse que, a medida que los clientes tienen más conocimientos tecnológicos, hay una tendencia hacia la especificación del *cómo* tanto como del *qué*. Sin embargo, la atención debe centrarse en el *qué*.

3 En un esfuerzo por entender mejor los requerimientos para el sistema, el equipo del software tiene la alternativa de escoger la creación de un prototipo (véase el capítulo 2).

Cita:

"Los problemas que es benéfico atacar demuestran su beneficio con un contragolpe."

Piet Hein

- *Debe minimizarse el acoplamiento a través del sistema.* Es importante representar las relaciones entre las clases y funciones. Sin embargo, si el nivel de "interconectividad" es extremadamente alto, deben hacerse esfuerzos para reducirlo.
- *Es seguro que el modelo de requerimientos agrega valor para todos los participantes.* Cada actor tiene su propio uso para el modelo. Por ejemplo, los participantes de negocios deben usar el modelo para validar los requerimientos; los diseñadores deben usarlo como pase para el diseño; el personal de aseguramiento de la calidad lo debe emplear como ayuda para planear las pruebas de aceptación.
- *Mantener el modelo tan sencillo como se pueda.* No genere diagramas adicionales si no agregan nueva información. No utilice notación compleja si basta una sencilla lista.

6.1.3 Análisis del dominio

WebRef

En la dirección **www.iturls.com/English/SoftwareEngineering/SE_mod5.asp**, existen muchos recursos útiles para el análisis del dominio.

Al estudiar la ingeniería de requerimientos (en el capítulo 5), se dijo que es frecuente que haya patrones de análisis que se repiten en muchas aplicaciones dentro de un dominio de negocio específico. Si éstos se definen y clasifican en forma tal que puedan reconocerse y aplicarse para resolver problemas comunes, la creación del modelo del análisis es más expedita. Más importante aún es que la probabilidad de aplicar patrones de diseño y componentes de software ejecutable se incrementa mucho. Esto mejora el tiempo para llegar al mercado y reduce los costos de desarrollo.

Pero, ¿cómo se reconocen por primera vez los patrones de análisis y clases? ¿Quién los define, clasifica y prepara para usarlos en los proyectos posteriores? La respuesta a estas preguntas está en el *análisis del dominio*. Firesmith [Fir93] lo describe del siguiente modo:

> El análisis del dominio del software es la identificación, análisis y especificación de los requerimientos comunes, a partir de un dominio de aplicación específica, normalmente para usarlo varias veces en múltiples proyectos dentro del dominio de la aplicación [...] [El análisis del dominio orientado a objetos es] la identificación, análisis y especificación de capacidades comunes y reutilizables dentro de un dominio de aplicación específica en términos de objetos, clases, subensambles y estructuras comunes.

PUNTO CLAVE

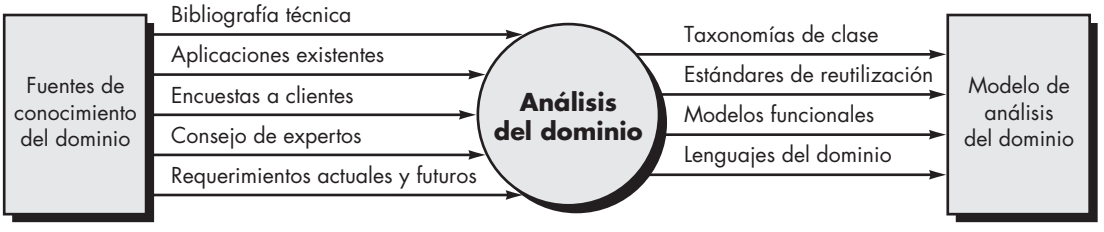
El análisis del dominio no busca en una aplicación específica, sino en el dominio en el que reside la aplicación. El objetivo es identificar elementos comunes para la solución de problemas, que sean útiles en todas las aplicaciones dentro del dominio.

El "dominio de aplicación específica" se extiende desde el control electrónico de aviones hasta la banca, de los juegos de video en multimedios al software incrustado en equipos médicos. La meta del análisis del dominio es clara: encontrar o crear aquellas clases o patrones de análisis que sean aplicables en lo general, de modo que puedan volverse a usar.[4]

Con el empleo de la terminología que se introdujo antes en este libro, el análisis del dominio puede considerarse como una actividad sombrilla para el proceso del software. Esto significa que el análisis del dominio es una actividad de la ingeniería de software que no está conectada

FIGURA 6.2

Entradas y salidas para el análisis del dominio



4 Un punto de vista complementario del análisis del dominio "involucra el modelado de éste, de manera que los ingenieros del software y otros participantes aprendan más al respecto [...] no todas las clases de dominio necesariamente dan como resultado el desarrollo de clases reutilizables [...]" [Let03a].

CasaSegura

Análisis del dominio

La escena: Oficina de Doug Miller, después de una reunión con personal de mercadotecnia.

Participantes: Doug Miller, gerente de ingeniería de software, y Vinod Raman, miembro del equipo de ingeniería de software.

La conversación:

Doug: Te necesito para un proyecto especial, Vinod. Voy a retirarte de las reuniones para recabar los requerimientos.

Vinod (con el ceño fruncido): Muy mal. Ese formato en verdad funciona… Estaba sacando algo de ahí. ¿Qué pasa?

Doug: Jamie y Ed te cubrirán. De cualquier manera, el departamento de mercadotecnia insiste en que en la primera entrega de *CasaSegura* dispongamos de la capacidad de acceso por internet junto con la función de seguridad para el hogar. Estamos bajo fuego en esto… sin tiempo ni personal suficiente, así que tenemos que resolver ambos problemas a la vez: la interfaz de PC y la interfaz de web.

Vinod (confundido): No sabía que el plan era entregar… ni siquiera hemos terminado de recabar los requerimientos.

Doug (con una sonrisa tenue): Lo sé, pero los plazos son tan breves que decidí comenzar ya la estrategia con mercadotecnia… de cualquier modo, revisaremos cualquier plan tentativo una vez que tengamos la información de todas las juntas que se efectuarán para recabar los requerimientos.

Vinod: Está bien, ¿entonces? ¿Qué quieres que haga?

Doug: ¿Sabes qué es el "análisis del dominio"?

Vinod: Algo sé. Buscas patrones similares en aplicaciones que hagan lo mismo que la que estés elaborando. Entonces, si es posible, calcas los patrones y los reutilizas en tu trabajo.

Doug: No estoy seguro de que la palabra sea *calcar*, pero básicamente tienes razón. Lo que me gustaría que hicieras es que comenzaras a buscar interfaces de usuario ya existentes para sistemas que controlen algo como *CasaSegura*. Quiero que propongas un conjunto de patrones y clases de análisis que sean comunes tanto a la interfaz basada en PC que estará en el hogar como a la basada en un navegador al que se accederá por internet.

Vinod: Ahorraríamos tiempo si las hiciéramos iguales… ¿por qué no las hacemos así?

Doug: Ah… es grato tener gente que piense como lo haces tú. Ése es el meollo del asunto: ahorraremos tiempo y esfuerzo si las dos interfaces son casi idénticas; las implementamos con el mismo código y acabamos con la insistencia de mercadotecnia.

Vinod: ¿Entonces, qué quieres?, ¿clases, patrones de análisis, patrones de diseño?

Doug: Todo eso. Nada formal en este momento. Sólo quiero que comencemos despacio con nuestros trabajos de análisis interno y de diseño.

Vinod: Iré a nuestra biblioteca de clases y veré qué tenemos. También usaré un formato de patrones que vi en un libro que leí hace unos meses.

Doug: Bien. Manos a la obra.

con ningún proyecto de software. En cierta forma, el papel del analista del dominio es similar al de un maestro herrero en un ambiente de manufactura pesada. El trabajo del herrero es diseñar y fabricar herramientas que utilicen muchas personas que hacen trabajos similares pero no necesariamente iguales. El papel del analista de dominio[5] es descubrir y definir patrones de análisis, clases de análisis e información relacionada que pueda ser utilizada por mucha gente que trabaje en aplicaciones similares, pero que no son necesariamente las mismas.

Cita:

"… el análisis es frustrante, está lleno de relaciones interpersonales complejas, indefinidas y difíciles. En una palabra, es fascinante. Una vez atrapado, los antiguos y fáciles placeres de la construcción de sistemas nunca más volverán a satisfacerte."

Tom DeMarco

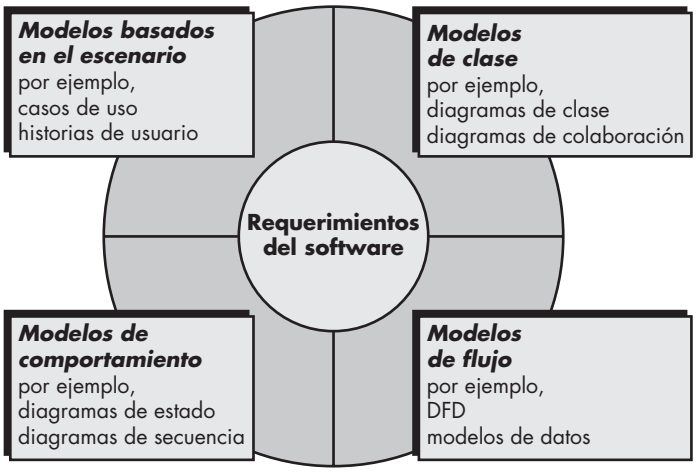
La figura 6.2 [Ara89] ilustra entradas y salidas clave para el proceso de análisis del dominio. Las fuentes de conocimiento del dominio se mapean con el fin de identificar los objetos que pueden reutilizarse a través del dominio.

6.1.4 Enfoques del modelado de requerimientos

Un enfoque del modelado de requerimientos, llamado *análisis estructurado*, considera que los datos y los procesos que los transforman son entidades separadas. Los objetos de datos se modelan de modo que se definan sus atributos y relaciones. Los procesos que manipulan a los objetos de datos se modelan en forma que se muestre cómo transforman a los datos a medida que los objetos que se corresponden con ellos fluyen por el sistema.

5 No suponga que el ingeniero de software no necesita entender el dominio de la aplicación tan sólo porque hay un analista del dominio trabajando. Todo miembro del equipo del software debe entender algo del dominio en el que se va a colocar el software.

FIGURA 6.3
Elementos del modelo de análisis



¿Cuáles son los diferentes puntos de vista que se usan para describir el modelo de requerimientos?

Un segundo enfoque del modelado del análisis, llamado *análisis orientado a objetos*, se centra en la definición de las clases y en la manera en la que colaboran uno con el otro para cumplir los requerimientos. El UML y el proceso unificado (véase el capítulo 2) están orientados a objetos, sobre todo.

Aunque el modelo de requerimientos propuesto en este libro combina características de ambos enfoques, los equipos de software escogen con frecuencia uno y excluyen todas las representaciones del otro. La pregunta no es cuál es mejor, sino qué combinación de representaciones proporcionará a los participantes el mejor modelo de requerimientos del software y el puente más eficaz para el diseño del mismo.

Cita:

"¿Por qué construimos modelos? ¿Por qué no construir sólo el sistema? La respuesta es que los construimos para que resalten o enfaticen ciertas características críticas de un sistema, al tiempo que ignoran otros aspectos del mismo."

Ed Yourdon

Cada elemento del modelo de requerimientos (véase la figura 6.3) presenta el problema desde diferentes puntos de vista. Los elementos basados en el escenario ilustran cómo interactúa el usuario con el sistema y la secuencia específica de actividades que ocurren cuando se utiliza el software. Los elementos basados en la clase modelan los objetos que el sistema manipulará, las operaciones que se aplicarán a ellos para realizar dicha manipulación, las relaciones (algunas jerárquicas) entre los objetos y las colaboraciones que ocurrirán entre las clases que se definan. Los elementos del comportamiento ilustran la forma en la que los eventos externos cambian el estado del sistema o las clases que residen dentro de éste. Por último, los elementos orientados al flujo representan al sistema como una transformación de la información e ilustran la forma en la que se transforman los objetos de datos cuando fluyen a través de las distintas funciones del sistema.

El modelado del análisis lleva a la obtención de cada uno de estos elementos de modelado. Sin embargo, el contenido específico de cada elemento (por ejemplo, los diagramas que se emplean para construir el elemento y el modelo) tal vez difiera de un proyecto a otro. Como se ha dicho varias veces en este libro, el equipo del software debe trabajar para mantenerlo sencillo. Sólo deben usarse elementos de modelado que agreguen valor al modelo.

6.2 MODELADO BASADO EN ESCENARIOS

Aunque el éxito de un sistema o producto basado en computadora se mide de muchas maneras, la satisfacción del usuario ocupa el primer lugar de la lista. Si se entiende cómo desean interactuar los usuarios finales (y otros actores) con un sistema, el equipo del software estará mejor preparado para caracterizar adecuadamente los requerimientos y hacer análisis significativos y

modelos del diseño. Entonces, el modelado de los requerimientos con UML[6] comienza con la creación de escenarios en forma de casos de uso, diagramas de actividades y diagramas tipo carril de natación.

6.2.1 Creación de un caso preliminar de uso

Cita:

"[Los casos de uso] simplemente son una ayuda para definir lo que existe fuera del sistema (actores) y lo que debe realizar el sistema (casos de uso)."

Ivar Jacobson

Alistair Cockburn caracteriza un caso de uso como un "contrato para el comportamiento" [Coc01b]. Como se dijo en el capítulo 5, el "contrato" define la forma en la que un actor[7] utiliza un sistema basado en computadora para alcanzar algún objetivo. En esencia, un caso de uso capta las interacciones que ocurren entre los productores y consumidores de la información y el sistema en sí. En esta sección se estudiará la forma en la que se desarrollan los casos de uso como parte de los requerimientos de la actividad de modelado.[8]

En el capítulo 5 se dijo que un caso de uso describe en lenguaje claro un escenario específico desde el punto de vista de un actor definido. Pero, ¿cómo se sabe sobre qué escribir, cuánto escribir sobre ello, cuán detallada hacer la descripción y cómo organizarla? Son preguntas que deben responderse si los casos de uso han de tener algún valor como herramienta para modelar los requerimientos.

CONSEJO

En ciertas situaciones, los casos de uso se convierten en el mecanismo dominante de la ingeniería de requerimientos. Sin embargo, esto no significa que deban descartarse otros métodos de modelado cuando resulten apropiados.

¿Sobre qué escribir? Las dos primeras tareas de la ingeniería de requerimientos —concepción e indagación— dan la información que se necesita para comenzar a escribir casos de uso. Las reuniones para recabar los requerimientos, el DEC, y otros mecanismos para obtenerlos se utilizan para identificar a los participantes, definir el alcance del problema, especificar los objetivos operativos generales, establecer prioridades, delinear todos los requerimientos funcionales conocidos y describir las cosas (objetos) que serán manipuladas por el sistema.

Para comenzar a desarrollar un conjunto de casos de uso, se enlistan las funciones o actividades realizadas por un actor específico. Éstas se obtienen de una lista de las funciones requeridas del sistema, por medio de conversaciones con los participantes o con la evaluación de los diagramas de actividades (véase la sección 6.3.1) desarrollados como parte del modelado de los requerimientos.

CASASEGURA

Desarrollo de otro escenario preliminar de uso

La escena: Sala de juntas, durante la segunda reunión para recabar los requerimientos.

Participantes: Jamie Lazar, miembro del equipo del software; Ed Robbins, integrante del equipo del software; Doug Miller, gerente de ingeniería de software; tres miembros de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Es hora de que hablemos sobre la función de vigilancia de *CasaSegura*. Vamos a desarrollar un escenario de usuario que accede a la función de vigilancia.

Jamie: ¿Quién juega el papel del actor aquí?

Facilitador: Creo que Meredith (persona de mercadotecnia) ha estado trabajando en dicha funcionalidad. ¿Por qué no adoptas tú ese papel?

Meredith: Quieres que lo hagamos de la misma forma que la vez pasada, ¿verdad?

Facilitador: Sí... en cierto modo.

Meredith: Bueno, es obvio que la razón de la vigilancia es permitir que el propietario de la casa la revise cuando se encuentre fuera, así como poder grabar y reproducir el video que se grabe... esa clase de cosas.

Ed: ¿Usaremos compresión para guardar el video?

---

6 En todo el libro se usará UML como notación para elaborar modelos. En el apéndice 1 se ofrece un método breve de enseñanza para aquellos lectores que no estén familiarizados con lo más básico de dicha notación.

7 Un actor no es una persona específica sino el rol que desempeña ésta (o un dispositivo) en un contexto específico. Un actor "llama al sistema para que entregue uno de sus servicios" [Coc01b].

8 Los casos de uso son una parte del modelado del análisis de importancia especial para las interfaces. El análisis de la interfaz se estudia en detalle en el capítulo 11.

Facilitador: Buena pregunta, Ed, pero por ahora pospondremos los aspectos de la implementación. ¿Meredith?

Meredith: Bien, básicamente hay dos partes en la función de vigilancia... la primera configura el sistema, incluso un plano de la planta —tiene que haber herramientas que ayuden al propietario a hacer esto—, y la segunda parte es la función real de vigilancia. Como el plano es parte de la actividad de configuración, me centraré en la función de vigilancia.

Facilitador (sonríe): Me quitaste las palabras de la boca.

Meredith: Mmm... quiero tener acceso a la función de vigilancia, ya sea por PC o por internet. Tengo la sensación de que el acceso por internet se usaría con más frecuencia. De cualquier manera, quisiera poder mostrar vistas de la cámara en una PC y controlar el ángulo y acercamiento de una cámara en particular. Especificaría la cámara seleccionándola en el plano de la casa. También quiero poder bloquear el acceso a una o más cámaras con una clave determinada. Además, desearía tener la opción de ver pequeñas ventanas con vistas de todas las cámaras y luego escoger una que desee agrandar.

Jamie: Ésas se llaman vistas reducidas.

Meredith: Bien, entonces quiero vistas reducidas de todas las cámaras. También quisiera que la interfaz de la función de vigilancia tuviera el mismo aspecto y sensación que todas las demás del sistema *CasaSegura*. Quiero que sea intuitiva, lo que significa que no tenga que leer un manual para usarla.

Facilitador: Buen trabajo. Ahora, veamos esta función con un poco más de detalle…

La función (subsistema) de vigilancia de *CasaSegura* estudiada en el recuadro identifica las funciones siguientes (lista abreviada) que va a realizar el actor **propietario**:

- Seleccionar cámara para ver.
- Pedir vistas reducidas de todas las cámaras.
- Mostrar vistas de las cámaras en una ventana de PC.
- Controlar el ángulo y acercamiento de una cámara específica.
- Grabar la salida de cada cámara en forma selectiva.
- Reproducir la salida de una cámara.
- Acceder por internet a la vigilancia con cámaras.

A medida que avanzan las conversaciones con el participante (quien juega el papel de propietario), el equipo que recaba los requerimientos desarrolla casos de uso para cada una de las funciones estudiadas. En general, los casos de uso se escriben primero en forma de narración informal. Si se requiere más formalidad, se reescribe el mismo caso con el empleo de un formato estructurado, similar al propuesto en el capítulo y que se reproduce en un recuadro más adelante, en esta sección.

Para ilustrar esto, considere la función *acceder a la vigilancia con cámaras por internet-mostrar vistas de cámaras* **(AVC-MVC)**. El participante que tenga el papel del actor llamado **propietario** escribiría una narración como la siguiente:

Caso de uso: acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC)

Actor: propietario

Si estoy en una localidad alejada, puedo usar cualquier PC con un software de navegación apropiado para entrar al sitio web de *Productos CasaSegura*. Introduzco mi identificación de usuario y dos niveles de claves; una vez validadas, tengo acceso a toda la funcionalidad de mi sistema instalado. Para acceder a la vista de una cámara específica, selecciono "vigilancia" de los botones mostrados para las funciones principales. Luego selecciono "escoger una cámara" y aparece el plano de la casa. Después elijo la cámara que me interesa. Alternativamente, puedo ver la vista de todas las cámaras simultáneamente si selecciono "todas las cámaras". Una vez que escojo una, selecciono "vista" y en la ventana que cubre la cámara aparece una vista con velocidad de un cuadro por segundo. Si quiero cambiar entre las cámaras, selecciono "escoger una cámara" y desaparece la vista original y de nuevo se muestra el plano de la casa. Después, selecciono la cámara que me interesa. Aparece una nueva ventana de vistas.

Una variación de la narrativa del caso de uso presenta la interacción como una secuencia ordenada de acciones del usuario. Cada acción está representada como enunciado declarativo. Al visitar la función **ACS-DCV**, se escribiría lo siguiente:

Caso de uso: acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC)

Actor: propietario

1. El propietario accede al sitio web *Productos CasaSegura*.
2. El propietario introduce su identificación de usuario.
3. El propietario escribe dos claves (cada una de al menos ocho caracteres de longitud).
4. El sistema muestra los botones de todas las funciones principales.
5. El propietario selecciona "vigilancia" de los botones de las funciones principales.
6. El propietario elige "seleccionar una cámara".
7. El sistema presenta el plano de la casa.
8. El propietario escoge el ícono de una cámara en el plano de la casa.
9. El propietario selecciona el botón "vista".
10. El sistema presenta la ventana de vista identificada con la elección de la cámara.
11. El sistema muestra un video dentro de la ventana a velocidad de un cuadro por segundo.

Cita:

"Los casos de uso se emplean en muchos procesos [de software]. Nuestro favorito es el que es iterativo y guiado por el riesgo."

Gerl Schneider y Jason Winters

Es importante observar que esta presentación en secuencia no considera interacciones alternativas (la narración fluye con más libertad y representa varias alternativas). Los casos de este tipo en ocasiones se denominan *escenarios primarios* [Sch98a].

6.2.2 Mejora de un caso de uso preliminar

Para entender por completo la función que describe un caso de uso, es esencial describir interacciones alternativas. Después se evalúa cada paso en el escenario primario, planteando las preguntas siguientes [Sch98a]:

? Cuando desarrollo un caso de uso, ¿cómo examino los cursos alternativos de acción?

- *¿El actor puede emprender otra acción en este punto?*
- *¿Es posible que el actor encuentre alguna condición de error en este punto?* Si así fuera, ¿cuál podría ser?
- *En este punto, ¿es posible que el actor encuentre otro comportamiento (por ejemplo, alguno que sea invocado por cierto evento fuera del control del actor)?* En ese caso, ¿cuál sería?

Las respuestas a estas preguntas dan como resultado la creación de un conjunto de *escenarios secundarios* que forman parte del caso de uso original, pero que representan comportamientos alternativos. Por ejemplo, considere los pasos 6 y 7 del escenario primario ya descrito:

6. El propietario elige "seleccionar una cámara".
7. El sistema presenta el plano de la casa.

¿El actor puede emprender otra acción en este punto? La respuesta es "sí". Al analizar la narración de flujo libre, el actor puede escoger mirar vistas de todas las cámaras simultáneamente. Entonces, un escenario secundario sería "observar vistas instantáneas de todas las cámaras".

¿Es posible que el actor encuentre alguna condición de error en este punto? Cualquier número de condiciones de error puede ocurrir cuando opera un sistema basado en computadora. En este contexto, sólo se consideran las condiciones que sean probables como resultado directo de la acción descrita en los pasos 6 o 7. De nuevo, la respuesta es "sí". Tal vez nunca se haya configurado un plano con íconos de cámara. Entonces, elegir "seleccionar una cámara" da como

resultado una condición de error: "No hay plano configurado para esta casa."[9] Esta condición de error se convierte en un escenario secundario.

En este punto, ¿es posible que el actor encuentre otro comportamiento (por ejemplo, alguno que sea invocado por cierto evento fuera del control del actor)? Otra vez, la respuesta es "sí". A medida que ocurran los pasos 6 y 7, el sistema puede hallar una condición de alarma. Esto dará como resultado que el sistema desplegará una notificación especial de alarma (tipo, ubicación, acción del sistema) y proporcionará al actor varias opciones relevantes según la naturaleza de la alarma. Como este escenario secundario puede ocurrir en cualquier momento para prácticamente todas las interacciones, no se vuelve parte del caso de uso **AVC-MVC**. En vez de ello, se desarrollará un caso de uso diferente —**Condición de alarma encontrada**— al que se hará referencia desde otros casos según se requiera.

Cada una de las situaciones descritas en los párrafos precedentes se caracteriza como una excepción al caso de uso. Una *excepción* describe una situación (ya sea condición de falla o alternativa elegida por el actor) que ocasiona que el sistema presente un comportamiento algo distinto.

Cockburn [Coc01b] recomienda el uso de una sesión de "lluvia de ideas" para obtener un conjunto razonablemente complejo de excepciones para cada caso de uso. Además de las tres preguntas generales ya sugeridas en esta sección, también deben explorarse los siguientes aspectos:

- *¿Existen casos en los que ocurra alguna "función de validación" durante este caso de uso?* Esto implica que la función de validación es invocada y podría ocurrir una potencial condición de error.
- *¿Hay casos en los que una función (o actor) de soporte falle en responder de manera apropiada?* Por ejemplo, una acción de usuario espera una respuesta pero la función que ha de responder se cae.
- *¿El mal desempeño del sistema da como resultado acciones inesperadas o impropias?* Por ejemplo, una interfaz con base en web responde con demasiada lentitud, lo que da como resultado que un usuario haga selecciones múltiples en un botón de procesamiento. Estas selecciones se forman de modo equivocado y, en última instancia, generan un error.

La lista de extensiones desarrollada como consecuencia de preguntar y responder estas preguntas debe "racionalizarse" [Coc01b] con el uso de los siguientes criterios: una excepción debe describirse dentro del caso de uso si el software la puede detectar y debe manejarla una vez detectada. En ciertos casos, una excepción precipitará el desarrollo de otro caso de uso (el de manejar la condición descrita).

6.2.3 Escritura de un caso de uso formal

En ocasiones, para modelar los requerimientos es suficiente con los casos de uso informales presentados en la sección 6.2.1. Sin embargo, cuando un caso de uso involucra una actividad crítica o cuando describe un conjunto complejo de etapas con un número significativo de excepciones, es deseable un enfoque más formal.

El caso de uso **AVC-MVC** mostrado en el recuadro de la página 136 sigue el guión común para los casos de uso formales. El *objetivo en contexto* identifica el alcance general del caso de

9 En este caso, otro actor, **administrador del sistema**, tendría que configurar el plano de la casa, instalar e inicializar todas las cámaras (por ejemplo, asignar una identificación a los equipos) y probar cada una para garantizar que se encuentren accesibles por el sistema y a través del plano de la casa.

uso. La *precondición* describe lo que se sabe que es verdadero antes de que inicie el caso de uso. El *disparador* (o *trigger*) identifica el evento o condición que "hace que comience el caso de uso" [Coc01b]. El *escenario* enlista las acciones específicas que requiere el actor, y las respuestas apropiadas del sistema. Las *excepciones* identifican las situaciones detectadas cuando se mejora el caso de uso preliminar (véase la sección 6.2.2). Pueden incluirse o no encabezados adicionales y se explican por sí mismos en forma razonable.

CASASEGURA

Formato de caso de uso para vigilancia

Caso de uso: Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC).

Iteración: 2, última modificación: 14 de enero por V. Raman.

Actor principal: Propietario.

Objetivo en contexto: Ver la salida de las cámaras colocadas en la casa desde cualquier ubicación remota por medio de internet.

Precondiciones: El sistema debe estar configurado por completo; deben obtenerse las identificaciones y claves de usuario apropiadas.

Disparador: El propietario decide ver dentro de la casa mientras está fuera.

Escenario:

1. El propietario se registra en el sitio web *Productos CasaSegura*.
2. El propietario introduce su identificación de usuario.
3. El propietario proporciona dos claves (cada una con longitud de al menos ocho caracteres).
4. El sistema despliega todos los botones de las funciones principales.
5. El propietario selecciona "vigilancia" entre los botones de funciones principales.
6. El propietario escoge "seleccionar una cámara".
7. El sistema muestra el plano de la casa.
8. El propietario selecciona un ícono de cámara en el plano de la casa.
9. El propietario pulsa el botón "vista".
10. El sistema muestra la ventana de la vista de la cámara identificada.
11. El sistema presenta una salida de video dentro de la ventana de vistas, con una velocidad de un cuadro por segundo.

Excepciones:

1. La identificación o las claves son incorrectas o no se reconocen (véase el caso de uso **Validar identificación y claves**).
2. La función de vigilancia no está configurada para este sistema (el sistema muestra el mensaje de error apropiado; véase el caso de uso **Configurar la función de vigilancia**).
3. El propietario selecciona "Mirar vistas reducidas de todas las cámaras" (véase el caso de uso **Mirar vistas reducidas de todas las cámaras**).
4. No se dispone o no se ha configurado el plano de la casa (se muestra el mensaje de error apropiado y véase el caso de uso **Configurar plano de la casa**).
5. Se encuentra una condición de alarma (véase el caso de uso **Condición de alarma encontrada**).

Prioridad: Moderada, por implementarse después de las funciones básicas.

Cuándo estará disponible: En el tercer incremento.

Frecuencia de uso: Frecuencia moderada.

Canal al actor: A través de un navegador con base en PC y conexión a internet.

Actores secundarios: Administrador del sistema, cámaras.

Canales a los actores secundarios:

1. Administrador del sistema: sistema basado en PC.
2. Cámaras: conectividad inalámbrica.

Asuntos pendientes:

1. ¿Qué mecanismos protegen el uso no autorizado de esta capacidad por parte de los empleados de *Productos CasaSegura*?
2. Es suficiente la seguridad? El acceso ilegal a esta característica representaría una invasión grave de la privacidad.
3. ¿Será aceptable la respuesta del sistema por internet dado el ancho de banda que requieren las vistas de las cámaras?
4. ¿Desarrollaremos una capacidad que provea el video a una velocidad más alta en cuadros por segundo cuando se disponga de conexiones con un ancho de banda mayor?

WebRef

¿Cuándo se ha terminado de escribir casos de uso? Para un análisis benéfico de esto, consulte la dirección **ootips.org/use-cases-done.html**

En muchos casos, no hay necesidad de crear una representación gráfica de un escenario de uso. Sin embargo, la representación con diagramas facilita la comprensión, en particular cuando el escenario es complejo. Como ya se dijo en este libro, UML cuenta con la capacidad de hacer diagramas de casos de uso. La figura 6.4 ilustra un diagrama de caso de uso preliminar para el producto *CasaSegura*. Cada caso de uso está representado por un óvalo. En esta sección sólo se estudia el caso de uso **AVC-MVC**.

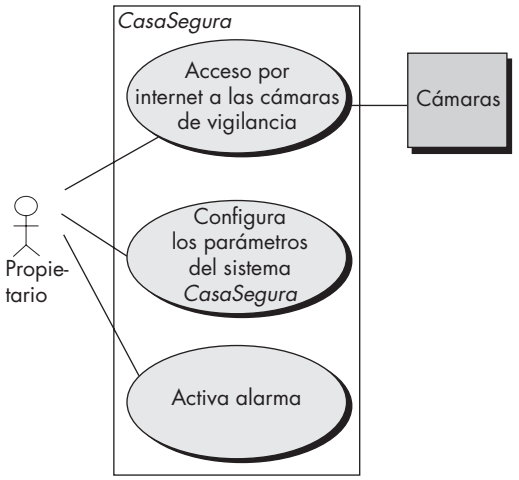


FIGURA 6.4
Diagrama de caso de uso preliminar para el sistema *CasaSegura*

Toda notación de modelado tiene sus limitaciones, y la del caso de uso no es la excepción. Como cualquier otra forma de descripción escrita, un caso de uso es tan bueno como lo sea(n) su(s) autor(es). Si la descripción es poco clara, el caso de uso será confuso o ambiguo. Un caso de uso se centra en los requerimientos funcionales y de comportamiento, y por lo general es inapropiado para requerimientos disfuncionales. Para situaciones en las que el modelo de requerimientos deba tener detalle y precisión significativos (por ejemplo, sistemas críticos de seguridad), tal vez no sea suficiente un caso de uso.

Sin embargo, el modelado basado en escenarios es apropiado para la gran mayoría de todas las situaciones que encontrará un ingeniero de software. Si se desarrolla bien, el caso de uso proporciona un beneficio sustancial como herramienta de modelado.

6.3 MODELOS UML QUE PROPORCIONAN EL CASO DE USO

Hay muchas situaciones de modelado de requerimientos en las que un modelo basado en texto —incluso uno tan sencillo como un caso de uso— tal vez no brinde información en forma clara y concisa. En tales casos, es posible elegir de entre una amplia variedad de modelos UML gráficos.

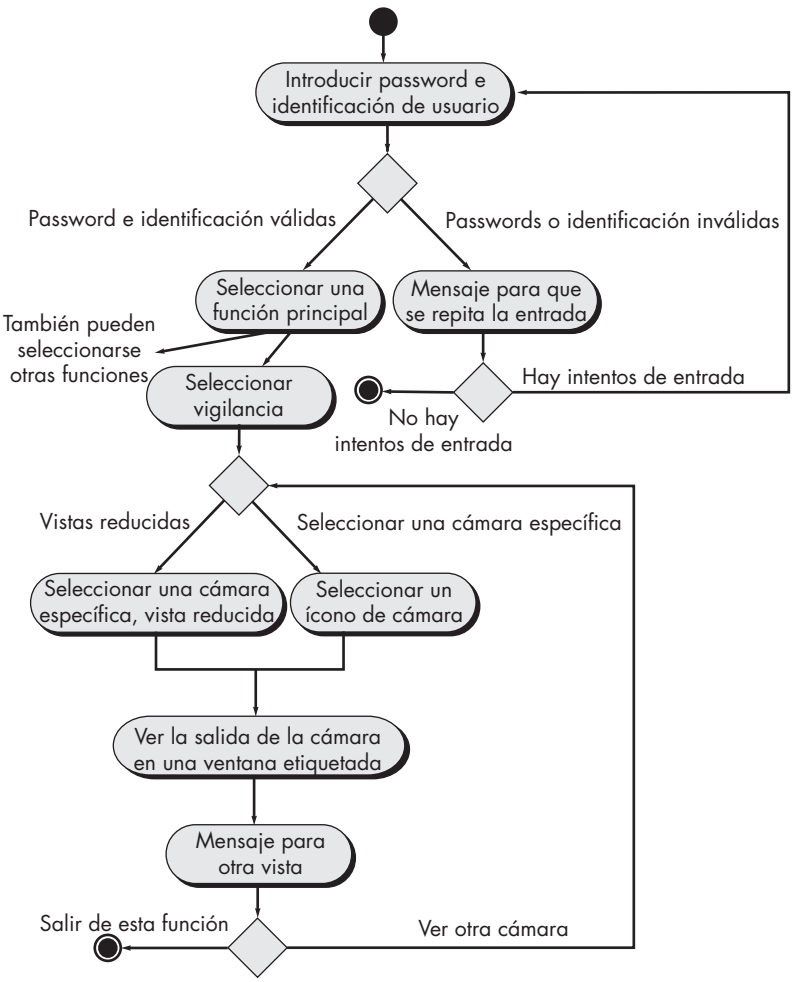
6.3.1 Desarrollo de un diagrama de actividades

PUNTO CLAVE
Un diagrama de actividades UML representa las acciones y decisiones que ocurren cuando se realiza cierta función.

El diagrama de actividad UML enriquece el caso de uso al proporcionar una representación gráfica del flujo de interacción dentro de un escenario específico. Un diagrama de actividades es similar a uno de flujo, y utiliza rectángulos redondeados para denotar una función específica del sistema, flechas para representar flujo a través de éste, rombos de decisión para ilustrar una ramificación de las decisiones (cada flecha que salga del rombo se etiqueta) y líneas continuas para indicar que están ocurriendo actividades en paralelo. En la figura 6.5 se presenta un diagrama de actividades para el caso de uso **AVC-MVC**. Debe observarse que el diagrama de actividades agrega detalles adicionales que no se mencionan directamente (pero que están implícitos) en el caso de uso.

Por ejemplo, un usuario quizá sólo haga algunos intentos de introducir su **identificación** y **password**. Esto se representa por el rombo de decisión debajo de "Mensaje para que se repita la entrada".

FIGURA 6.5
Diagrama de actividades para la función Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras.



6.3.2 Diagramas de canal (*swimlane*)

PUNTO CLAVE
Un diagrama de canal (*swimlane*) representa el flujo de acciones y decisiones e indica qué actores efectúan cada una de ellas.

El *diagrama de canal* de UML es una variación útil del diagrama de actividades y permite representar el flujo de actividades descritas por el caso de uso; al mismo tiempo, indica qué actor (si hubiera muchos involucrados en un caso específico de uso) o clase de análisis (se estudia más adelante, en este capítulo) es responsable de la acción descrita por un rectángulo de actividad. Las responsabilidades se representan con segmentos paralelos que dividen el diagrama en forma vertical, como los canales o carriles de una alberca.

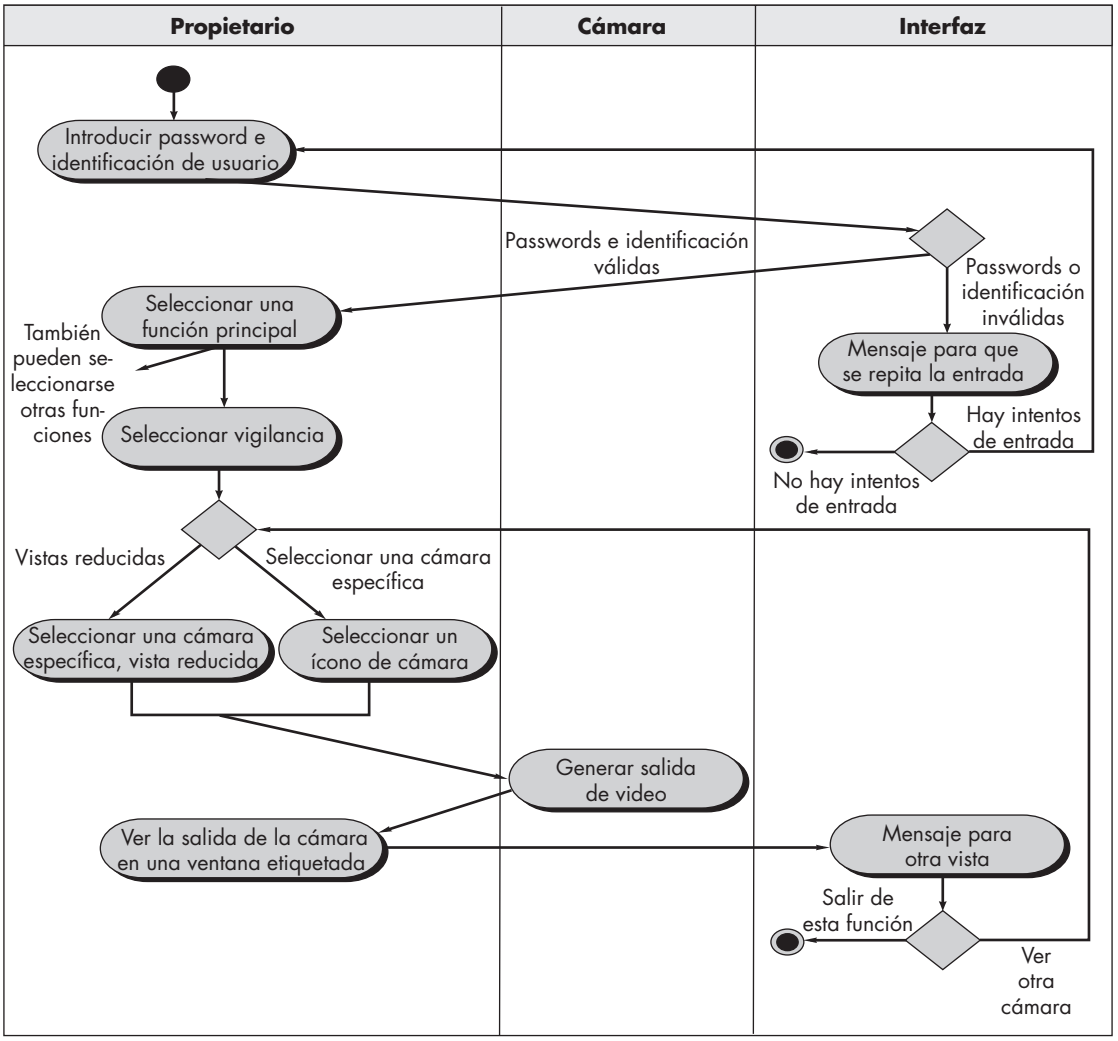
Son tres las clases de análisis: **Propietario**, **Cámara** e **Interfaz**, que tienen responsabilidad directa o indirecta en el contexto del diagrama de actividades representado en la figura 6.5. En relación con la figura 6.6, el diagrama de actividades se reacomodó para que las actividades asociadas con una clase de análisis particular queden dentro del canal de dicha clase. Por ejemplo, la clase **Interfaz** representa la interfaz de usuario como la ve el propietario. El diagrama de actividades tiene dos mensajes que son responsabilidad de la interfaz: "mensaje para que se repita la entrada" y "mensaje para otra vista". Estos mensajes y las decisiones asociadas con ellos caen dentro del canal **Interfaz**. Sin embargo, las flechas van de ese canal de regreso al de **Propietario**, donde ocurren las acciones de éste.

Cita:
"Un buen modelo guía el pensamiento; uno malo lo desvía."
Brian Marick

Los casos de uso, junto con los diagramas de actividades y de canal, están orientados al procedimiento. Representan la manera en la que los distintos actores invocan funciones específicas (u otros pasos del procedimiento) para satisfacer los requerimientos del sistema. Pero

FIGURA 6.6

Diagrama de canal para la función Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras



una vista del procedimiento de los requerimientos representa una sola dimensión del sistema. En la sección 6.4 se estudia el espacio de información y la forma en la que se representan los datos de requerimientos.

6.4 CONCEPTOS DE MODELADO DE DATOS

WebRef

En la dirección **www.datamodel.org**, hay información útil sobre el modelado de datos.

Si los requerimientos del software incluyen la necesidad de crear, ampliar o hacer interfaz con una base de datos, o si deben construirse y manipularse estructuras de datos complejas, el equipo del software tal vez elija crear un *modelo de datos* como parte del modelado general de los requerimientos. Un ingeniero o analista de software define todos los objetos de datos que se procesan dentro del sistema, la relación entre ellos y otro tipo de información que sea pertinente para las relaciones. El *diagrama entidad-relación* (DER) aborda dichos aspectos y representa todos los datos que se introducen, almacenan, transforman y generan dentro de una aplicación.

? ¿Cómo se manifiesta un objeto de datos en el contexto de una aplicación?

6.4.1 Objetos de datos

Un *objeto de datos* es una representación de información compuesta que debe ser entendida por el software. *Información compuesta* quiere decir algo que tiene varias propiedades o atributos

diferentes. Por tanto, el ancho (un solo valor) no sería un objeto de datos válido, pero las **dimensiones** (que incorporan altura, ancho y profundidad) sí podrían definirse como un objeto.

Un objeto de datos puede ser una entidad externa (por ejemplo, cualquier cosa que produzca o consuma información), una cosa (por ejemplo, un informe o pantalla), una ocurrencia (como una llamada telefónica) o evento (por ejemplo, una alarma), un rol (un vendedor), una unidad organizacional (por ejemplo, el departamento de contabilidad), un lugar (como una bodega) o estructura (como un archivo). Por ejemplo, una **persona** o un **auto** pueden considerarse como objetos de datos en tanto cada uno se define en términos de un conjunto de atributos. La descripción del objeto de datos incorpora a ésta todos sus atributos.

PUNTO CLAVE

Un objeto de datos es una representación de cualquier información compuesta que se procese en el software.

Un objeto de datos contiene sólo datos —dentro de él no hay referencia a las operaciones que se apliquen sobre los datos—.[10] Entonces, el objeto de datos puede representarse en forma de tabla, como la que se muestra en la figura 6.7. Los encabezados de la tabla reflejan atributos del objeto. En este caso, un auto se define en términos de **fabricante, modelo, número de serie, tipo de carrocería, color** y **propietario**. El cuerpo de la tabla representa instancias específicas del objeto de datos. Por ejemplo, un Chevy Corvette es una instancia del objeto de datos **auto**.

6.4.2 Atributos de los datos

PUNTO CLAVE

Los atributos nombran a un objeto de datos, describen sus características y, en ciertos casos, hacen referencia a otro objeto.

Los *atributos de los datos* definen las propiedades de un objeto de datos y tienen una de tres diferentes características. Se usan para 1) nombrar una instancia del objeto de datos, 2) describir la instancia o 3) hacer referencia a otra instancia en otra tabla. Además, debe definirse como identificador uno o más de los atributos —es decir, el atributo identificador se convierte en una "llave" cuando se desea encontrar una instancia del objeto de datos—. En ciertos casos, los valores para el (los) identificador(es) son únicos, aunque esto no es una exigencia. En relación con el objeto de datos **auto**, un identificador razonable sería el **número de serie**.

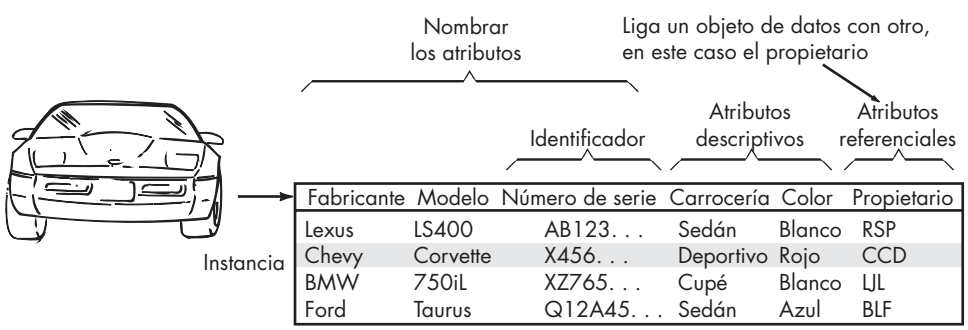
WebRef

Para aquellos que intentan hacer modelado de datos, es importante un concepto llamado "normalización". En la dirección **www.datamodel.org** se encuentra una introducción útil.

El conjunto de atributos que es apropiado para un objeto de datos determinado se define entendiendo el contexto del problema. Los atributos para **auto** podrían servir bien para una aplicación que usara un departamento de vehículos motorizados, pero serían inútiles para una compañía automotriz que necesitara hacer software de control de manufactura. En este último caso, los atributos para **auto** quizá también incluyan **número de serie, tipo de carrocería** y **color**, pero tendrían que agregarse muchos otros (por ejemplo, **código interior, tipo de tracción, indicador de paquete de recorte, tipo de transmisión**, etc.) para hacer de **auto** un objeto significativo en el contexto de control de manufactura.

FIGURA 6.7

Representación tabular de objetos de datos



10 Esta distinción separa al objeto de datos de la clase u objeto definidos como parte del enfoque orientado a objetos (véase el apéndice 2).

INFORMACIÓN

Objetos de datos y clases orientadas a objetos: ¿son lo mismo?

Al analizar objetos de datos es común que surja una pregunta: ¿un objeto de datos es lo mismo que una clase orientada[11] a objetos? La respuesta es "no".

Un objeto de datos define un aspecto de datos compuestos; es decir, incorpora un conjunto de características de datos individuales (atributos) y da al conjunto un nombre (el del objeto de datos).

Una clase orientada a objetos encierra atributos de datos, pero también incorpora las operaciones (métodos) que los manipulan y que están determinadas por dichos atributos. Además, la definición de clases implica una infraestructura amplia que es parte del enfoque de la ingeniería de software orientada a objetos. Las clases se comunican entre sí por medio de mensajes, se organizan en jerarquías y tienen características hereditarias para los objetos que son una instancia de una clase.

6.4.3 Relaciones

PUNTO CLAVE

Las relaciones indican la manera en la que los objetos de datos se conectan entre sí.

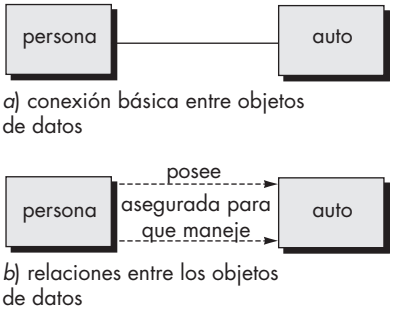
Los objetos de datos están conectados entre sí de diferentes maneras. Considere dos objetos de datos, **persona** y **auto**. Estos objetos se representan con la notación simple que se ilustra en la figura 6.8*a*). Se establece una conexión entre **persona** y **auto** porque ambos objetos están relacionados. Pero, ¿cuál es esa relación? Para determinarlo, debe entenderse el papel de las personas (propiedad, en este caso) y los autos dentro del contexto del software que se va a elaborar. Se establece un conjunto de parejas objeto/relación que definan las relaciones relevantes. Por ejemplo,

- Una persona *posee* un auto.
- Una persona *es asegurada para que maneje* un auto.

Las relaciones *posee* y *es asegurada para que maneje* definen las conexiones relevantes entre **persona** y **auto**. La figura 6.8*b*) ilustra estas parejas objeto-relación. Las flechas en esa figura dan información importante sobre la dirección de la relación y es frecuente que reduzcan las ambigüedades o interpretaciones erróneas.

FIGURA 6.8

Relaciones entre objetos de datos



11 Los lectores que no estén familiarizados con los conceptos y terminología de la orientación a objetos deben consultar el breve instructivo que se presenta en el apéndice 2.

INFORMACIÓN

Diagramas entidad-relación

La pareja objeto-relación es la piedra angular del modelo de datos. Estas parejas se representan gráficamente con el uso del diagrama entidad-relación (DER).[12] El DER fue propuesto por primera vez por Peter Chen [Che77] para diseñar sistemas de bases de datos relacionales y ha sido ampliado por otras personas. Se identifica un conjunto de componentes primarios para el DER: objetos de datos, atributos, relaciones y distintos indicadores de tipo. El propósito principal del DER es representar objetos de datos y sus relaciones.

Ya se presentó la notación DER básica. Los objetos de datos se representan con un rectángulo etiquetado. Las relaciones se indican con una línea etiquetada que conecta objetos. En ciertas variantes del DER, la línea de conexión contiene un rombo con la leyenda de la relación. Las conexiones entre los objetos de datos y las relaciones se establecen con el empleo de varios símbolos especiales que indican cardinalidad y modalidad.[13] Si el lector está interesado en obtener más información sobre el modelado de datos y el diagrama entidad-relación, consulte [Hob06] o [Sim05].

HERRAMIENTAS DE SOFTWARE

Modelado de datos

Objetivo: Las herramientas de modelado de datos dan a un ingeniero de software la capacidad de representar objetos de datos, sus características y relaciones. Se usan sobre todo para aplicaciones de grandes bases de datos y otros proyectos de sistemas de información, y proveen medios automatizados para crear diagramas completos de entidad-relación, diccionarios de objetos de datos y modelos relacionados.

Mecánica: Las herramientas de esta categoría permiten que el usuario describa objetos de datos y sus relaciones. En ciertos casos, utilizan notación DER. En otros, modelan relaciones con el empleo de un mecanismo diferente. Es frecuente que las herramientas en esta categoría se usen como parte del diseño de una base de datos y que permitan la creación de su modelo con la generación de un esquema para sistemas comunes de administración de bases de datos comunes (DBMS).

Herramientas representativas:[14]

AllFusion ERWin, desarrollada por Computer Associates (**www3.ca.com**), ayuda en el diseño de objetos de datos, estructura apropiada y elementos clave para las bases de datos.

ER/Studio, desarrollada por Embarcadero Software (**www.embarcadero.com**), da apoyo al modelado entidad-relación.

Oracle Designer, desarrollada por Oracle Systems (**www.oracle.com**), "modela procesos de negocios, entidades y relaciones de datos [que] se transforman en diseños para los que se generan aplicaciones y bases de datos completas".

Visible Analyst, desarrollada por Visible Systems (**www.visible.com**), da apoyo a varias funciones de modelado del análisis, incluso modelado de datos.

6.5 MODELADO BASADO EN CLASES

El modelado basado en clases representa los objetos que manipulará el sistema, las operaciones (también llamadas *métodos* o *servicios*) que se aplicarán a los objetos para efectuar la manipulación, las relaciones (algunas de ellas jerárquicas) entre los objetos y las colaboraciones que tienen lugar entre las clases definidas. Los elementos de un modelo basado en clases incluyen las clases y los objetos, atributos, operaciones, modelos clase-responsabilidad-colaborador (CRC), diagramas de colaboración y paquetes. En las secciones siguientes se presenta una serie de lineamientos informales que ayudarán a su identificación y representación.

---

12 Aunque algunas aplicaciones de diseño de bases de datos aún emplean el DER, ahora se utiliza la notación UML (véase el apéndice 1) para el diseño de datos.

13 La *cardinalidad* de una pareja objeto-relación especifica "el número de ocurrencias de uno [objeto] que se relaciona con el número de ocurrencias de otro [objeto]" [Til93]. La *modalidad* de una relación es 0 si no hay necesidad explícita para que ocurra la relación o si ésta es opcional. La modalidad es 1 si una ocurrencia de la relación es obligatoria.

14 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

6.5.1 Identificación de las clases de análisis

Cita:

"El problema realmente difícil es descubrir en primer lugar cuáles son los objetos correctos [clases]."

Carl Argila

Al mirar una habitación, se observa un conjunto de objetos físicos que se identifican, clasifican y definen fácilmente (en términos de atributos y operaciones). Pero cuando se "ve" el espacio del problema de una aplicación de software, las clases (y objetos) son más difíciles de concebir.

Se comienza por identificar las clases mediante el análisis de los escenarios de uso desarrollados como parte del modelo de requerimientos y la ejecución de un "análisis gramatical" [Abb83] sobre los casos de uso desarrollados para el sistema que se va a construir. Las clases se determinan subrayando cada sustantivo o frase que las incluya para introducirlo en una tabla simple. Deben anotarse los sinónimos. Si la clase (sustantivo) se requiere para implementar una solución, entonces forma parte del espacio de solución; de otro modo, si sólo es necesaria para describir la solución, es parte del espacio del problema.

Pero, ¿qué debe buscarse una vez identificados todos los sustantivos? Las *clases de análisis* se manifiestan en uno de los modos siguientes:

? ¿Cómo se manifiestan las clases en tantos elementos del espacio de solución?

- *Entidades externas* (por ejemplo, otros sistemas, dispositivos y personas) que producen o consumen la información que usará un sistema basado en computadora.
- *Cosas* (reportes, pantallas, cartas, señales, etc.) que forman parte del dominio de información para el problema.
- *Ocurrencias o eventos* (como una transferencia de propiedad o la ejecución de una serie de movimientos de un robot) que suceden dentro del contexto de la operación del sistema.
- *Roles* (gerente, ingeniero, vendedor, etc.) que desempeñan las personas que interactúan con el sistema.
- *Unidades organizacionales* (división, grupo, equipo, etc.) que son relevantes para una aplicación.
- *Lugares* (piso de manufactura o plataforma de carga) que establecen el contexto del problema y la función general del sistema.
- *Estructuras* (sensores, vehículos de cuatro ruedas, computadoras, etc.) que definen una clase de objetos o clases relacionadas de éstos.

Esta clasificación sólo es una de muchas propuestas en la bibliografía.[15] Por ejemplo, Budd [Bud96] sugiere una taxonomía de clases que incluye *productores* (fuentes) y *consumidores* (sumideros) de datos, *administradores de datos*, *vista*, *clases de observador* y *clases de auxiliares*.

También es importante darse cuenta de lo que no son las clases u objetos. En general, una clase nunca debe tener un "nombre de procedimiento imperativo" [Cas89]. Por ejemplo, si los desarrolladores del software de un sistema de imágenes médicas definieron un objeto con el nombre **InvertirImagen** o incluso **InversióndeImagen**, cometerían un error sutil. La **Imagen** obtenida del software podría ser, por supuesto, una clase (algo que es parte del dominio de la información). La inversión de la imagen es una operación que se aplica al objeto. Es probable que la inversión esté definida como una operación para el objeto **Imagen**, pero no lo estaría como clase separada con la connotación "inversión de imagen". Como afirma Cashman [Cas89]: "el intento de la orientación a objetos es contener, pero mantener separados, los datos y las operaciones sobre ellos".

Para ilustrar cómo podrían definirse las clases del análisis durante las primeras etapas del modelado, considere un análisis gramatical (los sustantivos están subrayados, los verbos apa-

15 En la sección 6.5.4 se estudia otra clasificación importante que define las clases *entidad*, *frontera* y *controladora*.

recen en cursivas) de una narración de procesamiento[16] para la función de seguridad de *CasaSegura*.

> La función de seguridad CasaSegura *permite* que el propietario *configure* el sistema de seguridad cuando se *instala*, *vigila* todos los sensores *conectados* al sistema de seguridad e *interactúa* con el propietario a través de internet, una PC o panel de control.
>
> Durante la instalación, la PC de CasaSegura se utiliza para *programar* y *configurar* el sistema. Se asigna a cada sensor un número y tipo, se programa un password maestro para *activar* y *desactivar* el sistema y se *introducen* números telefónicos para *marcar* cuando ocurre un evento de sensor.
>
> Cuando se *reconoce* un evento de sensor, el software *invoca* una alarma audible instalada en el sistema. Después de un tiempo de retraso que *especifica* el propietario durante las actividades de configuración del sistema, el software marca un número telefónico de un servicio de monitoreo, *proporciona* información acerca de la ubicación y *reporta* la naturaleza del evento detectado. El número telefónico se *vuelve a marcar* cada 20 segundos hasta que se *obtiene* la conexión telefónica.
>
> El propietario *recibe* información de seguridad a través de un panel de control, la PC o un navegador, lo que en conjunto se llama interfaz. La interfaz *despliega* mensajes de aviso e información del estado del sistema en el panel de control, la PC o la ventana del navegador. La interacción del propietario tiene la siguiente forma…

CONSEJO

El análisis gramatical no es a prueba de todo, pero da un impulso excelente para arrancar si se tienen dificultades para definir objetos de datos y las transformaciones que operan sobre ellos.

Con los sustantivos se proponen varias clases potenciales:

Clase potencial	Clasificación general
propietario	rol de entidad externa
sensor	entidad externa
panel de control	entidad externa
instalación	ocurrencia
sistema (alias sistema de seguridad)	cosa
número, tipo	no objetos, atributos de sensor
password maestro	cosa
número telefónico	cosa
evento de sensor	ocurrencia
alarma audible	entidad externa
servicio de monitoreo	unidad organizacional o entidad externa

La lista continuará hasta que se hayan considerado todos los sustantivos en la narrativa de procesamiento. Observe que cada entrada en la lista se llama objeto potencial. El lector debe considerar cada una antes de tomar la decisión final.

Coad y Yourdon [Coa91] sugieren seis características de selección que deben usarse cuando se considere cada clase potencial para incluirla en el modelo de análisis:

? ¿Cómo determino si una clase potencial debe, en realidad, ser una clase de análisis?

1. *Información retenida.* La clase potencial será útil durante el análisis sólo si debe recordarse la información sobre ella para que el sistema pueda funcionar.
2. *Servicios necesarios*. La clase potencial debe tener un conjunto de operaciones identificables que cambien en cierta manera el valor de sus atributos.

16 Una narración de procesamiento es similar al caso de uso en su estilo, pero algo distinto en su propósito. La narración de procesamiento hace una descripción general de la función que se va a desarrollar. No es un escenario escrito desde el punto de vista de un actor. No obstante, es importante observar que el análisis gramatical también puede emplearse para todo caso de uso desarrollado como parte de la obtención de requerimientos (indagación).

3. *Atributos múltiples.* Durante el análisis de los requerimientos, la atención debe estar en la información "principal"; en realidad, una clase con un solo atributo puede ser útil durante el diseño, pero es probable que durante la actividad de análisis se represente mejor como un atributo de otra clase.
4. *Atributos comunes*. Para la clase potencial se define un conjunto de atributos y se aplican éstos a todas las instancias de la clase.
5. *Operaciones comunes*. Se define un conjunto de operaciones para la clase potencial y éstas se aplican a todas las instancias de la clase.
6. *Requerimientos esenciales.* Las entidades externas que aparezcan en el espacio del problema y que produzcan o consuman información esencial para la operación de cualquier solución para el sistema casi siempre se definirán como clases en el modelo de requerimientos.

Cita:

"Las clases luchan, algunas triunfan, otras son eliminadas."

Mao Tse Tung

Para que se considere una clase legítima para su inclusión en el modelo de requerimientos, un objeto potencial debe satisfacer todas (o casi todas) las características anteriores. La decisión de incluir clases potenciales en el modelo de análisis es algo subjetiva, y una evaluación posterior tal vez haga que un objeto se descarte o se incluya de nuevo. Sin embargo, el primer paso del modelado basado en clases es la definición de éstas, y deben tomarse las medidas respectivas (aun las subjetivas). Con esto en mente, se aplicarán las características de selección a la lista de clases potenciales de *CasaSegura*:

Clase potencial	Número de característica que se aplica
propietario	rechazada: 1 y 2 fallan, aunque la 6 aplica
sensor	aceptada: se aplican todas
panel de control	aceptada: se aplican todas
instalación	rechazada
sistema (alias sistema de seguridad)	aceptada: se aplican todas
número, tipo	rechazada: 3 fallan, atributos de sensores
password maestro	rechazada: 3 falla
número telefónico	rechazada: 3 falla
evento de sensor	aceptada: se aplican todas
alarma audible	aceptada: se aplican 2, 3, 4, 5 y 6
servicio de monitoreo	rechazada: 1 y 2 fallan aunque la 6 aplica

Debe notarse que: 1) la lista anterior no es exhaustiva; para completar el modelo tendrían que agregarse clases adicionales; 2) algunas de las clases potenciales rechazadas se convertirán en atributos para otras que sí fueron aceptadas (por ejemplo, **número** y **tipo** son atributos de **Sensor**, y **password maestro** y **número telefónico** pueden convertirse en atributos de **Sistema**); 3) diferentes enunciados del problema harían que se tomaran decisiones distintas para "aceptar o rechazar" (por ejemplo, si cada propietario tuviera una clave individual o se identificara con reconocimiento de voz, la clase **Propietario** satisfaría las características 1 y 2, y se aceptaría).

6.5.2 Especificación de atributos

PUNTO CLAVE

Los atributos son el conjunto de objetos de datos que definen por completo la clase en el contexto del problema.

Los *atributos* describen una clase que ha sido seleccionada para incluirse en el modelo de requerimientos. En esencia, son los atributos los que definen la clase (esto aclara lo que significa la clase en el contexto del espacio del problema). Por ejemplo, si se fuera a construir un sistema que analiza estadísticas de jugadores de béisbol profesionales, los atributos de la clase **Jugador** serían muy distintos de los que tendría la misma clase cuando se usara en el contexto del sis-

tema de pensiones de dicho deporte. En la primera, atributos tales como **nombre, porcentaje de bateo, porcentaje de fildeo, años jugados** y **juegos jugados** serían relevantes. Para la segunda, algunos de los anteriores sí serían significativos, pero otros se sustituirían (o se crearían) por atributos tales como **salario promedio, crédito hacia el retiro completo, opciones del plan de pensiones elegido, dirección de correo**, etcétera.

Para desarrollar un conjunto de atributos significativos para una clase de análisis, debe estudiarse cada caso de uso y seleccionar aquellas "cosas" que "pertenezcan" razonablemente a la clase. Además, debe responderse la pregunta siguiente para cada clase: "¿qué aspectos de los datos (compuestos o elementales) definen por completo esta clase en el contexto del problema en cuestión?"

Para ilustrarlo, se considera la clase **Sistema** definida para *CasaSegura*. El propietario configura la función de seguridad para que refleje la información de los sensores, la respuesta de la alarma, la activación o desactivación, la identificación, etc. Estos datos compuestos se representan del modo siguiente:

información de identificación = identificación del sistema + número telefónico de verificación + estado del sistema
información de respuesta de la alarma = tiempo de retraso + número telefónico
información de activación o desactivación = password maestro + número de intentos permisibles + password temporal

Cada uno de los datos a la derecha del signo igual podría definirse más, hasta un nivel elemental, pero para nuestros propósitos constituye una lista razonable de atributos para la clase **Sistema** (parte sombreada de la figura 6.9).

Los sensores forman parte del sistema general *CasaSegura*, pero no están enlistados como datos o atributos en la figura 6.9. **Sensor** ya se definió como clase, y se asociarán múltiples objetos **Sensor** con la clase **Sistema**. En general, se evita definir algo como atributo si más de uno va a asociarse con la clase.

6.5.3 Definición de las operaciones

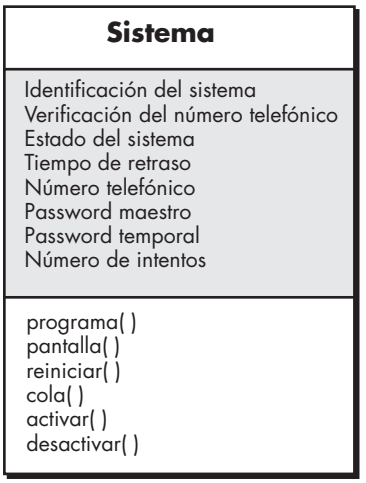
Cuando se definen operaciones para una clase de análisis, hay que centrarse en el comportamiento orientado al problema y no en los comportamientos requeridos para su implementación.

Las *operaciones* definen el comportamiento de un objeto. Aunque existen muchos tipos distintos de operaciones, por lo general se dividen en cuatro categorías principales: 1) operaciones que manipulan datos en cierta manera (por ejemplo, los agregan, eliminan, editan, seleccionan, etc.), 2) operaciones que realizan un cálculo, 3) operaciones que preguntan sobre el estado de un objeto y 4) operaciones que vigilan un objeto en cuanto a la ocurrencia de un evento de control. Estas funciones se llevan a cabo con operaciones sobre los atributos o sobre asociaciones de éstos (véase la sección 6.5.5). Por tanto, una operación debe tener "conocimiento" de la naturaleza de los atributos y de las asociaciones de la clase.

Como primera iteración al obtener un conjunto de operaciones para una clase de análisis, se estudia otra vez una narración del procesamiento (o caso de uso) y se eligen aquellas que pertenezcan razonablemente a la clase. Para lograr esto, de nuevo se efectúa el análisis gramatical y se aíslan los verbos. Algunos de éstos serán operaciones legítimas y se conectarán con facilidad a una clase específica. Por ejemplo, de la narración del procesamiento de *CasaSegura* ya presentada en este capítulo, se observa que "se *asigna* a sensor un número y tipo" o "se *programa* un password maestro para *activar y desactivar* el sistema" indican cierto número de cosas:

- Que una operación *asignar( )* es relevante para la clase **Sensor**.
- Que se aplicará una operación *programar( )* a la clase **Sistema**.
- Que *activar( )* y *desactivar( )* son operaciones que se aplican a la clase **Sistema**.

FIGURA 6.9
Diagrama de clase para la clase sistema



Hasta no hacer más investigaciones, es probable que la operación *programar( )* se divida en cierto número de suboperaciones específicas adicionales que se requieren para configurar el sistema. Por ejemplo, *programar( )* implica la especificación de números telefónicos, la configuración de las características del sistema (por ejemplo, crear la tabla de sensores, introducir las características de la alarma, etc.) y la introducción de la(s) clave(s). Pero, de momento, *programar( )* se especifica como una sola operación.

Además del análisis gramatical, se obtiene más perspectiva sobre otras operaciones si se considera la comunicación que ocurre entre los objetos. Éstos se comunican con la transmisión de mensajes entre sí. Antes de continuar con la especificación de operaciones, se estudiará esto con más detalle.

CASASEGURA

Modelos de clase

La escena: Cubículo de Ed, cuando comienza el modelado de los requerimientos.

Participantes: Jamie, Vinod y Ed, todos ellos miembros del equipo de ingeniería de software para *CasaSegura*.

La conversación:

[Ed ha estado trabajando para obtener las clases a partir del formato del caso de uso para AVC-MVC (presentado en un recuadro anterior de este capítulo) y expone a sus colegas las que ha obtenido].

Ed: Entonces, cuando el propietario quiere escoger una cámara, la tiene que elegir del plano. Definí una clase llamada **Plano**. Éste es el diagrama.
(Observan la figura 6.10.)

Jamie: Entonces, **Plano** es un objeto que agrupa paredes, puertas, ventanas y cámaras. Eso significa esas líneas con leyendas, ¿verdad?

Ed: Sí, se llaman "asociaciones". Una clase se asocia con otra de acuerdo con las asociaciones que se ven (las asociaciones se estudian en la sección 6.5.5).

Vinod: Es decir, el plano real está constituido por paredes que contienen en su interior cámaras y sensores. ¿Cómo sabe el plano dónde colocar estos objetos?

Ed: No lo sabe, pero las otras clases sí. Mira los atributos de, digamos, **SegmentodePared**, que se usa para construir una pared. El segmento de muro tiene coordenadas de inicio y final, y la operación *draw( )* hace el resto.

Jamie: Y lo mismo vale para las ventanas y puertas. Parece como si cámara tuviera algunos atributos adicionales.

Ed: Sí. Los necesito para dar información del alcance y el acercamiento.

Vinod: Tengo una pregunta. ¿Por qué tiene la cámara una identificación pero las demás no? Veo que tienes un atributo llamado **ParedSiguiente**. ¿Cómo sabe **SegmentodePared** cuál será la pared siguiente?

Ed: Buena pregunta, pero, como dijimos, ésa es una decisión de diseño, por lo que la voy a retrasar hasta...

Jamie: Momento... Apuesto a que ya lo has imaginado.

Ed (sonríe con timidez): Es cierto, voy a usar una estructura de lista que modelaré cuando vayamos a diseñar. Si somos puristas en cuanto a separar el análisis y el diseño, el nivel de detalle podría parecer sospechoso.

Jamie: Me parece muy bien, pero tengo más preguntas.

(Jamie hace preguntas que dan como resultado modificaciones menores.)

Vinod: ¿Tienes tarjetas CRC para cada uno de los objetos? Si así fuera, debemos actuar con ellas, sólo para estar seguros de que no hemos omitido nada.

Ed: No estoy seguro de cómo hacerlas.

Vinod: No es difícil y en verdad convienen. Te mostraré.

6.5.4 Modelado clase-responsabilidad-colaborador (CRC)

Cita:

"Un propósito de las tarjetas CRC es que fallen pronto, con frecuencia y en forma barata. Es mucho más barato desechar tarjetas que reorganizar una gran cantidad de código fuente."

C. Horstman

El modelado clase-responsabilidad-colaborador (CRC) [Wir90] proporciona una manera sencilla de identificación y organización de las clases que son relevantes para los requerimientos de un sistema o producto. Ambler [Amb95] describe el modelado CRC en la siguiente forma:

> Un modelo CRC en realidad es un conjunto de tarjetas índice estándar que representan clases. Las tarjetas se dividen en tres secciones. En la parte superior de la tarjeta se escribe el nombre de la clase, en la parte izquierda del cuerpo se enlistan las responsabilidades de la clase y en la derecha, los colaboradores.

En realidad, el modelo CRC hace uso de tarjetas índice reales o virtuales. El objetivo es desarrollar una representación organizada de las clases. Las *responsabilidades* son los atributos y ope-

FIGURA 6.10

Diagrama de clase para Plano (véase el análisis en el recuadro)

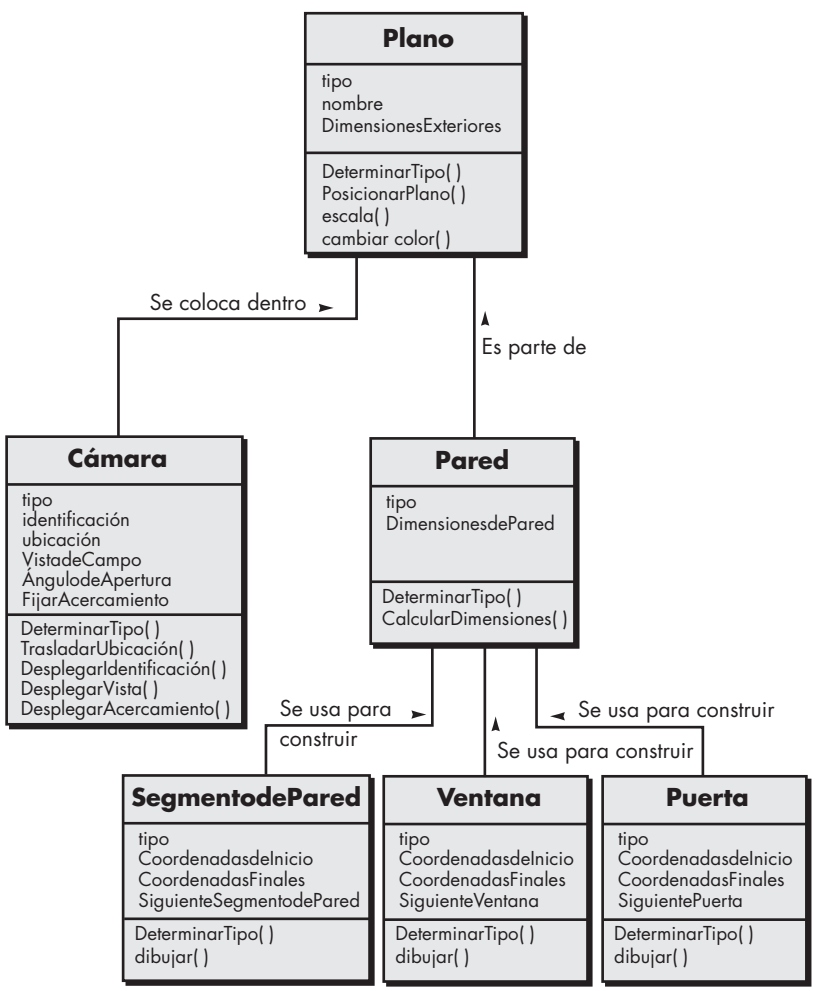
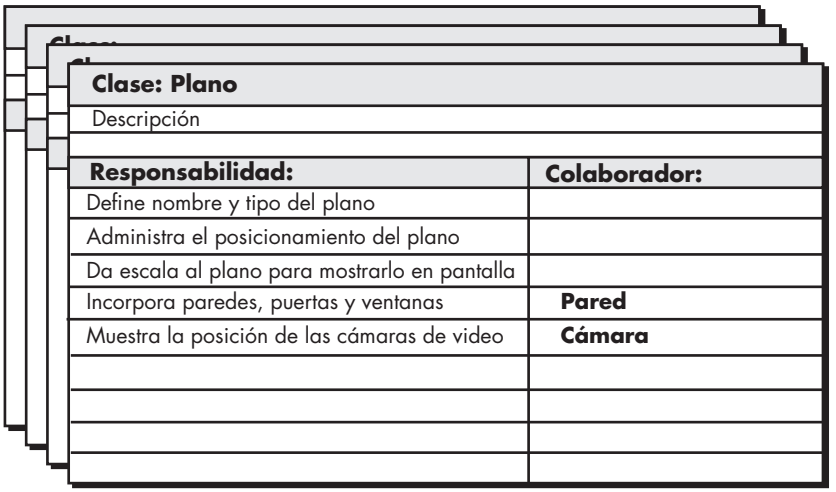


FIGURA 6.11
Modelo de tarjeta CRC índice



raciones relevantes para la clase. En pocas palabras, una responsabilidad es "cualquier cosa que la clase sepa o haga" [Amb95]. Los *colaboradores* son aquellas clases que se requieren para dar a una clase la información necesaria a fin de completar una responsabilidad. En general, una *colaboración* implica una solicitud de información o de cierta acción.

En la figura 6.11 se ilustra una tarjeta CRC índice sencilla para la clase **Plano**: la lista de responsabilidades en la tarjeta CRC es preliminar y está sujeta a agregados o modificaciones. Las clases **Pared** y **Cámara** se anotan frente a la responsabilidad que requerirá su colaboración.

WebRef
En la dirección **www.theumlcafe.com/a0079.htm** hay un análisis excelente de estos tipos de clase.

Clases. Al inicio de este capítulo se presentaron lineamientos básicos para identificar clases y objetos. La taxonomía de tipos de clase presentada en la sección 6.5.1 puede ampliarse con las siguientes categorías:

- *Clases de entidad*, también llamadas clases *modelo* o *de negocio*, se extraen directamente del enunciado del problema (por ejemplo, **Plano** y **Sensor**). Es común que estas clases representen cosas almacenadas en una base de datos y que persistan mientras dure la aplicación (a menos que se eliminen en forma específica).
- *Clases de frontera* se utilizan para crear la interfaz (por ejemplo, pantallas interactivas o reportes impresos) que el usuario mira y con la que interactúa cuando utiliza el software. Los objetos de entidad contienen información que es importante para los usuarios, pero no se muestran por sí mismos. Las clases de frontera se diseñan con la responsabilidad de administrar la forma en la que se presentan a los usuarios los objetos de entidad. Por ejemplo, una clase de frontera llamada **VentanadeCámara** tendría la responsabilidad de desplegar la salida de una cámara de vigilancia para el sistema *CasaSegura*.
- *Clases de controlador* administran una "unidad de trabajo" [UML03] de principio a fin. Es decir, las clases de controlador están diseñadas para administrar 1) la creación o actualización de objetos de entidad, 2) las instancias de los objetos de frontera en tanto obtienen información de los objetos de entidad, 3) la comunicación compleja entre conjuntos de objetos y 4) la validación de datos comunicados entre objetos o entre el usuario y la aplicación. En general, las clases de controlador no se consideran hasta haber comenzado la actividad de diseño.

Cita:
"Pueden clasificarse científicamente los objetos en tres grandes categorías: los que no funcionan, los que se descomponen y los que se pierden."
Rusell Baker

? ¿Qué lineamientos se aplican para asignar responsabilidades a las clases?

Responsabilidades. En las secciones 6.5.2 y 6.5.3 se presentaron los lineamientos básicos para identificar responsabilidades (atributos y operaciones). Wirfs-Brock *et al.* [Wir90] sugieren cinco lineamientos para asignar responsabilidades a las clases:

1. **La inteligencia del sistema debe estar distribuida entre las clases para enfrentar mejor las necesidades del problema.** Toda aplicación contiene cierto grado de inteligencia, es decir, lo que el sistema sabe y lo que puede hacer. Esta inteligencia se distribuye entre las clases de diferentes maneras. Las clases "tontas" (aquellas que tienen pocas responsabilidades) pueden modelarse para que actúen como subordinadas de ciertas clases "inteligentes" (las que tienen muchas responsabilidades). Aunque este enfoque hace directo el flujo del control en un sistema, tiene algunas desventajas: concentra toda la inteligencia en pocas clases, lo que hace que sea más difícil hacer cambios, y tiende a que se requieran más clases y por ello más trabajo de desarrollo.

   Si la inteligencia del sistema tiene una distribución más pareja entre las clases de una aplicación, cada objeto sabe algo, sólo hace unas cuantas cosas (que por lo general están bien identificadas) y la cohesión del sistema mejora.[17] Esto facilita el mantenimiento del software y reduce el efecto de los resultados colaterales del cambio.

   Para determinar si la inteligencia del sistema está distribuida en forma apropiada, deben evaluarse las responsabilidades anotadas en cada modelo de tarjeta CRC índice a fin de definir si alguna clase tiene una lista demasiado larga de responsabilidades. Esto indica una concentración de inteligencia.[18] Además, las responsabilidades de cada clase deben tener el mismo nivel de abstracción. Por ejemplo, entre las operaciones enlistadas para una clase agregada llamada **RevisarCuenta**, un revisor anota dos responsabilidades: *hacer el balance de la cuenta* y *eliminar comprobaciones concluidas*. La primera operación (responsabilidad) implica un procedimiento matemático complejo y lógico. La segunda es una simple actividad de oficina. Como estas dos operaciones no están en el mismo nivel de abstracción, *eliminar comprobaciones concluidas* debe colocarse dentro de las responsabilidades de **RevisarEntrada**, clase que está incluida en la clase agregada **RevisarCuenta**.
2. **Cada responsabilidad debe enunciarse del modo más general posible.** Este lineamiento implica que las responsabilidades generales (tanto atributos como operaciones) deben residir en un nivel elevado de la jerarquía de clases (porque son generales y se aplicarán a todas las subclases).
3. **La información y el comportamiento relacionado con ella deben residir dentro de la misma clase.** Esto logra el principio orientado a objetos llamado *encapsulamiento*. Los datos y los procesos que los manipulan deben empacarse como una unidad cohesiva.
4. **La información sobre una cosa debe localizarse con una sola clase, y no distribuirse a través de muchas.** Una sola clase debe tener la responsabilidad de almacenar y manipular un tipo específico de información. En general, esta responsabilidad no debe ser compartida por varias clases. Si la información está distribuida, es más difícil dar mantenimiento al software y más complicado someterlo a prueba.
5. **Cuando sea apropiado, las responsabilidades deben compartirse entre clases relacionadas.** Hay muchos casos en los que varios objetos relacionados deben tener el mismo comportamiento al mismo tiempo. Por ejemplo, considere un juego de video que deba tener en la pantalla las clases siguientes: **Jugador**, **CuerpodelJugador**, **BrazosdelJugador**, **PiernasdelJugador** y **CabezadelJugador**. Cada una de estas clases tiene sus propios atributos (como **posición**, **orientación**, **color** y **velocidad**) y todas deben actualizarse y desplegarse a medida que el usuario manipula una palanca de juego. Las res-

17 La cohesión es un concepto de diseño que se estudia en el capítulo 8.

18 En tales casos, puede ser necesario dividir la clase en una multiplicidad de ellas o completar subsistemas con el objeto de distribuir la inteligencia de un modo más eficaz.

ponsabilidades *actualizar( )* y *desplegar( )* deben, por tanto, ser compartidas por cada uno de los objetos mencionados. **Jugador** sabe cuando algo ha cambiado y requiere *actualizarse( )*. Colabora con los demás objetos para obtener una nueva posición u orientación, pero cada objeto controla su propio despliegue en la pantalla.

Colaboraciones. Una clase cumple sus responsabilidades en una de dos formas: 1) usa sus propias operaciones para manipular sus propios atributos, con lo que satisface una responsabilidad particular o 2) colabora con otras clases. Wirfs-Brock *et al.* [Wir90] definen las colaboraciones del modo siguiente:

> Las colaboraciones representan solicitudes que hace un cliente a un servidor para cumplir con sus responsabilidades. Una colaboración es la materialización del contrato entre el cliente y el servidor [...] Decimos que un objeto colabora con otro si, para cumplir una responsabilidad, necesita enviar al otro objeto cualesquiera mensajes. Una sola colaboración fluye en una dirección: representa una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor.

Las colaboraciones se identifican determinando si una clase puede cumplir cada responsabilidad. Si no es así, entonces necesita interactuar con otra clase. Ésa es una colaboración.

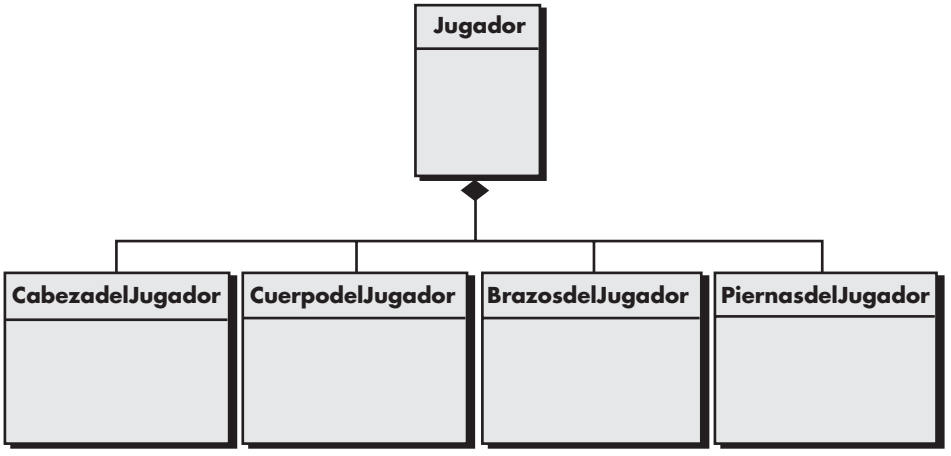
Como ejemplo, considere la función de seguridad de *CasaSegura*. Como parte del procedimiento de activación, el objeto **PaneldeControl** debe determinar si están abiertos algunos sensores. Se define una responsabilidad llamada *determinar-estado-delsensor( )*. Si los sensores están abiertos, **PaneldeControl** debe fijar el atributo **estado** como "no está listo". La información del sensor se adquiere de cada objeto **Sensor**. Por tanto, la responsabilidad *determinar-estado-delsensor( )* se cumple sólo si **PaneldeControl** trabaja en colaboración con **Sensor**.

Para ayudar a identificar a los colaboradores, se estudian tres relaciones generales diferentes entre las clases [Wir90]: 1) la relación *es-parte-de*, 2) la relación *tiene-conocimiento-de* y 3) la relación *depende-de*. En los párrafos siguientes se analizan brevemente cada una de estas tres responsabilidades generales.

Todas las clases que forman parte de una clase agregada se conectan a ésta por medio de una relación *es-parte-de*. Considere las clases definidas por el juego mencionado antes, la clase **CuerpodelJugador** *es-parte-de* **Jugador**, igual que **BrazosdelJugador, PiernasdelJugador** y **CabezadelJugador**. En UML, estas relaciones se representan como el agregado que se ilustra en la figura 6.12.

Cuando una clase debe adquirir información de otra, se establece la relación *tiene-conocimiento-de*. La responsabilidad *determinar-estado-delsensor( )* ya mencionada es un ejemplo de ello.

FIGURA 6.12
Una clase agregada compuesta



La relación *depende-de* significa que dos clases tienen una dependencia que no se determina por *tiene-conocimiento-de* ni por *es-parte-de*. Por ejemplo, **CabezadelJugador** siempre debe estar conectada a **CuerpodelJugador** (a menos que el juego de video sea particularmente violento), pero cada objeto puede existir sin el conocimiento directo del otro. Un atributo del objeto **CabezadelJugador**, llamado **posición-central**, se determina a partir de la posición central de **CuerpodelJugador**. Esta información se obtiene por medio de un tercer objeto, **Jugador**, que la obtiene de **CuerpodelJugador**. Entonces, **CabezadelJugador** *depende-de* **CuerpodelJugador**.

En todos los casos, el nombre de la clase colaboradora se registra en el modelo de tarjeta CRC índice, junto a la responsabilidad que produce la colaboración. Por tanto, la tarjeta índice contiene una lista de responsabilidades y las colaboraciones correspondientes que hacen que se cumplan (véase la figura 6.11).

Cuando se ha desarrollado un modelo CRC completo, los participantes lo revisan con el empleo del enfoque siguiente [Amb95]:

1. Se da a todos los participantes que intervienen en la revisión (del modelo CRC) un subconjunto del modelo de tarjetas índice CRC. Deben separarse aquellas que colaboran (de modo que ningún revisor deba tener dos tarjetas que colaboren).
2. Todos los escenarios de casos de uso (y los diagramas correspondientes) deben organizarse en dos categorías.
3. El líder de la revisión lee el caso de uso en forma deliberada. Cuando llega a un objeto con nombre, entrega una ficha a la persona que tenga la tarjeta índice de la clase correspondiente. Por ejemplo, un caso de uso de *CasaSegura* contiene la narración siguiente:

   El propietario observa el panel de control de *CasaSegura* para determinar si el sistema está listo para recibir una entrada. Si el sistema no está listo, el propietario debe cerrar físicamente las puertas y ventanas de modo que el indicador *listo* aparezca [un indicador *no está listo* implica que un sensor se encuentra abierto, es decir, que una puerta o ventana está abierta].

   Cuando en la narración del caso de uso el líder de la revisión llega a "panel de control", entrega la ficha a la persona que tiene la tarjeta índice **PaneldeControl**. La frase "implica que un sensor está abierto" requiere que la tarjeta índice contenga una responsabilidad que validará esta implicación (esto lo logra la responsabilidad *determinar-estado-delsensor( )*. Junto a la responsabilidad, en la tarjeta índice se encuentra el **Sensor** colaborador. Entonces, la ficha pasa al objeto **Sensor**.
4. Cuando se pasa la ficha, se pide al poseedor de la tarjeta **Sensor** que describa las responsabilidades anotadas en la tarjeta. El grupo determina si una (o más) de las responsabilidades satisfacen el requerimiento del caso de uso.
5. Si las responsabilidades y colaboraciones anotadas en las tarjetas índice no se acomodan al caso de uso, éstas se modifican. Lo anterior tal vez incluya la definición de nuevas clases (y las tarjetas CRC índice correspondientes) o la especificación en las tarjetas existentes de responsabilidades o colaboraciones nuevas o revisadas.

Este modo de operar continúa hasta terminar el caso de uso. Cuando se han revisado todos los casos de uso, continúa el modelado de los requerimientos.

6.5.5 Asociaciones y dependencias

PUNTO CLAVE

Una asociación define una relación entre clases. La multiplicidad define cuántas de una clase se relacionan con cuántas de otra clase.

En muchos casos, dos clases de análisis se relacionan de cierto modo con otra, en forma muy parecida a como dos objetos de datos se relacionan entre sí (véase la sección 6.4.3). En UML, estas relaciones se llaman *asociaciones*. Al consultar la figura 6.10, la clase **Plano** se define con la identificación de un conjunto de asociaciones entre **Plano** y otras dos clases, **Cámara** y **Pa-**

CasaSegura

Modelos CRC

La escena: Cubículo de Ed cuando comienza el modelado de los requerimientos.

Participantes: Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

[Vinod ha decidido enseñar a Ed con un ejemplo cómo desarrollar las tarjetas CRC.]

Vinod: Mientras tú trabajabas en la vigilancia y Jamie lo hacía con la seguridad, yo estaba en la función de administración del hogar.

Ed: ¿Cuál es el estado de eso? Mercadotecnia cambia lo que quiere a cada rato.

Vinod: Aquí está la primera versión de caso de uso para toda la función... la mejoramos un poco, pero debe darte el panorama general...

Caso de uso: Función de administración de *CasaSegura*.

Narración: Queremos usar la interfaz de administración del hogar en una PC o en una conexión de internet para controlar los dispositivos electrónicos que tengan controladores de interfaz inalámbrica. El sistema debe permitir encender y apagar focos específicos, controlar aparatos conectados a una interfaz inalámbrica y fijar el sistema de calefacción y aire acondicionado a la temperatura que desee. Para hacer esto, quiero seleccionar los aparatos en el plano de la casa. Cada equipo debe estar identificado en el plano. Como característica opcional, quiero controlar todos los equipos audiovisuales: sonido, televisión, DVD, grabadoras digitales, etcétera.

Con una sola selección, quiero preparar toda la casa para distintas situaciones. Una es *casa*, otra es *salir*, la tercera es *viaje nocturno* y la cuarta es *viaje largo*. Todas estas situaciones tienen especificaciones que se aplicarán a todos los equipos. En los estados de *viaje nocturno* y *viaje largo*, el sistema debe encender y apagar focos en momentos elegidos al azar (para que parezca que hay alguien en casa) y controlar el sistema de calefacción y aire acondicionado. Debo poder hacer esta preparación por internet, con la protección de claves adecuadas...

Ed: ¿El personal de hardware ya tiene listas todas las interfaces inalámbricas?

Vinod (sonríe): Están trabajando en eso; dicen que no hay problema. De cualquier forma, obtuve muchas clases para la administración del hogar y podemos usar una como ejemplo. Tomemos la clase **InterfazdeAdministracióndelHogar**.

Ed: Bien... entonces, las responsabilidades son... los atributos y operaciones para la clase, y las colaboraciones son las clases que indican las responsabilidades.

Vinod: Pensé que no habías entendido el concepto CRC.

Ed: Un poco, quizá, pero continúa.

Vinod: Aquí está mi definición de la clase **InterfazdeAdministracióndelHogar**.

Atributos:

PaneldeOpciones: contiene información sobre los botones que permiten al usuario seleccionar funcionalidad.

PaneldeSituación: contiene información acerca de los botones que permiten que el usuario seleccione la situación.

Plano: igual que el objeto de vigilancia, pero éste muestra los equipos.

ÍconosdeAparatos: informa sobre los íconos que representan luces, aparatos, calefacción y aire acondicionado, etcétera.

PaneldeAparatos: simula el panel de control de un aparato o equipo; permite controlarlo.

Operaciones:

DesplegarControl( ), SeleccionarControl( ), DesplegarSituación( ), SeleccionarSituación( ), AccederaPlano( ), SeleccionarÍconodeEquipo( ), DesplegarPaneldeEquipo( ), AccederaPaneldeEquipo( ),...

Clase: InterfazdeAdministracióndelHogar

Responsabilidad	**Colaborador**
DesplegarControl( )	**PaneldeOpciones** (clase)
SeleccionarControl( )	**PaneldeOpciones** (clase)
DesplegarSituación( )	**PaneldeSituación** (clase)
SeleccionarSituación( )	**PaneldeSituación** (clase)
AccederaPlano( )	**Plano** (clase) . . .

. . .

Ed: De modo que cuando se invoque a operación *AccederaPlano( )*, colabora con el objeto **Plano** de igual manera que el que desarrollamos para vigilancia. Espera, aquí tengo su descripción (ven la figura 6.10).

Vinod: Exactamente. Y si quisiéramos revisar todo el modelo de la clase, podríamos comenzar con esta tarjeta índice, luego iríamos a la del colaborador y de ahí a una de los colaboradores del colaborador, y así sucesivamente.

Ed: Buena forma de encontrar omisiones o errores.

Vinod: Sí.

red. La clase **Pared** está asociada con tres clases que permiten que se construya ésta, y que son **SegmentodePared, Ventana** y **Puerta**.

En ciertos casos, una asociación puede definirse con más detalle si se indica *multiplicidad*. En relación con la figura 6.10, un objeto **Pared** se construye a partir de uno o más objetos **SegmentodePared**. Además, el objeto **Pared** puede contener 0 o más objetos **Ventana** y 0 o más objetos **Puerta**. Estas restricciones de multiplicidad se ilustran en la figura 6.13, donde "uno o

FIGURA 6.13
Multiplicidad

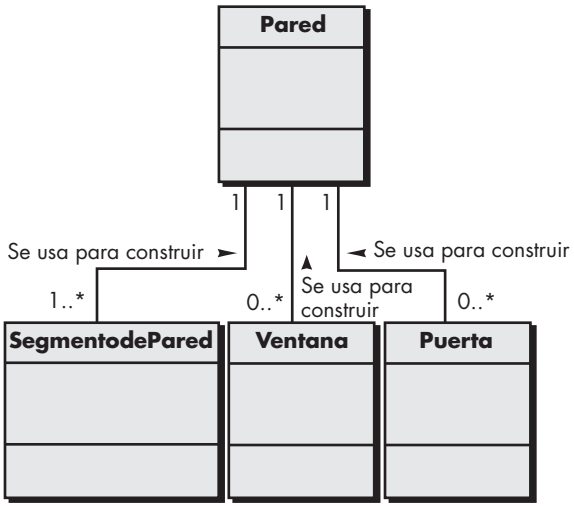
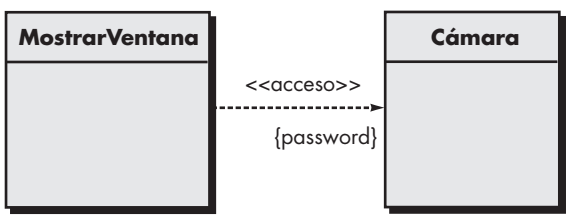


FIGURA 6.14
Dependencias



más" se representa con 1...*, y para "0 o más" se usa 0...*. En LMU, el asterisco indica una frontera ilimitada en ese rango.[19]

¿Qué es un estereotipo?

Sucede con frecuencia que entre dos clases de análisis existe una relación cliente-servidor. En tales casos, una clase cliente depende de algún modo de la clase servidor, y se establece una *relación de dependencia*. Las dependencias están definidas por un estereotipo. Un *estereotipo* es un "mecanismo extensible" [Arl02] dentro del UML que permite definir un elemento especial de modelado con semántica y especialización determinadas. En UML, los estereotipos se representan entre paréntesis dobles angulares (por ejemplo, <<estereotipo>>).

PUNTO CLAVE
Un paquete se utiliza para ensamblar un conjunto de clases relacionadas

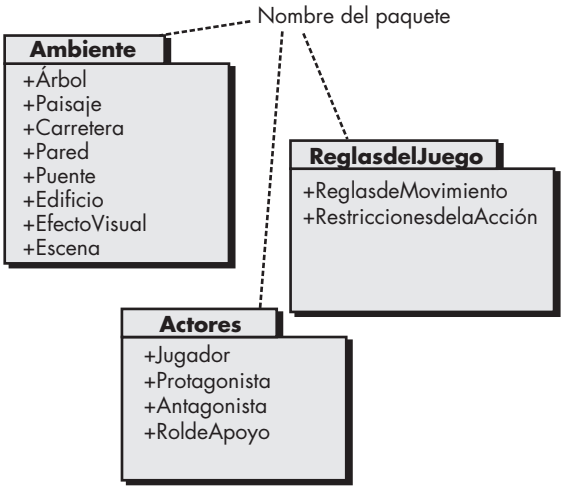
Como ilustración de una dependencia simple dentro del sistema de vigilancia *CasaSegura*, un objeto **Cámara** (la clase servidora, en este caso) proporciona una imagen a un objeto **MostrarVentana** (la clase cliente). La relación entre estos dos objetos no es una asociación simple sino de dependencia. En el caso de uso escrito para la vigilancia (que no se presenta aquí), debe darse una clave especial a fin de observar ubicaciones específicas de las cámaras. Una forma de lograr esto es hacer que **Cámara** pida un password y luego asegure el permiso a **MostrarVentana** para que presente el video. Esto se representa en la figura 6.14, donde <<acceso>> implica que el uso de la salida de cámara se controla con una clave especial.

6.5.6 Paquetes de análisis

Una parte importante del modelado del análisis es la categorización. Es decir, se clasifican distintos elementos del modelo de análisis (por ejemplo, casos de uso y clases de análisis) de ma-

19 Como parte de una asociación, pueden indicarse otras relaciones de multiplicidad: una a una, una a muchas, muchas a muchas, una a un rango específico con límites inferior y superior, y otras.

FIGURA 6.15
Paquetes



nera que se agrupen en un paquete —llamado *paquete de análisis*— al que se da un nombre representativo.

Para ilustrar el uso de los paquetes de análisis, considere el juego de video que se mencionó antes. A medida que se desarrolla el modelo de análisis para el juego de video, se obtiene un gran número de clases. Algunas se centran en el ambiente del juego —las escenas visuales que el usuario ve cuando lo usa—. En esta categoría quedan clases tales como **Árbol, Paisaje, Carretera, Pared, Puente, Edificio** y **EfectoVisual**. Otras se centran en los caracteres dentro del juego y describen sus características físicas, acciones y restricciones. Pueden definirse clases como **Jugador** (ya descrita), **Protagonista**, **Antagonista** y **RolesdeApoyo**. Otras más describen las reglas del juego —cómo se desplaza un jugador por el ambiente—. Candidatas para esto son clases como **ReglasdeMovimiento** y **RestriccionesdelaAcción**. Pueden existir muchas otras categorías. Estas clases se agrupan en los paquetes de análisis que se observan en la figura 6.15.

El signo *más* (suma) que precede al nombre de la clase de análisis en cada paquete, indica que las clases tienen visibilidad pública, por lo que son accesibles desde otros paquetes. Aunque no se aprecia en la figura, hay otros símbolos que preceden a un elemento dentro de un paquete. El signo *menos* (resta) indica que un elemento queda oculto desde todos los demás paquetes. Y el símbolo # señala que un elemento es accesible sólo para los paquetes contenidos dentro de un paquete dado.

6.6 RESUMEN

El objetivo del modelado de los requerimientos es crear varias representaciones que describan lo que necesita el cliente, establecer una base para generar un diseño de software y definir un conjunto de requerimientos que puedan ser validados una vez construido el software. El modelo de requerimientos cruza la brecha entre la representación del sistema que describe el sistema en su conjunto y la funcionalidad del negocio, y un diseño de software que describe la arquitectura de la aplicación del software, la interfaz de usuario y la estructura de componentes.

Los modelos basados en el escenario ilustran los requerimientos del software desde el punto de vista del usuario. El caso de uso —descripción, hecha con una narración o un formato, de una interacción entre un actor y el software— es el principal elemento del modelado. El caso de uso se obtiene durante la indagación de los requerimientos y define las etapas clave de una función o interacción específica. El grado de formalidad del caso de uso y su nivel de detalle varía, pero

el resultado final da las entradas necesarias a todas las demás actividades del modelado. Los escenarios también pueden ser descritos con el uso de un diagrama de actividades —representación gráfica parecida a un diagrama de flujo que ilustra el flujo del procesamiento dentro de un escenario específico—. Los diagramas de canal (swimlane) ilustran la forma en la que se asigna el flujo del procesamiento a distintos actores o clases.

El modelado de datos se utiliza para describir el espacio de información que será construido o manipulado por el software. El modelado de datos comienza con la representación de los objetos de datos —información compuesta que debe ser entendida por el software—. Se identifican los atributos de cada objeto de datos y se describen las relaciones entre estos objetos.

El modelado basado en clases utiliza información obtenida de los elementos del modelado basado en el escenario y en datos, para identificar las clases de análisis. Se emplea un análisis gramatical para obtener candidatas a clase, atributos y operaciones, a partir de narraciones basadas en texto. Se definen criterios para definir una clase. Para definir las relaciones entre clases, se emplean tarjetas índice clase-responsabilidad-colaborador. Además, se aplican varios elementos de la notación UML para definir jerarquías, relaciones, asociaciones, agregaciones y dependencias entre clases. Se emplean paquetes de análisis para clasificar y agrupar clases, de manera que sean más manejables en sistemas grandes.

Problemas y puntos por evaluar

6.1. ¿Es posible comenzar a codificar de inmediato después de haber creado un modelo de análisis? Explique su respuesta y luego defienda el punto de vista contrario.

6.2. Una regla práctica del análisis es que el modelo "debe centrarse en los requerimientos visibles dentro del dominio del problema o negocio". ¿Qué tipos de requerimientos *no* son visibles en dichos dominios? Dé algunos ejemplos.

6.3. ¿Cuál es el propósito del análisis del dominio? ¿Cómo se relaciona con el concepto de patrones de requerimientos?

6.4. ¿Es posible desarrollar un modelo de análisis eficaz sin desarrollar los cuatro elementos que aparecen en la figura 6.3? Explique su respuesta.

6.5. Se pide al lector que construya uno de los siguientes sistemas:

a) Sistema de inscripción a la universidad basado en red.
b) Sistema de procesamiento de órdenes basado en web para una tienda de computadoras.
c) Sistema de facturación simple para un negocio pequeño.
d) Libro de cocina basado en internet, construido en un horno eléctrico o de microondas.

Seleccione el sistema que le interese y desarrolle un diagrama entidad-relación que describa los objetos de datos, relaciones y atributos.

6.6. El departamento de obras públicas de una gran ciudad ha decidido desarrollar un sistema de seguimiento y reparación de baches, basado en web (SSRB).

Cuando se reportan los baches, se registran en "sistema de reparación del departamento de obras públicas" y se les asigna un número de identificación, almacenado según la calle, tamaño (en una escala de 1 a 10), ubicación (en medio, cuneta, etc.), distrito (se determina con la dirección en la calle) y prioridad de reparación (determinada por el tamaño del bache). Los datos de la orden de trabajo se asocian con cada bache e incluyen su ubicación y tamaño, número de identificación del equipo de reparación, número de personas en dicho equipo, equipo asignado, horas dedicadas a la reparación, estado del bache (trabajo en proceso, reparado, reparación temporal, no reparado), cantidad de material de relleno utilizado y costo de la reparación (calculado a partir de las horas dedicadas, número de personas, materiales y equipo empleado). Por último, se crea un archivo de daños para mantener la información sobre daños reportados debido al bache, y se incluye el nombre y dirección del ciudadano, número telefónico, tipo de daño y cantidad de dinero por el daño. El SSRB es un sistema en línea, todas las solicitudes se harán en forma interactiva.

a) Dibuje un diagrama UML para el caso de uso del sistema SSRB. Tendrá que hacer algunas suposiciones sobre la manera en la que un usuario interactúa con el sistema.
b) Desarrolle un modelo de clase para el sistema SSRB.

6.7. Escriba un caso de uso basado en formato para el sistema de administración del hogar *CasaSegura* descrito de manera informal en el recuadro de la sección 6.5.4.

6.8. Desarrolle un conjunto completo de tarjetas índice de modelo CRC, sobre el producto o sistema que elija como parte del problema 6.5.

6.9. Revise con sus compañeros las tarjetas índice CRC. ¿Cuántas clases, responsabilidades y colaboradores adicionales fueron agregados como consecuencia de la revisión?

6.10. ¿Qué es y cómo se usa un paquete de análisis?

Lecturas adicionales y fuentes de información

Los casos de uso son el fundamento de todos los enfoques del modelado de los requerimientos. El tema se analiza con amplitud en Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander y Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Bittner y Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01b] y en otras referencias mencionadas en los capítulos 5 y 6.

El modelado de datos constituye un método útil para examinar el espacio de información. Los libros de Hoberman [Hob06] y Simsion y Witt [Sim05] hacen tratamientos razonablemente amplios. Además, Allen y Terry (*Beginning Relational Data Modeling*, 2a. ed., Apress, 2005), Allen (*Data Modeling for Everyone*, Word Press, 2002), Teorey *et al.* (*Database Modeling and Design: Logical Design*, 4a. ed., Morgan Kaufmann, 2005) y Carlis y Maguire (*Mastering Data Modeling*, Addison-Wesley, 2000) presentan métodos de aprendizaje detallados para crear modelos de datos de calidad industrial. Un libro interesante escrito por Hay (*Data Modeling Patterns*, Dorset House, 1995) presenta patrones comunes de modelos de datos que se encuentran en muchos negocios diferentes.

Análisis de técnicas de modelado UML que pueden aplicarse tanto para el análisis como para el diseño se encuentran en O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, Wiley, 2005), Arlow y Neustadt (*UML, 2 and the Unified Process*, 2a. ed., Addison-Wesley, 2005), Roques (*UML in Practice*, Wiley, 2004), Dennis *et al.* (*Systems Analysis and Design with UML Version 2.0*, Wiley, 2004), Larman (*Applying UML and Patterns*, 2a. ed., Prentice-Hall, 2001) y Rosenberg y Scott (*Use Case Driven Object Modeling with UML*, Addison-Wesley, 1999).

En internet existe una amplia variedad de fuentes de información sobre el modelado de requerimientos. En el sitio web del libro, en **www.mhhe.com/engcs/comsci/pressman/professioal/olc/ser.htm**, se halla una lista actualizada de referencias en web que son relevantes para el modelado del análisis.

CAPÍTULO

7 MODELADO DE LOS REQUERIMIENTOS: FLUJO, COMPORTAMIENTO, PATRONES Y WEBAPPS

CONCEPTOS CLAVE

Después de estudiar en el capítulo 6 los casos de uso, modelado de datos y modelos basados en clase, es razonable preguntar: "¿no son suficientes representaciones del modelado de los requerimientos?"

La única respuesta razonable es: "depende".

Para ciertos tipos de software, el caso de uso puede ser la única representación para modelar los requerimientos que se necesite. Para otros, se escoge un enfoque orientado a objetos y se desarrollan modelos basados en clase. Pero en otras situaciones, los requerimientos de las aplicaciones complejas demandan el estudio de la manera como se transforman los objetos de datos cuando se mueven a través del sistema; cómo se comporta una aplicación a consecuencia de eventos externos; si el conocimiento del dominio existente puede adaptarse al problema en cuestión; o, en el caso de sistemas y aplicaciones basados en web, cómo unificar el contenido y la funcionalidad para dar al usuario final la capacidad de navegar con éxito por una *webapp* a fin de lograr sus objetivos.

7.1 REQUERIMIENTOS QUE MODELAN LAS ESTRATEGIAS

Un punto de vista del modelado de los requerimientos, llamada *análisis estructurado*, considera como entidades separadas los datos y los procesos que los transforman. Los objetos de datos se modelan en una forma que define sus atributos y relaciones. Los procesos que manipulan objetos de datos se modelan de una forma que muestra cómo transforman los datos cuando los

UNA MIRADA RÁPIDA

¿Qué es? El modelo de requerimientos tiene muchas dimensiones diferentes. En este capítulo, el lector aprenderá acerca de modelos orientados al flujo, de modelos de comportamiento y de las consideraciones especiales del análisis de requerimientos que entran en juego cuando se desarrollan *webapps*. Cada una de estas representaciones de modelado complementa los casos de uso, modelos de datos y modelos basados en clases que se estudiaron en el capítulo 6.

¿Quién lo hace? Un ingeniero de software (a veces llamado analista) construye el modelo con el uso de los requerimientos recabados entre varios participantes.

¿Por qué es importante? La perspectiva de los requerimientos del software crece en proporción directa al número de dimensiones distintas del modelado de los requerimientos. Aunque quizá no se tenga el tiempo, los recursos o la inclinación para desarrollar cada representación sugerida en este capítulo y en el anterior, debe reconocerse que cada enfoque diferente de modelado proporciona una forma distinta de ver el problema. En consecuencia, el lector (y otros participantes) estará mejor preparado para evaluar si ha especificado en forma apropiada aquello que debe lograrse.

¿Cuáles son los pasos? El modelado orientado al flujo da una indicación de la forma en la que las funciones de procesamiento transforman los objetos de datos. El modelado del comportamiento ilustra los estados del sistema y sus clases, así como el efecto que tienen los eventos sobre dichos estados. El modelado basado en patrones utiliza el conocimiento del dominio existente para facilitar el análisis de los requerimientos. Los modelos de requerimientos con *webapps* están adaptados especialmente para representar requerimientos relacionados con contenido, interacción, función y configuración.

¿Cuál es el producto final? Para el modelado de los requerimientos, es posible escoger una gran variedad de formas basadas en texto y diagramas. Cada una de estas representaciones da una perspectiva de uno o más de los elementos del modelo.

¿Cómo me aseguro de que lo hice bien? Debe revisarse si los productos del trabajo del modelado de los requerimientos son correctos, completos y congruentes. Deben reflejar las necesidades de todos los participantes y establecer los fundamentos desde los que se llevará a cabo el diseño.

objetos de datos fluyen por el sistema. Un segundo enfoque del modelado de análisis, llamado *análisis orientado a objetos*, se centra en la definición de clases y en el modo en el que colaboran una con otra para cumplir con los requerimientos del cliente.

Aunque el modelo de análisis que se propone en este libro combina características de ambos enfoques, es frecuente que los equipos del software elijan uno de ellos y excluyan las representaciones del otro. La pregunta no es cuál es mejor, sino qué combinación de representaciones dará a los participantes el mejor modelo de los requerimientos del software y cuál será el mejor puente para cruzar la brecha hacia el diseño del software.

7.2 MODELADO ORIENTADO AL FLUJO

Aunque algunos ingenieros de software perciben el modelado orientado al flujo como una técnica obsoleta, sigue siendo una de las notaciones más usadas actualmente para hacer el análisis de los requerimientos.[1] Si bien el *diagrama de flujo de datos* (DFD) y la información relacionada no son una parte formal del UML, se utilizan para complementar los diagramas de éste y amplían la perspectiva de los requerimientos y del flujo del sistema.

Algunas personas afirman que los DFD son obsoletos y que no hay lugar para ellos en la práctica moderna. Ese punto de vista excluye un modo potencialmente útil de representación en el nivel del análisis. Si ayuda, use DFD.

El DFD adopta un punto de vista del tipo entrada-proceso-salida para el sistema. Es decir, los objetos de datos entran al sistema, son transformados por elementos de procesamiento y los objetos de datos que resultan de ello salen del software. Los objetos de datos se representan con flechas con leyendas y las transformaciones, con círculos (también llamados burbujas). El DFD se presenta en forma jerárquica. Es decir, el primer modelo de flujo de datos (en ocasiones llamado DFD de nivel 0 o *diagrama de contexto*) representa al sistema como un todo. Los diagramas posteriores de flujo de datos mejoran el diagrama de contexto y dan cada vez más detalles en los niveles siguientes.

7.2.1 Creación de un modelo de flujo de datos

Cita:

"El propósito de los diagramas de flujo de datos es proveer un puente semántico entre los usuarios y los desarrolladores de sistemas."

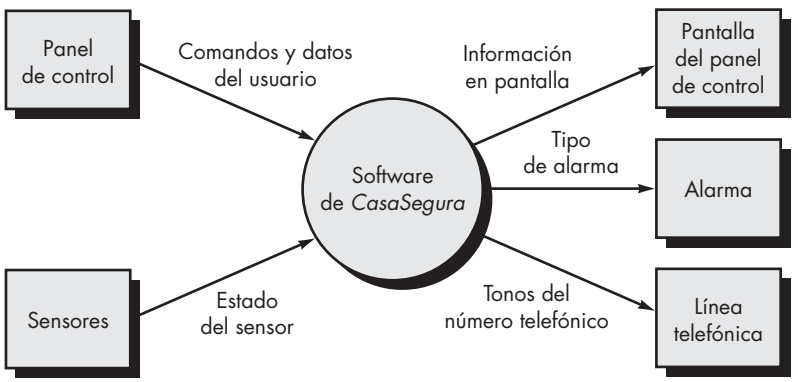
Kenneth Kozar

El diagrama de flujo de datos permite desarrollar modelos del dominio de la información y del dominio funcional. A medida que el DFD se mejora con mayores niveles de detalle, se efectúa la descomposición funcional implícita del sistema. Al mismo tiempo, la mejora del DFD da como resultado el refinamiento de los datos conforme avanzan por los procesos que constituyen la aplicación.

Unos cuantos lineamientos sencillos ayudan muchísimo durante la elaboración del diagrama de flujo de los datos: 1) el nivel 0 del diagrama debe ilustrar el software o sistema como una sola

FIGURA 7.1

DFD en el nivel de contexto para la función de seguridad de *CasaSegura*



1 El modelado del flujo de datos es una actividad fundamental del *análisis estructurado*.

burbuja; 2) debe anotarse con cuidado las entradas y salidas principales; 3) la mejora debe comenzar por aislar procesos candidatos, objetos de datos y almacenamiento de éstos, para representarlos en el siguiente nivel; 4) todas las flechas y burbujas deben etiquetarse con nombres significativos; 5) de un nivel a otro, debe mantenerse la *continuidad del flujo de información*,[2] y 6) debe mejorarse una burbuja a la vez. Existe la tendencia natural a complicar innecesariamente el diagrama de flujo de los datos. Esto sucede cuando se trata de ilustrar demasiados detalles en una etapa muy temprana o representar aspectos de procedimiento del software en lugar del flujo de la información.

Conforme se mejora cada nivel del DFD, debe mantenerse la continuidad del flujo de la información. Esto significa que las entradas y salidas en cierto nivel deben ser las mismas en un nivel mejorado.

Para ilustrar el uso del DFD y la notación relacionada, consideremos de nuevo la función de seguridad de *CasaSegura*. En la figura 7.1 se muestra un DFD de nivel 0 para dicha función. Las *entidades externas* principales (cuadrados) producen información para uso del sistema y consumen información generada por éste. Las flechas con leyendas representan objetos de datos o jerarquías de éstos. Por ejemplo, los **comandos** y **datos del usuario** agrupan todos los comandos de configuración, todos los comandos de activación/desactivación, todas las diferentes interacciones y todos los datos que se introducen para calificar o expandir un comando.

Ahora debe expandirse el DFD de nivel 0 a un modelo de flujo de datos de nivel 1. Pero, ¿cómo hacerlo? Según el enfoque sugerido en el capítulo 6, debe aplicarse un "análisis gramatical" [Abb83] a la narración del caso de uso que describe la burbuja en el nivel del contexto. Es decir, se aíslan todos los sustantivos (y frases sustantivadas) y verbos (y frases verbales) en la narración del procesamiento de *CasaSegura* obtenida durante la primera reunión realizada para recabar los requerimientos. Recordemos el análisis gramatical del texto que narra el procesamiento presentado en la sección 6.5.1:

El análisis gramatical no es a prueba de todo, pero da un impulso excelente para arrancar si se tienen dificultades para definir objetos de datos y las transformaciones que operan sobre ellos.

> La función de seguridad CasaSegura *permite* que el propietario *configure* el sistema de seguridad cuando se *instala*, *vigila* todos los sensores *conectados* al sistema de seguridad e *interactúa* con el propietario a través de internet, una PC o un panel de control.
>
> Durante la instalación, la PC de CasaSegura se utiliza para *programar* y *configurar* el sistema. Se asigna a cada sensor un número y un tipo, se programa una clave maestra para *activar* y *desactivar* el sistema, y se *introducen* números telefónicos para *marcar* cuando ocurre un evento de sensor.
>
> Cuando se *reconoce* un evento de sensor, el software *invoca* una alarma audible instalada en el sistema. Después de un tiempo de retraso que especifica el propietario durante las actividades de configuración del sistema, el software marca un número telefónico de un servicio de monitoreo, *proporciona* información acerca de la ubicación y *reporta* la naturaleza del evento detectado. El número telefónico *vuelve a marcarse* cada 20 segundos hasta que se *obtiene* la conexión telefónica.
>
> El propietario *recibe* información de seguridad a través de un panel de control, la PC o un navegador, lo que en conjunto se llama interfaz. La interfaz *despliega* en el panel de control, en la PC o en la ventana del navegador mensajes de aviso e información del estado del sistema. La interacción del propietario tiene la siguiente forma...

Asegúrese de que la narración del procesamiento que se va a analizar gramaticalmente está escrita con el mismo nivel de abstracción.

En relación con el análisis gramatical, los verbos son los procesos de *CasaSegura* y se representarán como burbujas en un DFD posterior. Los sustantivos son entidades externas (cuadros), datos u objetos de control (flechas) o almacenamiento de datos (líneas dobles). De lo estudiado en el capítulo 6 se recuerda que los sustantivos y verbos se asocian entre sí (por ejemplo, a cada sensor se asigna un número y tipo; entonces, **número** y **tipo** son atributos del objeto de datos **sensor**). De modo que al realizar un análisis gramatical de la narración de procesamiento en cualquier nivel del DFD, se genera mucha información útil sobre la manera de proceder para la mejora del nivel siguiente. En la figura 7.2 se presenta un DFD de nivel 1 con el empleo de esta información. El proceso en el nivel de contexto que se ilustra en la figura 7.1 ha sido expandido

2 Es decir, los objetos de datos que entran al sistema o a cualquier transformación en cierto nivel deben ser los mismos objetos de datos (o sus partes constitutivas) que entran a la transformación en un nivel mejorado.

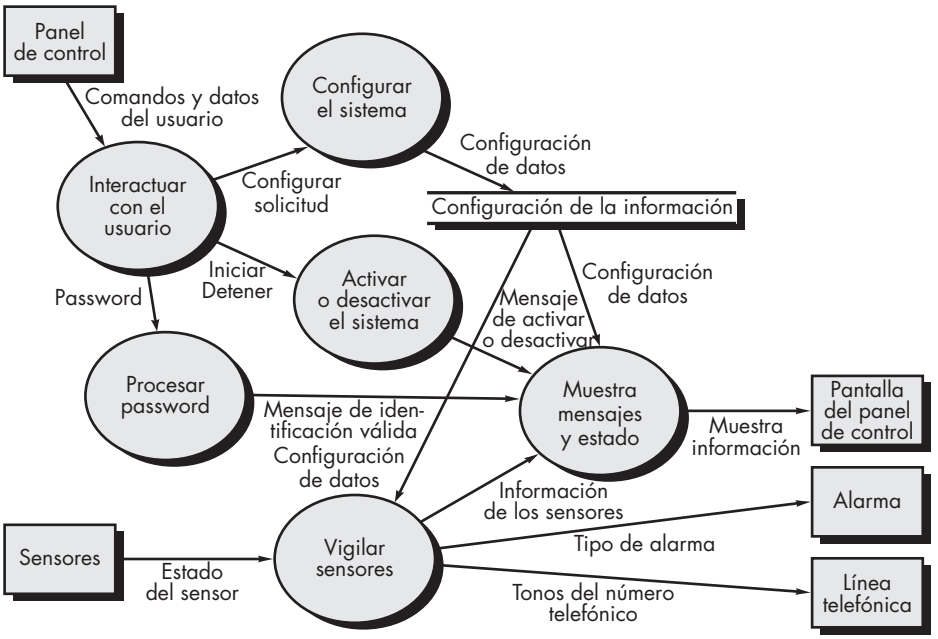


FIGURA 7.2
DFD de nivel 1 para la función de seguridad de *CasaSegura*

a seis procesos derivados del estudio del análisis gramatical. De manera similar, el flujo de información entre procesos del nivel 1 ha surgido de dicho análisis. Además, entre los niveles 0 y 1 se mantiene la continuidad del flujo de información.

Los procesos representados en el nivel 1 del DFD pueden mejorarse más hacia niveles inferiores. Por ejemplo, el proceso *vigilar sensores* se mejora en el DFD de nivel 2, como se aprecia en la figura 7.3. De nuevo, observe que entre los niveles se ha mantenido la continuidad del flujo de información.

La mejoría de los DFD continúa hasta que cada burbuja realiza una función simple. Es decir, hasta que el proceso representado por la burbuja ejecuta una función que se implementaría fácilmente como componente de un programa. En el capítulo 8 se estudia un concepto llamado

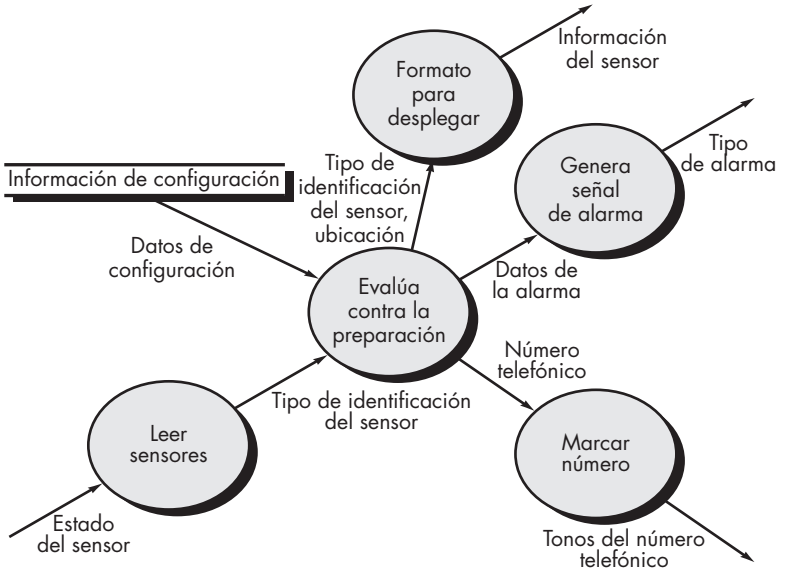


FIGURA 7.3
DFD de nivel 2 que mejora el proceso *vigilar sensores*

cohesión, que se utiliza para evaluar el objeto del procesamiento de una función dada. Por ahora, se trata de mejorar los DFD hasta que cada burbuja tenga "un solo pensamiento".

7.2.2 Creación de un modelo de flujo de control

Para ciertos tipos de aplicaciones, el modelo de datos y el diagrama de flujo de datos es todo lo que se necesita para obtener una visión significativa de los requerimientos del software. Sin embargo, como ya se dijo, un gran número de aplicaciones son "motivadas" por eventos y no por datos, producen información de control en lugar de reportes o pantallas, y procesan información con mucha atención en el tiempo y el desempeño. Tales aplicaciones requieren el uso del *modelado del flujo de control*, además de modelar el flujo de datos.

Se dijo que un evento o aspecto del control se implementa como valor booleano (por ejemplo, verdadero o falso, encendido o apagado, 1 o 0) o como una lista discreta de condiciones (vacío, bloqueado, lleno, etc.). Se sugieren los lineamientos siguientes para seleccionar eventos candidatos potenciales:

? ¿Cómo seleccionar los eventos potenciales para un diagrama de flujo de control, de estado o CSPEC?

- Enlistar todos los sensores que son "leídos" por el software.
- Enlistar todas las condiciones de interrupción.
- Enlistar todos los "interruptores" que son activados por un operador.
- Enlistar todas las condiciones de los datos.
- Revisar todos los "aspectos de control" como posibles entradas o salidas de especificación del control, según el análisis gramatical de sustantivos y verbos que se aplicó a la narración del procesamiento.
- Describir el comportamiento de un sistema con la identificación de sus estados, identificar cómo se llega a cada estado y definir las transiciones entre estados.
- Centrarse en las posibles omisiones, error muy común al especificar el control; por ejemplo, se debe preguntar: "¿hay otro modo de llegar a este estado o de salir de él?"

Entre los muchos eventos y aspectos del control que forman parte del software de *CasaSegura*, se encuentran **evento de sensor** (por ejemplo, un sensor se descompone), **bandera de cambio** (señal para que la pantalla cambie) e **interruptor iniciar/detener** (señal para encender o apagar el sistema).

7.2.3 La especificación de control

Una *especificación de control* (CSPEC) representa de dos maneras distintas el comportamiento del sistema (en el nivel desde el que se hizo referencia a él).[3] La CSPEC contiene un diagrama de estado que es una especificación secuencial del comportamiento. También puede contener una tabla de activación del programa, especificación combinatoria del comportamiento.

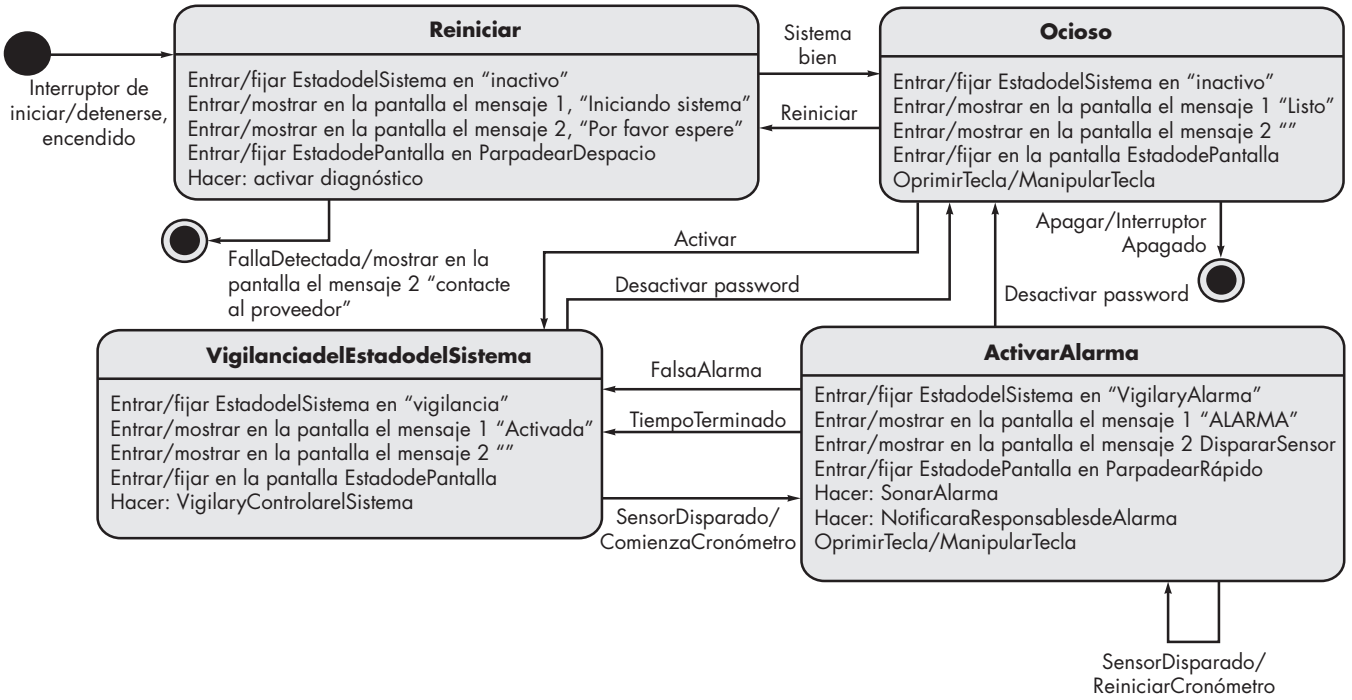
La figura 7.4 ilustra un diagrama de estado preliminar[4] para el nivel 1 del modelo de flujo de control para *CasaSegura*. El diagrama indica cómo responde el sistema a eventos conforme pasa por los cuatro estados definidos en este nivel. Con la revisión del diagrama de estado se determina el comportamiento del sistema, y, lo que es más importante, se investiga si existen "agujeros" en el comportamiento especificado.

Por ejemplo, el diagrama de estado (véase la figura 7.4) indica que las transiciones del estado **Ocioso** ocurren si el sistema se reinicia, se activa o se apaga. Si el sistema se activa (por ejem-

3 En la sección 7.3 se presenta notación adicional de modelado por comportamiento.

4 La notación del diagrama de estado que se emplea aquí sigue la del UML. En el análisis estructurado se dispone de un "diagrama de transición de estado", pero el formato UML es mejor en contenido y representación de la información.

FIGURA 7.4 Diagrama de estado para la función de seguridad de *CasaSegura*



plo, se enciende el sistema de alarma), ocurre una transición al estado **VigilanciadelEstadodelSistema**, los mensajes en la pantalla cambian como se muestra y se invoca el proceso *SistemadeVigilanciayControl*. Fuera del estado **SistemadeVigilanciayControl** ocurren dos transiciones: 1) cuando se desactiva el sistema hay una transición de regreso al estado **Ocioso**; 2) cuando se dispara un sensor en el estado **ActivarAlarma**. Durante la revisión se consideran todas las transiciones y el contenido de todos los estados.

La tabla de activación del proceso (TAP) es un modo algo distinto de representar el comportamiento. La TAP representa la información contenida en el diagrama de estado en el contexto de los procesos, no de los estados. Es decir, la tabla indica cuáles procesos (burbujas) serán invocados en el modelo del flujo cuando ocurra un evento. La TAP se usa como guía para un diseñador que debe construir una ejecución que controle los procesos representados en este nivel. En la figura 7.5 se aprecia una TAP para el nivel 1 del modelo de flujo del software de *CasaSegura*.

La CSPEC describe el comportamiento del sistema, pero no da información acerca del funcionamiento interno de los procesos que se activan como resultado de dicho comportamiento. En la sección 7.2.4 se estudia la notación de modelación que da esta información.

7.2.4 La especificación del proceso

La *especificación del proceso* (PSPEC) se utiliza para describir todos los procesos del modelo del flujo que aparecen en el nivel final de la mejora. El contenido de la especificación del proceso incluye el texto narrativo, una descripción del lenguaje de diseño del programa[5] del algoritmo del proceso, ecuaciones matemáticas, tablas o diagramas de actividad UML. Si se da una PSPEC

5 El lenguaje de diseño del programa (LDP) mezcla la sintaxis del lenguaje de programación con el texto narrativo a fin de dar detalles del diseño del procedimiento. En el capítulo 10 se analiza el LDP.

FIGURA 7.5

Tabla de activación del proceso para la función de seguridad de *CasaSegura*

eventos de entrada						
evento de sensor	0	0	0	0	1	0
bandera de parpadeo	0	0	1	1	0	0
interruptor de iniciar o detener	0	1	0	0	0	0
estado de la acción en pantalla terminado	0	0	0	1	0	0
en marcha	0	0	1	0	0	0
tiempo terminado	0	0	0	0	0	1
salida						
señal de alarma	0	0	0	0	1	0
activación del proceso						
vigilar y controlar el sistema	0	1	0	0	1	1
activar o desactivar el sistema	0	1	0	0	0	0
mostrar mensajes y estado	1	0	1	1	1	1
interactuar con el usuario	1	0	0	1	0	1

PUNTO CLAVE

La PSPEC es una "miniespecificación" de cada transformación en el nivel más bajo de mejora de un DFD.

que acompañe a cada burbuja del modelo del flujo, se crea una "miniespecificación" que sirve como guía para diseñar la componente del software que implementará la burbuja.

Para ilustrar el uso de la PSPEC, considere la transformación *procesar password* representada en el modelo de flujo de la figura 7.2. La PSPEC de esta función adopta la forma siguiente:

> **PSPEC: procesar password (en el panel de control).** La transformación *procesar password* realiza la validación en el panel de control para la función de seguridad de *CasaSegura*. *Procesar password* recibe un password de cuatro dígitos de la función *interactuar con usuario*. Primero, el password se compara con el password maestro almacenado dentro del sistema. Si el password maestro coincide, se pasa <mensaje de identificación válida = verdadero> a la función *mostrar mensaje y estado*. Si el

CASASEGURA

Modelado del flujo de datos

La escena: Cubículo de Jamie, después de que terminó la última junta para recabar los requerimientos.

Participantes: Jamie, Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

(Jamie presenta a Ed y a Vinod los dibujos que hizo de los modelos que se muestran en las figuras 7.1 a 7.5.)

Jamie: En la universidad tomé un curso de ingeniería de software y aprendí esto. El profesor dijo que es un poco anticuado, pero, saben, me ayuda a aclarar las cosas.

Ed: Está muy bien. Pero no veo ninguna clase de objetos ahí.

Jamie: No... Esto sólo es un modelo del flujo con un poco de comportamiento ilustrado.

Vinod: Así que estos DFD representan la E-P-S del software, ¿o no?

Ed: ¿E-P-S?

Vinod: Entrada-proceso-salida. En realidad, los DFD son muy intuitivos... si se observan un rato, indican cómo fluyen los objetos de datos por el sistema y cómo se transforman mientras lo hacen.

Ed: Parece que pudiéramos convertir cada burbuja en un componente ejecutable... al menos en el nivel más bajo del DFD.

Jamie: Ésa es la mejor parte, sí se puede. En realidad, hay una forma de traducir los DFD a una arquitectura de diseño.

Ed: ¿En verdad?

Jamie: Sí, pero primero tenemos que desarrollar un modelo completo de los requerimientos, y esto no lo es.

Vinod: Bueno, es un primer paso, pero vamos a tener que enfrentar los elementos basados en clases y también los aspectos de comportamiento, aunque el diagrama de estado y la TAP hacen algo de eso.

Ed: Tenemos mucho trabajo por hacer y poco tiempo.

(Entra al cubículo Doug, el gerente de ingeniería de software.)

Doug: Así que dedicaremos los siguientes días a desarrollar el modelo de los requerimientos, ¿eh?

Jamie (con orgullo): Ya comenzamos.

Doug: Qué bueno, tenemos mucho trabajo por hacer y poco tiempo.

(Los tres ingenieros de software se miran entre sí y sonríen.)

password maestro no coincide, los cuatro dígitos se comparan con una tabla de passwords secundarios (que deben asignarse para recibir invitados o trabajadores que necesiten entrar a la casa cuando el propietario no esté presente). Si el password coincide con una entrada de la tabla, se pasa <mensaje de identificación válida = verdadero> a la función *mostrar mensaje y estado*. Si no coinciden, se pasa <mensaje de identificación válida = falso> a la función de mostrar mensaje y estado.

Si en esta etapa se desean detalles algorítmicos adicionales, también puede incluirse una representación del lenguaje de diseño del programa (LDP) como parte de la PSPEC. Sin embargo, muchos profesionales piensan que la versión LDP debe posponerse hasta comenzar el diseño de los componentes.

HERRAMIENTAS DE SOFTWARE

Análisis estructurado

Objetivo: Las herramientas de análisis estructurado permiten que un ingeniero de software cree modelos de datos, de flujo y de comportamiento en una forma que permite la consistencia y continuidad con facilidad para hacer la revisión, edición y ampliación. Los modelos creados con estas herramientas dan al ingeniero de software la perspectiva de la representación del análisis y lo ayudan a eliminar errores antes de que éstos se propaguen al diseño o, lo que sería peor, a la implementación.

Mecánica: Las herramientas de esta categoría son un "diccionario de datos", como la base de datos central para describir todos los objetos de datos. Una vez definidas las entradas del diccionario, se crean diagramas entidad-relación y se desarrollan las jerarquías de los objetos. Las características de los diagramas de flujo de datos permiten que sea fácil crear este modelo gráfico y también proveen de características para generar PSPEC y CSPEC. Asimismo, las herramientas de análisis permiten que el ingeniero de software produzca modelos de comportamiento con el empleo del diagrama de estado como notación operativa.

Herramientas representativas.[6]

MacA&D, WinA&D, desarrolladas por software Excel (**www.excelsoftware.com**), brinda un conjunto de herramientas de análisis y diseño sencillas y baratas para computadoras Mac y Windows.

MetaCASE Workbench, desarrollada por MetaCase Consulting (**www.metacase.com**), es una metaherramienta utilizada para definir un método de análisis o diseño (incluso análisis estructurado) y sus conceptos, reglas, notaciones y generadores.

System Architect, desarrollado por Popkin Software (**www.popkin.com**), da una amplia variedad de herramientas de análisis y diseño, incluso para modelar datos y hacer análisis estructurado.

7.3 CREACIÓN DE UN MODELO DE COMPORTAMIENTO

? ¿Cómo se modela la reacción del software ante algún evento externo?

La notación de modelado que hemos estudiado hasta el momento representa elementos estáticos del modelo de requerimientos. Es hora de hacer la transición al comportamiento dinámico del sistema o producto. Para hacerlo, dicho comportamiento se representa como función de eventos y tiempo específicos.

El *modelo de comportamiento* indica la forma en la que responderá el software a eventos o estímulos externos. Para generar el modelo deben seguirse los pasos siguientes:

1. Evaluar todos los casos de uso para entender por completo la secuencia de interacción dentro del sistema.
2. Identificar los eventos que conducen la secuencia de interacción y que entienden el modo en el que éstos se relacionan con objetos específicos.
3. Crear una secuencia para cada caso de uso.
4. Construir un diagrama de estado para el sistema.
5. Revisar el modelo de comportamiento para verificar la exactitud y consistencia.

En las secciones siguientes se estudia cada uno de estos pasos.

6 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

7.3.1 Identificar los eventos con el caso de uso

En el capítulo 6 se aprendió que el caso de uso representa una secuencia de actividades que involucra a los actores y al sistema. En general, un evento ocurre siempre que el sistema y un actor intercambian información. En la sección 7.2.3 se dijo que un evento *no* es la información que se intercambia, sino el hecho de que se intercambió la información.

Un caso de uso se estudia para efectos del intercambio de información. Para ilustrarlo, volvamos al caso de uso de una parte de la función de seguridad de *CasaSegura*.

> El propietario utiliza el teclado para escribir un password de cuatro dígitos. El password se compara con el password válido guardado en el sistema. Si el password es incorrecto, el panel de control emitirá un sonido una vez y se reiniciará para recibir entradas adicionales. Si el password es correcto, el panel de control queda en espera de otras acciones.

Las partes subrayadas del escenario del caso de uso indican eventos. Debe identificarse un actor para cada evento, anotarse la información que se intercambia y enlistarse cualesquiera condiciones o restricciones.

Como ejemplo de evento común considere la frase subrayada en el caso de uso "el propietario utiliza el teclado para escribir un password de cuatro dígitos". En el contexto del modelo de los requerimientos, el objeto **PropietariodeCasa**[7] transmite un evento al objeto **PaneldeControl**. El evento tal vez se llame *password introducido*. La información que se transfiere son los cuatro dígitos que constituyen el password, pero ésta no es una parte esencial del modelo de comportamiento. Es importante observar que ciertos eventos tienen un efecto explícito en el flujo del control del caso de uso, mientras que otros no lo tienen. Por ejemplo, el evento *password introducido* no cambia explícitamente el flujo del control del caso de uso, pero los resultados del evento *password comparado* (derivado de la interacción el "password se compara con el password válido guardado en el sistema") tendrán un efecto explícito en el flujo de información y control del software *CasaSegura*.

Una vez identificados todos los eventos, se asignan a los objetos involucrados. Los objetos son responsables de la generación de eventos (por ejemplo, **Propietario** genera el evento *password introducido*) o de reconocer los eventos que hayan ocurrido en cualquier lugar (**PaneldeControl** reconoce el resultado binario del evento *password comparado*).

7.3.2 Representaciones de estado

En el contexto del modelado del comportamiento deben considerarse dos caracterizaciones diferentes de los estados: 1) el estado de cada clase cuando el sistema ejecuta su función y 2) el estado del sistema según se observa desde el exterior cuando realiza su función.[8]

PUNTO CLAVE

El sistema tiene estados que representan un comportamiento específico observable desde el exterior; una clase tiene estados que representan su comportamiento cuando el sistema realiza sus funciones.

El estado de una clase tiene características tanto pasivas como activas [Cha93]. Un *estado pasivo* es sencillamente el estado actual de todos los atributos de un objeto. Por ejemplo, el estado pasivo de la clase **Jugador** (en la aplicación de juego de video que se vio en el capítulo 6) incluiría los atributos actuales **posición** y **orientación** de **Jugador**, así como otras características de éste que sean relevantes para el juego (por ejemplo, un atributo que incluya **deseos mágicos restantes**). El *estado activo* de un objeto indica el estado actual del objeto conforme pasa por una transformación o procesamiento continuos. La clase **Jugador** tal vez tenga los siguientes estados activos: *moverse, en descanso, herido, en curación, atrapado, perdido*, etc. Debe ocurrir un evento (en ocasiones llamado *disparador o trigger*) para forzar a un objeto a hacer una transición de un estado activo a otro.

---

7 En este ejemplo se supone que cada usuario (propietario) que interactúe con *CasaSegura* tiene un password de identificación, por lo que es un objeto legítimo.

8 Los diagramas de estado presentados en el capítulo 6 y en la sección 7.3.2 ilustran el estado del sistema. En esta sección, nuestro análisis se centrará en el estado de cada clase dentro del modelo del análisis.

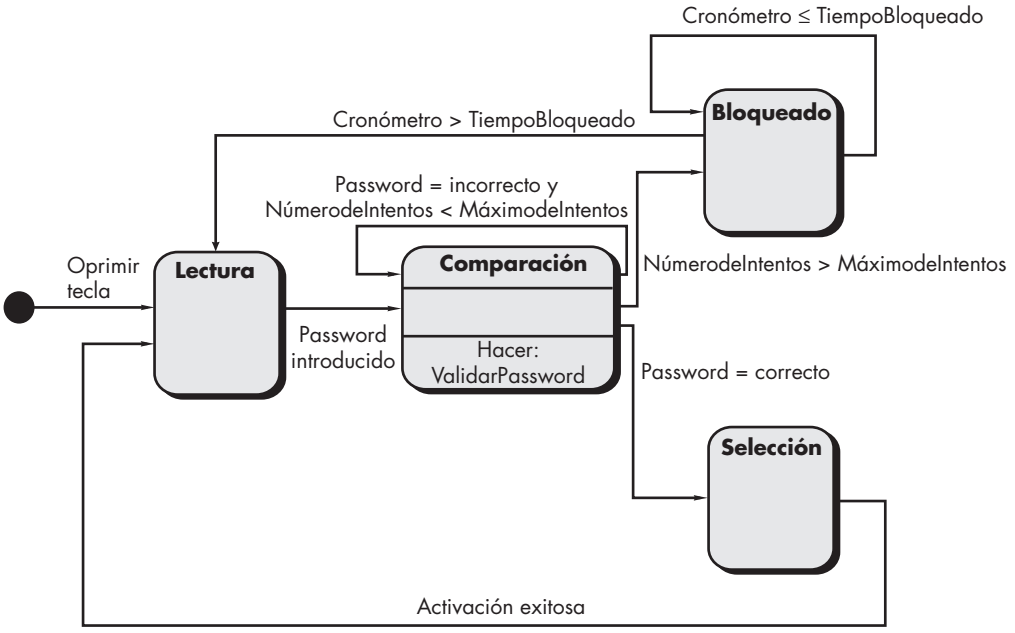


FIGURA 7.6
Diagrama de estado para la clase PaneldeControl

En los párrafos que siguen se analizan dos representaciones distintas del comportamiento. La primera indica la manera en la que una clase individual cambia su estado con base en eventos externos, y la segunda muestra el comportamiento del software como una función del tiempo.

Diagramas de estado para clases de análisis. Un componente de modelo de comportamiento es un diagrama de estado UML[9] que representa estados activos para cada clase y los eventos (disparadores) que causan cambios en dichos estados activos. La figura 7.6 ilustra un diagrama de estado para el objeto **PaneldeControl** en la función de seguridad de *CasaSegura*.

Cada flecha que aparece en la figura 7.6 representa una transición de un estado activo de un objeto a otro. Las leyendas en cada flecha representan el evento que dispara la transición. Aunque el modelo de estado activo da una perspectiva útil de la "historia de la vida" de un objeto, es posible especificar información adicional para llegar a más profundidad en la comprensión del comportamiento de un objeto. Además de especificar el evento que hace que la transición ocurra, puede especificarse una guardia y una acción [Cha93]. Una *guardia* es una condición booleana que debe satisfacerse para que tenga lugar una transición. Por ejemplo, la guardia para la transición del estado de "lectura" al de "comparación" en la figura 7.6 se determina con el análisis del caso de uso:

Si (password de entrada = 4 dígitos) entonces comparar con el password guardado

En general, la guardia para una transición depende del valor de uno o más atributos de un objeto. En otras palabras, depende del estado pasivo del objeto.

Una *acción* sucede en forma concurrente con la transición de estado o como consecuencia de ella, y por lo general involucra una o más operaciones (responsabilidades) del objeto. Por

9 Si el lector no está familiarizado con UML, puede consultar en el apéndice 1 una breve introducción a esta importante notación de modelación.

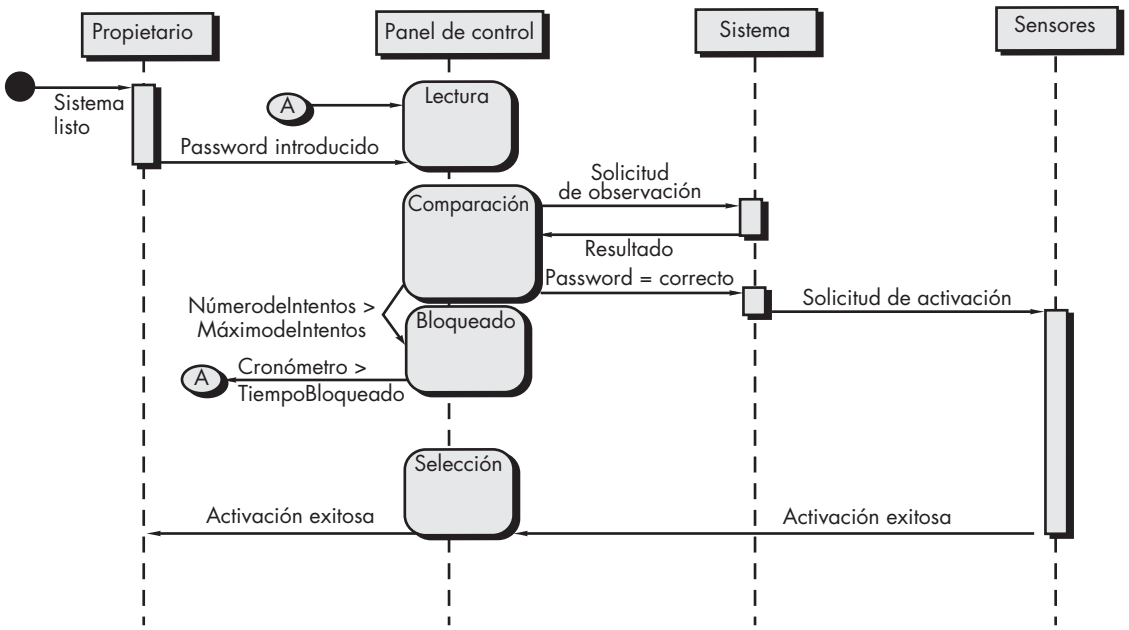


FIGURA 7.7
Diagrama de secuencia (parcial) para la función *CasaSegura*

ejemplo, la acción conectada con el evento *password introducido* (véase la figura 7.6) es una operación llamada *ValidarPassword ( )* que accede a un objeto **password** y realiza una comparación dígito por dígito para validar el password introducido.

Diagramas de secuencia. El segundo tipo de representación del comportamiento, llamado *diagrama de secuencia* en UML, indica la forma en la que los eventos provocan transiciones de un objeto a otro. Una vez identificados los objetos por medio del análisis del caso de uso, el modelador crea un diagrama de secuencia: representación del modo en el que los eventos causan el flujo de uno a otro como función del tiempo. En esencia, el diagrama de secuencia es una versión taquigráfica del caso de uso. Representa las clases password y los eventos que hacen que el comportamiento avance de una clase a otra.

PUNTO CLAVE
A diferencia de un diagrama de estado que representa el comportamiento sin fijarse en las clases involucradas, un diagrama de secuencia representa el comportamiento, describiendo la forma en la que las clases pasan de un estado a otro.

La figura 7.7 ilustra un diagrama parcial de secuencia para la función de seguridad de *CasaSegura*. Cada flecha representa un evento (derivado de un caso de uso) e indica la forma en la que éste canaliza el comportamiento entre los objetos de *CasaSegura*. El tiempo se mide en la dirección vertical (hacia abajo) y los rectángulos verticales angostos representan el que toma el procesamiento de una actividad. Los estados se presentan a lo largo de una línea de tiempo vertical.

El primer evento, *sistema listo*, se deriva del ambiente externo y canaliza el comportamiento al objeto **Propietario**. El propietario introduce un password. El evento *solicitud de observación* pasa al **Sistema**, que observa el password en una base de datos sencilla y devuelve un *resultado* (*encontrada o no encontrada*) a **PaneldeControl** (ahora en el estado de *comparación*). Un password válido da como resultado el evento *password = correcto* hacia **Sistema**, lo que activa a **Sensores** con un evento de *solicitud de activación*. Por último, el control pasa de nuevo al propietario con el evento *activación exitosa*.

Una vez que se ha desarrollado un diagrama de secuencia completo, todos los eventos que causan transiciones entre objetos del sistema se recopilan en un conjunto de eventos de entrada y de salida (desde un objeto). Esta información es útil en la generación de un diseño eficaz para el sistema que se va a construir.

HERRAMIENTAS DE SOFTWARE

Modelación de análisis generalizado en UML

Objetivo: Las herramientas de modelado del análisis dan la capacidad de desarrollar modelos basados en el escenario, en la clase y en el comportamiento con el uso de notación UML.

Mecánica: Las herramientas en esta categoría dan apoyo a toda la variedad de diagramas UML requeridos para construir un modelo del análisis (estas herramientas también apoyan el modelado del diseño). Además de los diagramas, las herramientas en esta categoría 1) hacen revisiones respecto de la consistencia y corrección para todos los diagramas UML, 2) proveen vínculos para producir el diseño y generar el código, y 3) construyen una base de datos que permite administrar y evaluar modelos UML grandes requeridos en sistemas complejos.

Herramientas representativas:[10]

Las herramientas siguientes apoyan toda la variedad de diagramas UML que se requieren para modelar el análisis:

ArgoUML es una herramienta de fuente abierta disponible en **argouml.tigris.org**.
Enterprise Architect, desarrollada por Sparx Systems (**www.sparxsystems.com.au**).
PowerDesigner, desarrollada por Sybase (**www.sybase.com**).
Rational Rose, desarrollada por IBM (Rational) (**www01.ibm.com/software/rational/**).
System Architect, desarrollada por Popkin Software (**www.popkin.com**).
UML Studio, desarrollada por Pragsoft Corporation (**www.pragsoft.com**).
Visio, desarrollada por Microsoft (**www.microsoft.com**).
Visual UML, desarrollada por Visual Object Modelers (**www.visualuml.com**).

7.4 PATRONES PARA EL MODELADO DE REQUERIMIENTOS

Los patrones de software son un mecanismo para capturar conocimiento del dominio, en forma que permita que vuelva a aplicarse cuando se encuentre un problema nuevo. En ciertos casos, el conocimiento del dominio se aplica a un nuevo problema dentro del mismo dominio de la aplicación. En otros, el conocimiento del dominio capturado por un patrón puede aplicarse por analogía a otro dominio de una aplicación diferente por completo.

El autor original de un patrón de análisis no "crea" el patrón, sino que lo *descubre* a medida que se realiza el trabajo de ingeniería de requerimientos. Una vez descubierto el patrón, se documenta describiendo "explícitamente el problema general al que es aplicable el patrón, la solución prescrita, las suposiciones y restricciones del uso del patrón en la práctica y, con frecuencia, alguna otra información sobre éste, como la motivación y las fuerzas que impulsan el empleo del patrón, el análisis de las ventajas y desventajas del mismo y referencias a algunos ejemplos conocidos de su empleo en aplicaciones prácticas" [Dev01].

En el capítulo 5 se presentó el concepto de patrones de análisis y se indicó que éstos representan una solución que con frecuencia incorpora una clase, función o comportamiento dentro del dominio de aplicación. El patrón vuelve a utilizarse cuando se hace el modelado de los requerimientos para una aplicación dentro del dominio.[11] Los patrones de análisis se guardan en un depósito para que los miembros del equipo de software usen herramientas de búsqueda para encontrarlos y volverlos a emplear. Una vez seleccionado un patrón apropiado, se integra en el modelo de requerimientos, haciendo referencia a su nombre.

7.4.1 Descubrimiento de patrones de análisis

El modelo de requerimientos está formado por una amplia variedad de elementos: basados en el escenario (casos de uso), orientados a datos (el modelo de datos), basados en clases, orientados al flujo y del comportamiento. Cada uno de estos elementos estudia el problema desde

10 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.
11 En el capítulo 12 se presenta un análisis a profundidad del uso de patrones durante el diseño del software.

una perspectiva diferente y da la oportunidad de descubrir patrones que tal vez suceden en un dominio de aplicación o por analogía en distintos dominios de aplicación.

El elemento más fundamental en la descripción de un modelo de requerimientos es el caso de uso. En el contexto de este análisis, un conjunto coherente de casos de uso sirve como base para descubrir uno o más patrones de análisis. Un *patrón de análisis semántico* (PAS) "es un patrón que describe un conjunto pequeño de casos de uso coherentes que describen a su vez una aplicación general" [Fer00].

Considere el siguiente caso de uso preliminar para el software que se requiere a fin de controlar y vigilar una cámara de visión real y un sensor de proximidad para un automóvil:

> **Caso de uso:** ***Vigilar el movimiento en reversa***
>
> **Descripción:** Cuando se coloca el vehículo en *reversa*, el software de control permite que se transmita un video a una pantalla que está en el tablero, desde una cámara colocada en la parte posterior. El software superpone varias líneas de orientación y distancia en la pantalla a fin de que el operador del auto mantenga la orientación cuando éste se mueve en reversa. El software de control también vigila un sensor de proximidad con el fin de determinar si un objeto se encuentra dentro de una distancia de 10 pies desde la parte trasera del carro. Esto frenará al vehículo de manera automática si el sensor de proximidad indica que hay un objeto a *x* pies de la defensa trasera, donde *x* se determina con base en la velocidad del automóvil.

Este caso de uso implica varias funciones que se mejorarían y elaborarían (en un conjunto coherente de casos de uso) durante la reunión para recabar y modelar los requerimientos. Sin importar cuánta elaboración se logre, los casos de uso sugieren un PAS sencillo pero con amplias aplicaciones (la vigilancia y control de sensores y actuadores de un sistema físico con base en software). En este caso, los "sensores" dan información en video sobre la proximidad. El "actuador" es el sistema de frenado del vehículo (que se invoca si hay un objeto muy cerca de éste). Pero en un caso más general, se descubre un patrón de aplicación muy amplio.

En muchos dominios distintos de aplicación, se requiere software para vigilar sensores y controlar actuadores físicos. Se concluye que podría usarse mucho un patrón de análisis que describa los requerimientos generales para esta capacidad. El patrón, llamado **Actuador-Sensor**, se aplicaría como parte del modelo de requerimientos para *CasaSegura* y se analiza en la sección 7.4.2, a continuación.

7.4.2 Ejemplo de patrón de requerimientos: Actuador-Sensor[12]

Uno de los requerimientos de la función de seguridad de *CasaSegura* es la capacidad de vigilar sensores de seguridad (por ejemplo, sensores de frenado, de incendio, de humo o contenido de CO, de agua, etc.). Las extensiones basadas en internet para *CasaSegura* requerirán la capacidad de controlar el movimiento (por ejemplo, apertura, acercamiento, etc.) de una cámara de seguridad dentro de una residencia. La implicación es que el software de *CasaSegura* debe manejar varios sensores y "actuadores" (como los mecanismos de control de las cámaras).

Konrad y Cheng [Kon02] sugieren un patrón llamado **Actuador-Sensor** que da una guía útil para modelar este requerimiento dentro del software de *CasaSegura*. A continuación se presenta una versión abreviada del patrón **Actuador-Sensor**, desarrollada originalmente para aplicaciones automotrices.

Nombre del patrón. **Actuador-Sensor**

Objetivo. Especifica distintas clases de sensores y actuadores en un sistema incrustado.

Motivación. Por lo general, los sistemas incrustados tienen varias clases de sensores y actuadores, conectados en forma directa o indirecta con una unidad de control. Aunque muchos de

12 Esta sección se adaptó de [Kon02] con permiso de los autores.

los sensores y actuadores se ven muy distintos, su comportamiento es lo bastante similar como para estructurarlos en un patrón. Éste ilustra la forma de especificar los sensores y actuadores para un sistema, incluso los atributos y operaciones. El patrón **Actuador-Sensor** usa un mecanismo para *jalar* (solicitud explícita de información) **SensoresPasivos** y otro mecanismo para *empujar* (emisión de información) los **SensoresActivos**.

Restricciones

- Cada sensor pasivo debe tener algún método para leer la entrada de un sensor y los atributos que representan al valor del sensor.
- Cada sensor activo debe tener capacidades para emitir mensajes actualizados cuando su valor cambie.
- Cada sensor activo debe enviar un *latido de vida*, mensaje de estado que se emite cada cierto tiempo para detectar fallas.
- Cada actuador debe tener un método para invocar la respuesta apropiada determinada por el **CálculodeComponente**.
- Cada sensor y actuador deben tener una función implementada para revisar su propio estado de operación.
- Cada sensor y actuador debe ser capaz de someter a prueba la validez de los valores recibidos o enviados y fijar su estado de operación si los valores se encuentran fuera de las especificaciones.

Aplicabilidad. Es útil en cualquier sistema en el que haya varios sensores y actuadores.

Estructura. En la figura 7.8 se presenta un diagrama de clase UML para el patrón **Actuador-Sensor**. **Actuador**, **SensorPasivo** y **SensorActivo** son clases abstractas y están escritas con letra cursiva. En este patrón hay cuatro tipos diferentes de sensores y actuadores. Las clases **Booleano**, **Entero** y **Real** representan los tipos más comunes de sensores y actuadores. Las clases complejas de éstos son aquellas que usan valores que no se representan con facilidad en términos de tipos de datos primitivos, tales como los de un radar. No obstante, estos equipos

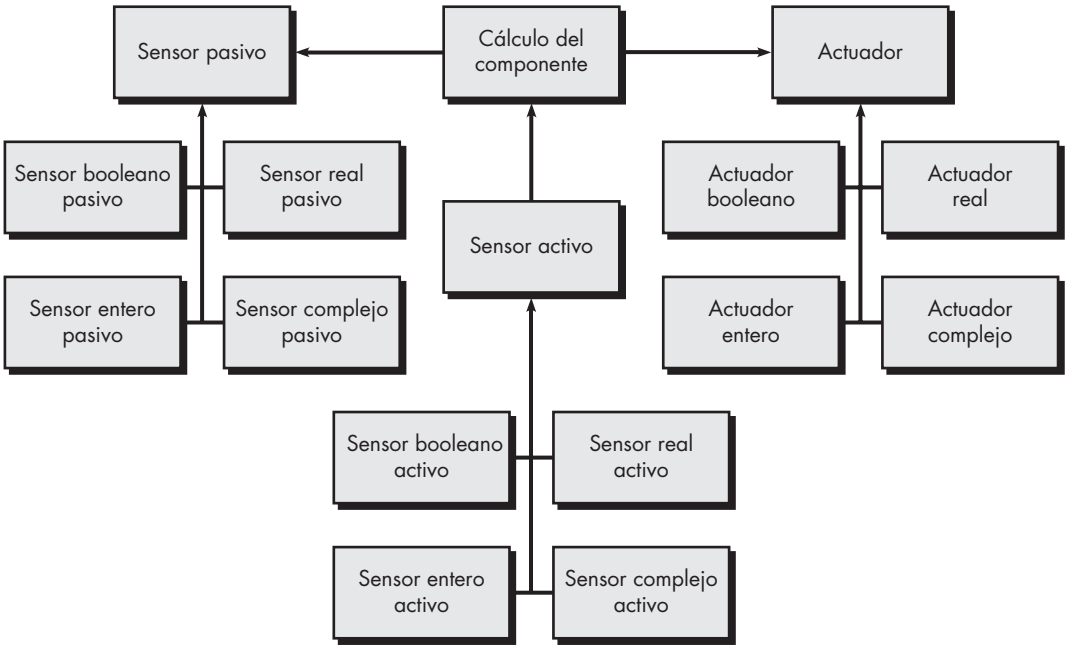
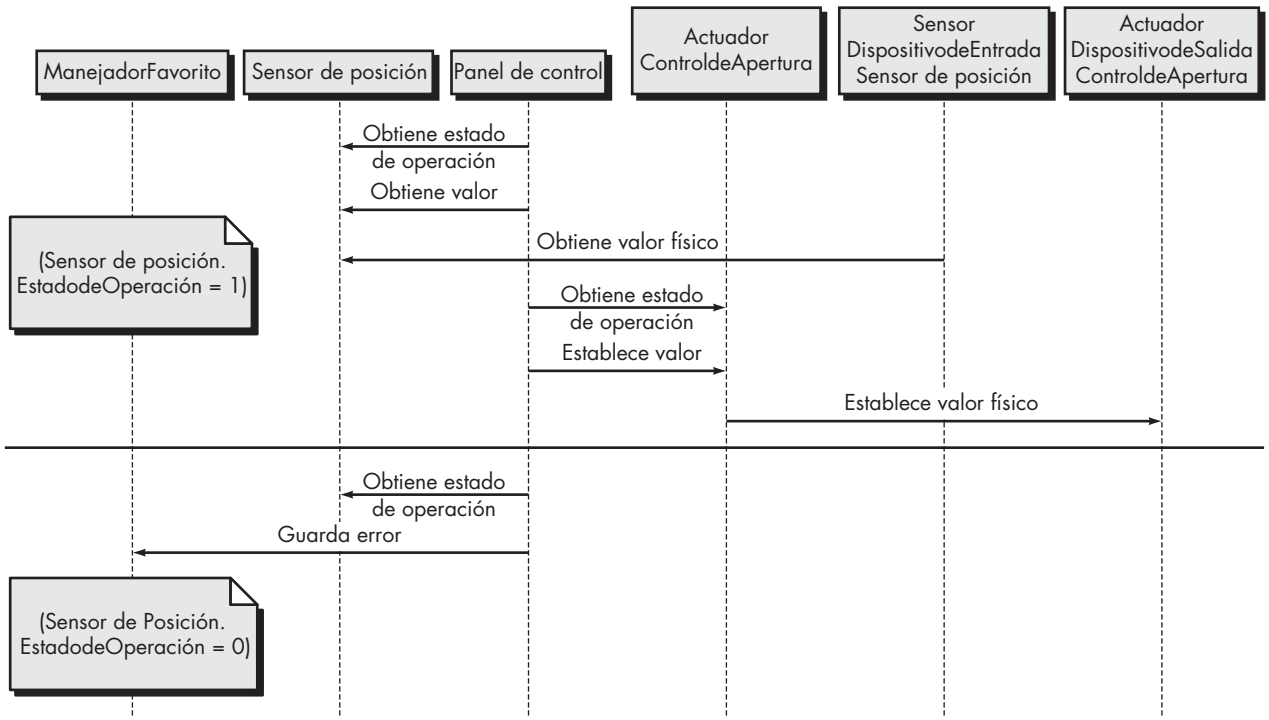


FIGURA 7.8

Diagrama de secuencia UML para el patrón Actuador-Sensor. ***Fuente:*** **Adaptado de [Kon02], con permiso.**

FIGURA 7.9 Diagrama de clase UML para el patrón Actuador-Sensor. ***Fuente:*** **Reimpreso de [Kon02], con permiso.**



deben heredar la interfaz de las clases abstractas, ya que deben tener funciones básicas, tales como consultar los estados de operación.

Comportamiento: La figura 7.9 presenta un diagrama de secuencia UML para un ejemplo de patrón **Actuador-Sensor** según podría aplicarse a la función de *CasaSegura* que controla el posicionamiento (como la apertura y el acercamiento) de una cámara de seguridad. Aquí, el **PaneldeControl**[13] consulta un sensor (uno de posición pasiva) y un actuador (control de apertura) para comprobar el estado de operación con fines de diagnóstico antes de leer o establecer un valor. Los mensajes *Establecer un Valor Físico* y *Obtener un Valor Físico* no son mensajes entre objetos. En vez de ello, describen la interacción entre los dispositivos físicos del sistema y sus contrapartes de software. En la parte inferior del diagrama, bajo la línea horizontal, el **SensordePosición** reporta que el estado de operación es igual a cero. Entonces, el **CálculodeComponente** (representado como **PaneldeControl**) envía el código de error para una falla de posición de un sensor al **ManejadordeFallas**, que decidirá cómo afecta este error al sistema y qué acciones se requieren. Obtiene los datos de los sensores y calcula la respuesta requerida por parte de los actuadores.

Participantes. Esta sección de la descripción de patrones "clasifica las clases u objetos incluidos en el patrón de requerimientos" [Kon02] y describe las responsabilidades de cada clase u objeto (véase la figura 7.8). A continuación se presenta una lista abreviada:

- **Resumen de SensorPasivo:** Define una interfaz para los sensores pasivos.
- **SensorBooleanoPasivo:** Define los sensores booleanos pasivos.
- **SensorEnteroPasivo:** Define los sensores enteros pasivos.

13 El patrón original usa la frase general **CálculodeComponente**.

- **SensorRealPasivo:** Define los sensores reales pasivos.
- **Resumen de SensorActivo:** Define una interfaz para los sensores activos.
- **SensorBooleanoActivo:** Define los sensores booleanos activos.
- **SensorEnteroActivo:** Define los sensores enteros activos.
- **SensorRealActivo:** Define los sensores reales activos.
- **Resumen de actuador:** Define una interfaz para los actuadores.
- **ActuadorBooleano:** Define los actuadores booleanos.
- **ActuadorEntero:** Define los actuadores enteros.
- **ActuadorReal:** Define los actuadores reales.
- **CálculodeComponente:** Parte central del controlador; obtiene los datos de los sensores y calcula la respuesta requerida para los actuadores.
- **SensorComplejoActivo:** Los sensores complejos activos tienen la funcionalidad básica de la clase **SensorActivo**, pero es necesario especificar métodos y atributos adicionales más elaborados.
- **SensorComplejoPasivo:** Los sensores complejos pasivos tienen la funcionalidad básica de la clase abstracta **SensorPasivo**, pero se necesita especificar métodos y atributos adicionales más elaborados.
- **ActuadorComplejo:** Los actuadores complejos también tienen la funcionalidad básica de la clase abstracta **Actuador**, pero se requiere especificar métodos y atributos adicionales más elaborados.

Colaboraciones. Esta sección describe cómo interactúan los objetos y clases entre sí, y cómo efectúa cada uno sus responsabilidades.

- Cuando **CálculodeComponente** necesita actualizar el valor de un **SensorPasivo**, consulta a los sensores y solicita el valor enviando el mensaje apropiado.
- Los **SensoresActivos** no son consultados. Inician la transmisión de los valores del sensor a la unidad de cálculo, con el uso del método apropiado para establecer el valor en **CálculodeComponente**. Durante un tiempo especificado, envían un latido de vida al menos una vez con el fin de actualizar sus parámetros de tiempo con el reloj del sistema.
- Cuando **CálculodeComponente** necesita establecer el valor de un actuador, envía el valor a éste.
- **CálculodeComponente** consulta y establece el estado de operación de los sensores y actuadores por medio de los métodos apropiados. Si un estado de operación es cero, entonces se envía el error al **ManejadordeFallas**, clase que contiene métodos para manejar mensajes de error, tales como reiniciar un mecanismo más elaborado de recuperación o un dispositivo de respaldo. Si no es posible la recuperación, entonces el sistema sólo usa el último valor conocido para el sensor o uno preestablecido.
- Los **SensoresActivos** ofrecen métodos para agregar o retirar los evaluadores o evalúan rangos de los componentes que quieren que reciban los mensajes en caso de un cambio de valor.

Consecuencias

1. Las clases sensor y actuador tienen una interfaz común.
2. Sólo puede accederse a los atributos de clase a través de mensajes y la clase decide si se aceptan o no. Por ejemplo, si se establece el valor de un actuador por arriba del

máximo, entonces la clase actuador tal vez no acepte el mensaje, o quizá emplee un valor máximo preestablecido.

3. La complejidad del sistema es potencialmente reducida debido a la uniformidad de las interfaces para los actuadores y sensores.

La descripción del patrón de requerimientos también da referencias acerca de otros requerimientos y patrones de diseño relacionados.

7.5 Modelado de requerimientos para webapps[14]

Es frecuente que los desarrolladores de web manifiesten escepticismo cuando se plantea la idea del análisis de los requerimientos para webapps. Acostumbran decir: "después de todo, el proceso de desarrollo en web debe ser ágil y el análisis toma tiempo. Nos hará ser lentos justo cuando necesitemos diseñar y construir la *webapp*".

El análisis de los requerimientos lleva tiempo, pero resolver el problema equivocado toma aún más tiempo. La pregunta que debe responder todo desarrollador en web es sencilla: ¿estás seguro de que entiendes los requerimientos del problema? Si la respuesta es un "sí" inequívoco, entonces tal vez sea posible omitir el modelado de los requerimientos, pero si la respuesta es "no", entonces ésta debe llevarse a cabo.

7.5.1 ¿Cuánto análisis es suficiente?

El grado en el que se profundice en el modelado de los requerimientos para las *webapps* depende de los factores siguientes:

- Tamaño y complejidad del incremento de la *webapp*.
- Número de participantes (el análisis ayuda a identificar los requerimientos conflictivos que provienen de distintas fuentes).
- Tamaño del equipo de la *webapp*.
- Grado en el que los miembros del equipo han trabajado juntos antes (el análisis ayuda a desarrollar una comprensión común del proyecto).
- Medida en la que el éxito de la organización depende directamente del éxito de la *webapp*.

El inverso de los puntos anteriores es que a medida que el proyecto se hace más chico, que el número de participantes disminuye, que el equipo de desarrollo es más cohesivo y que la aplicación es menos crítica, es razonable aplicar un enfoque más ligero para el análisis.

Aunque es una buena idea analizar el problema *antes* de que comience el diseño, no es verdad que *todo* el análisis deba preceder a *todo* el diseño. En realidad, el diseño de una parte específica de la *webapp* sólo demanda un análisis de los requerimientos que afectan a sólo esa parte de la *webapp*. Como un ejemplo proveniente de *CasaSegura*, podría diseñarse con validez la estética general del sitio web (formatos, colores, etc.) sin tener que analizar los requerimientos funcionales de las capacidades de comercio electrónico. Sólo se necesita analizar aquella parte del problema que sea relevante para el trabajo de diseño del incremento que se va a entregar.

7.5.2 Entrada del modelado de los requerimientos

En el capítulo 2 se analizó una versión ágil del proceso de software general que puede aplicarse cuando se hace la ingeniería de las *webapps*. El proceso incorpora una actividad de comunica-

14 Esta sección se adaptó de Pressman y Lowe [Pre08], con permiso.

ción que identifica a los participantes y las categorías de usuario, el contexto del negocio, las metas definidas de información y aplicación, requerimientos generales de *webapps* y los escenarios de uso, información que se convierte en la entrada del modelado de los requerimientos. Esta información se representa en forma de descripciones hechas en lenguaje natural, a grandes rasgos, en bosquejos y otras representaciones no formales.

El análisis toma esta información, la estructura con el empleo de un esquema de representación definido formalmente (donde sea apropiado) y luego produce como salida modelos más rigurosos. El modelo de requerimientos brinda una indicación detallada de la verdadera estructura del problema y da una perspectiva de la forma de la solución.

En el capítulo 6 se introdujo la función **AVC-MVC** (vigilancia con cámaras). En ese momento, esta función parecía relativamente clara y se describió con cierto detalle como parte del caso de uso (véase la sección 6.2.1). Sin embargo, la revisión del caso de uso quizá revele información oculta, ambigua o poco clara.

Algunos aspectos de esta información faltante emergerían de manera natural durante el diseño. Los ejemplos quizá incluyan el formato específico de los botones de función, su aspecto y percepción estética, el tamaño de las vistas instantáneas, la colocación del ángulo de las cámaras y el plano de la casa, o incluso minucias tales como las longitudes máxima y mínima de las claves. Algunos de estos aspectos son decisiones de diseño (como el aspecto de los botones) y otros son requerimientos (como la longitud de las claves) que no influyen de manera fundamental en los primeros trabajos de diseño.

Pero cierta información faltante sí podría influir en el diseño general y se relaciona más con la comprensión real de los requerimientos. Por ejemplo:

P_1: ¿Cuál es la resolución del video de salida que dan las cámaras de *CasaSegura*?

P_2: ¿Qué ocurre si se encuentra una condición de alarma mientras la cámara está siendo vigilada?

P_3: ¿Cómo maneja el sistema las cámaras con vistas panorámicas y de acercamiento?

P_4: ¿Qué información debe darse junto con la vista de la cámara (por ejemplo, ubicación, fecha y hora, último acceso, etcétera)?

Ninguna de estas preguntas fue identificada o considerada en el desarrollo inicial del caso de uso; no obstante, las respuestas podrían tener un efecto significativo en los diferentes aspectos del diseño.

Por tanto, es razonable concluir que aunque la actividad de comunicación provea un buen fundamento para entender, el análisis de los requerimientos mejora este entendimiento al dar una interpretación adicional. Como la estructura del problema se delinea como parte del modelo de requerimientos, invariablemente surgen preguntas. Son éstas las que llenan los huecos y, en ciertos casos, en realidad ayudan a encontrarlos.

En resumen, la información obtenida durante la actividad de comunicación será la entrada del modelo de los requerimientos, cualquiera que sea, desde un correo electrónico informal hasta un proyecto detallado con escenarios de uso exhaustivos y especificaciones del producto.

7.5.3 Salida del modelado de los requerimientos

El análisis de los requerimientos provee un mecanismo disciplinado para representar y evaluar el contenido y funcionamiento de las *webapp*, los modos de interacción que hallarán los usuarios y el ambiente e infraestructura en las que reside la *webapp*.

Cada una de estas características se representa como un conjunto de modelos que permiten que los requerimientos de la *webapp* sean analizados en forma estructurada. Si bien los modelos específicos dependen en gran medida de la naturaleza de la *webapp*, hay cinco clases principales de ellos: