



El Lenguaje Unificado de Modelado Guía del usuario

SEGUNDA EDICIÓN

Grady Booch
James Rumbaugh
Ivar Jacobson

Traducción y Revisión Técnica

Jesús J. García Molina

José Sáez Martínez

Departamento de Informática y Sistemas

Universidad de Murcia

▲Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • París • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico Ciry

Datos de catalogación bibliográfica

Grady Booch; James Rumbaugh; Ivar Jacobson
El Lenguaje Unificado de Modelado
PEARSON EDUCACIÓN, S.A. Madrid, 2006
ISBN 10: 84-7829-076-1
ISBN 13: 978-84-7829-076-5
Materia: Informática, 0004.4
Formato: 195 x 250 mm Páginas: ???

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (arts. 270 y sigs. Código Penal).

DERECHOS RESERVADOS
© 2006, PEARSON EDUCACIÓN, S.A.
Ribera del Loira, 28
28042 Madrid (España)

Grady Booch; James Rumbaugh; Ivar Jacobson
El Lenguaje Unificado de Modelado

ISBN 10: 84-7829-076-1
ISBN 13: 978-84-7829-076-5
Depósito Legal: M. 26.458-2006
ADDISON WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Authorized translation from the English language edition, entitled UNIFIED MODELING LANGUAGE USER GUIDE, THE, 2ND Edition by BOOCH, GRADY; RUMBAUGH, JAMES; JACOBSON, IVAR, published by Pearson Education Inc, publishing as Addison Wesley, Copyright © 2005

Equipo editorial:

Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

Equipo de producción:

Director: José Antonio Clares

Diseño de cubierta: Equipo de diseño de PEARSON EDUCACIÓN, S. A.

Composición: Opción K

Impreso por: Fernández Ciudad, S. L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

*A mi querida esposa, Jan y a mi ahijada,
Elyse, quienes me llenan de felicidad.*

CONTENIDO

	Pág.
Prólogo	xiii
Parte 1 Introducción	1
Capítulo 1 Por qué modelamos	3
La importancia de modelar	4
Principios del modelado	8
Modelado orientado a objetos	11
Capítulo 2 Presentación de UML	15
Visión general de UML	16
Un modelo conceptual de UML	19
Arquitectura	35
Ciclo de vida del desarrollo de software	38
Capítulo 3 ¡Hola, mundo!	41
Abstracciones clave	42
Mecanismos	45
Artefactos	47
Parte 2 Modelado estructural básico	49
Capítulo 4 Clases	51
Introducción	51
Términos y conceptos	53
Técnicas comunes de modelado	59
Sugerencias y consejos	64

	Pág.		Pág.
Capítulo 5 Relaciones	65	Capítulo 11 Interfaces, tipos y roles	161
Introducción	66	Introducción	161
Términos y conceptos	67	Términos y conceptos	163
Técnicas comunes de modelado	74	Técnicas comunes de modelado	167
Sugerencias y consejos	79	Sugerencias y consejos	171
Capítulo 6 Mecanismos comunes	81	Capítulo 12 Paquetes	173
Introducción	82	Introducción	174
Términos y conceptos	83	Términos y conceptos	175
Técnicas comunes de modelado	90	Técnicas comunes de modelado	181
Sugerencias y consejos	94	Sugerencias y consejos	184
Capítulo 7 Diagramas	97	Capítulo 13 Instancias	187
Introducción	98	Introducción	187
Términos y conceptos	99	Términos y conceptos	189
Técnicas comunes de modelado	105	Técnicas comunes de modelado	194
Sugerencias y consejos	110	Sugerencias y consejos	196
Capítulo 8 Diagramas de clases	113	Capítulo 14 Diagramas de objetos	197
Introducción	113	Introducción	197
Términos y conceptos	115	Términos y conceptos	199
Técnicas comunes de modelado	116	Técnicas comunes de modelado	201
Sugerencias y consejos	124	Sugerencias y consejos	204
Parte 3 Modelado estructural avanzado	125	Capítulo 15 Componentes	207
Capítulo 9 Características avanzadas de las clases	127	Introducción	207
Introducción	127	Términos y conceptos	208
Términos y conceptos	128	Técnicas comunes de modelado	218
Técnicas comunes de modelado	140	Sugerencias y consejos	221
Sugerencias y consejos	142	Parte 4 Modelado básico del comportamiento	223
Capítulo 10 Características avanzadas de las relaciones	143	Capítulo 16 Interacciones	225
Introducción	144	Introducción	226
Términos y conceptos	145	Términos y conceptos	227
Técnicas comunes de modelado	159	Técnicas comunes de modelado	238
Sugerencias y consejos	160	Sugerencias y consejos	240

	Pág.
Capítulo 17 Casos de uso	243
Introducción	244
Términos y conceptos	246
Técnicas comunes de modelado	255
Sugerencias y consejos	257
Capítulo 18 Diagramas de casos de uso	259
Introducción	259
Términos y conceptos	261
Técnicas comunes de modelado	262
Sugerencias y consejos	268
Capítulo 19 Diagramas de interacción	271
Introducción	272
Términos y conceptos	273
Técnicas comunes de modelado	284
Sugerencias y consejos	289
Capítulo 20 Diagramas de actividades	291
Introducción	292
Términos y conceptos	293
Técnicas comunes de modelado	304
Sugerencias y consejos	309
Parte 5 Modelado avanzado del comportamiento	311
Capítulo 21 Eventos y señales	313
Introducción	313
Términos y conceptos	314
Técnicas comunes de modelado	320
Sugerencias y consejos	323
Capítulo 22 Máquinas de estados	325
Introducción	326
Términos y conceptos	328
Técnicas comunes de modelado	346
Sugerencias y consejos	349

	Pág.
Capítulo 23 Procesos e hilos	351
Introducción	352
Términos y conceptos	353
Técnicas comunes de modelado	359
Sugerencias y consejos	362
Capítulo 24 Tiempo y escacio	365
Introducción	365
Términos y conceptos	367
Técnicas comunes de modelado	369
Sugerencias y consejos	372
Capítulo 25 Diagramas de estados	373
Introducción	374
Términos y conceptos	375
Técnicas comunes de modelado	378
Sugerencias y consejos	382
Parte 6 Modelado arquitectónico	385
Capítulo 26 Artefactos	387
Introducción	387
Términos y conceptos	389
Técnicas comunes de modelado	392
Sugerencias y consejos	397
Capítulo 27 Despliegue	399
Introducción	399
Términos y conceptos	400
Técnicas comunes de modelado	403
Sugerencias y consejos	406
Capítulo 28 Colaboraciones	407
Introducción	407
Términos y conceptos	409
Técnicas comunes de modelado	414
Sugerencias y consejos	420

	<u>Pág.</u>
Capítulo 29 Patrones y frameworks	423
Introducción	423
Términos y conceptos	425
Técnicas comunes de modelado	429
Sugerencias y consejos	435
Capítulo 30 Diagramas de artefactos	437
Introducción	437
Términos y conceptos	438
Técnicas comunes de modelado	440
Sugerencias y consejos	450
Capítulo 31 Diagramas de despliegue	453
Introducción	453
Términos y conceptos	455
Técnicas comunes de modelado	457
Sugerencias y consejos	463
Capítulo 32 Sistemas y modelos	465
Introducción	465
Términos y conceptos	467
Técnicas comunes de modelado	470
Sugerencias y consejos	473
Parte 7 Cierre	475
Capítulo 33 Utilización de UML	477
Transición a UML	477
Adónde ir a continuación	479
Apéndice A Notación UML	481
Apéndice B El proceso unificado de rational	487
Glosario	497
Índice	511

PRÓLOGO

El Lenguaje Unificado de Modelado (UML, *Unified Modeling Language*) es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software. UML proporciona una forma estándar de representar los planos de un sistema, y comprende tanto elementos conceptuales, como los procesos del negocio y las funciones del sistema, cuantos elementos concretos, como las clases escritas en un lenguaje de programación específico, esquemas de bases de datos y componentes software reutilizables.

Este libro enseña cómo utilizar UML de forma efectiva.

Este libro cubre la versión 2.0 de UML.

Objetivos

En este libro:

- Aprenderá qué es UML y qué no es, y por qué UML es relevante para el proceso de desarrollar sistemas con gran cantidad de software.
- Dominará el vocabulario, las reglas y las construcciones específicas de UML y, en general, aprenderá a “hablar” el lenguaje de forma efectiva.
- Comprenderá cómo utilizar UML para resolver varios problemas de modelado frecuentes.

Este libro proporciona una referencia del uso de características específicas de UML. Sin embargo, no tiene la intención de ser un manual de referencia completo de UML; éste es el tema de otro libro, *The Unified Modeling Language Reference Manual, Second Edition* (Rumbaugh, Jacobson, Booch, Addison-Wesley, 2005).

Este libro describe un proceso de desarrollo para utilizar con UML. Sin embargo, no tiene la intención de proporcionar una referencia completa de ese proceso;

éste es el tema de otro libro, *The Unified Software Development Process* (Jacobson, Booch, Rumbaugh, Addison-Wesley, 1999).

Por último, este libro proporciona sugerencias y consejos sobre cómo utilizar UML para resolver varios problemas de modelado comunes, pero no enseña cómo modelar. En este sentido es parecido a una guía de usuario de un lenguaje de programación, que enseña cómo utilizar el lenguaje pero no enseña a programar.

Destinatarios

UML puede ser utilizado por cualquier persona involucrada en la producción, el despliegue y el mantenimiento de software. Este libro se dirige principalmente a los miembros del equipo de desarrollo que crean modelos UML. Sin embargo, también es apropiado para aquellos que los leen, colaborando en la comprensión, construcción, pruebas y distribución de un sistema con gran cantidad de software. Aunque esto incluye casi todos los papeles en una organización de desarrollo de software, este libro es especialmente importante para los analistas y los usuarios finales (que especifican la estructura y el comportamiento requerido de un sistema), los arquitectos (que diseñan sistemas que satisfacen esos requisitos), los desarrolladores (que convierten esas arquitecturas en código ejecutable), el personal de control de calidad (que verifica y valida la estructura y el comportamiento del sistema), los encargados de las bibliotecas de software (que crean y catalogan los componentes) y los jefes del programa y del proyecto (que generalmente luchan contra el caos, proporcionan liderazgo y orientación, y orquestan los recursos necesarios para desarrollar un sistema que alcance el éxito).

Este libro supone un conocimiento básico de los conceptos orientados a objetos. La experiencia con algún lenguaje de programación orientado a objetos será útil aunque no es necesaria.

Cómo Utilizar Este Libro

Para el desarrollador que se acerca a UML por primera vez, lo mejor es leer este libro de forma secuencial. Debería prestar una atención especial al Capítulo 2, que presenta un modelo conceptual de UML. Todos los capítulos se estructuran de forma que cada uno se basa en el contenido de los anteriores, dando lugar a una progresión lineal.

El desarrollador experimentado, que busca respuestas a los problemas comunes de modelado que surgen al utilizar UML, puede leer el libro en cualquier orden.

Debería prestar una atención especial a los problemas comunes de modelado que aparecen en cada capítulo.

Organización y Características Especiales

Este libro se organiza en siete partes:

- Sección 1 Introducción
- Sección 2 Modelado Estructural Básico
- Sección 3 Modelado Estructural Avanzado
- Sección 4 Modelado de Comportamiento Básico
- Sección 5 Modelado de Comportamiento Avanzado
- Sección 6 Modelado de la Arquitectura
- Sección 7 Cierre

Este libro contiene dos apéndices: un resumen de la notación UML y un resumen del Proceso Unificado de Rational. También se incluye un glosario de términos básicos. Hay un índice al final.

Cada capítulo trata el uso de una característica específica de UML, y la mayoría se organiza en las cuatro siguientes secciones:

1. Introducción
2. Términos y Conceptos
3. Técnicas Comunes de Modelado
4. Sugerencias y Consejos

La tercera sección presenta y resuelve un conjunto de problemas de modelado comunes. Para facilitar el que se pueda ojear el libro en busca de estos casos de uso de UML, cada problema se identifica por un título, como en el siguiente ejemplo.

Modelado de Patrones Arquitectónicos

Cada capítulo comienza con un resumen de las características que estudia, como en el siguiente ejemplo.

En este capítulo

- Objetos activos, procesos e hilos.
- Modelado de múltiples flujos de control.
- Modelado de la comunicación entre procesos.
- Construcción de abstracciones con hilos seguros (*thread-safe*).

Asimismo, los comentarios explicativos y guías generales se dan aparte en forma de notas, como se muestra en el siguiente ejemplo.

Nota: Las operaciones abstractas corresponden a lo que C++ llama operaciones virtuales puras; las operaciones hoja en UML corresponden a las operaciones no virtuales de C++.

Los componentes se discuten en el Capítulo 25.

UML tiene una semántica muy rica. Por lo tanto, la presentación de una característica puede involucrar a otras de forma natural. En tales casos, se proporcionan referencias cruzadas en el margen, como en esta página.

En las figuras se utilizan palabras destacadas en gris para distinguir el texto que explica un modelo del texto que forma parte del propio modelo. El código se distingue mostrándolo en una fuente de anchura fija, como en este ejemplo.

Agradecimientos. Los autores desean agradecer a Bruce Douglas, Per Krol y Joaquin Miller su apoyo en la revisión del manuscrito de la segunda edición.

Una Breve Historia de UML

Normalmente se reconoce a Simula-67 como primer lenguaje orientado a objetos, desarrollado por Dahl y Nygaard en Noruega en 1967. Este lenguaje nunca tuvo muchos seguidores, pero sus conceptos sirvieron de gran inspiración a varios lenguajes posteriores. Smalltalk se popularizó a principios de los 80, y enseguida siguieron otros lenguajes como Objective C, C++ y Eiffel a finales de los 80. Los lenguajes de modelado orientados a objetos aparecieron en la década de los 80, mientras los metodólogos, enfrentados a los nuevos lenguajes de programación orientados a objetos y a unas aplicaciones cada vez más complejas, empezaron a experimentar con enfoques alternativos al análisis y al diseño. El número de métodos orientados a objetos se incrementó de menos de 10 a más de 50 durante el período entre 1989 y 1994. Muchos usuarios de estos métodos tenían problemas para encontrar un lenguaje de modelado que cubriera sus necesidades completamente, alimentando de esta forma la llamada guerra de métodos. Unos pocos métodos empezaron a ganar importancia, entre ellos el método de Booch, el método OOSE (*Object-Oriented Software Engineering*, Ingeniería del Software Orientada a Objetos) de Jacobson, y el método OMT (*Object Modeling Technique*, Técnica de Modelado de Objetos) de Rumbaugh. Otros métodos importantes fueron Fusion, Shlaer-Mellor y Coad-Yourdon. Cada uno de éstos era un método completo, aunque todos tenían sus puntos fuertes y sus debilidades. En pocas palabras, el método de Booch era particularmente

expresivo durante las fases de diseño y construcción de los proyectos; OOSE proporcionaba un soporte excelente para los casos de uso como forma de dirigir la captura de requisitos, el análisis y el diseño de alto nivel, y OMT era útil principalmente para el análisis y para los sistemas de información con gran cantidad de datos.

Una masa crítica de ideas comenzó a formarse en la primera mitad de los 90, cuando Grady Booch (Rational Software Corporation), James Rumbaugh (General Electric), Ivar Jacobson (Objectory) y otros empezaron a adoptar ideas de los otros métodos, los cuales habían sido reconocidos unánimemente como los tres principales métodos orientados a objetos a nivel mundial. Como creadores principales de los métodos de Booch, OOSE y OMT, nos sentimos motivados a crear un lenguaje unificado de modelado por tres razones. En primer lugar, cada uno de nuestros métodos ya estaba evolucionando independientemente hacia los otros dos. Tenía sentido continuar esa evolución de forma conjunta en vez de hacerlo por separado, eliminando la posibilidad de cualquier diferencia gratuita e innecesaria que confundiría aún más a los usuarios. En segundo lugar, al unificar nuestros métodos, podríamos proporcionar cierta estabilidad al mercado orientado a objetos, haciendo posible que los proyectos se asentaran sobre un lenguaje de modelado maduro y permitiendo a los constructores de herramientas que se centraran en proporcionar más características útiles. En tercer lugar, esperábamos que nuestra colaboración introduciría mejoras en los tres métodos anteriores, ayudándonos a partir de las lecciones aprendidas y a tratar problemas que ninguno de nuestros métodos había manejado bien anteriormente.

Cuando comenzamos con la unificación, establecimos tres metas para nuestro trabajo:

1. Modelar sistemas, desde el concepto hasta los artefactos ejecutables, utilizando técnicas orientadas a objetos
2. Tratar las cuestiones relacionadas con el tamaño inherentes a los sistemas complejos y críticos
3. Crear un lenguaje de modelado utilizable tanto por las personas como por las máquinas

Desarrollar un lenguaje para que se utilice en el análisis y el diseño orientados a objetos no es muy diferente de diseñar un lenguaje de programación. En primer lugar, teníamos que acotar el problema: ¿Debería incluir el lenguaje la especificación de requisitos? ¿Debería ser el lenguaje lo suficientemente potente para permitir programación visual? En segundo lugar, teníamos que conseguir un equilibrio entre expresividad y simplicidad. Un lenguaje demasiado simple

limitaría demasiado el abanico de problemas que se podrían resolver; un lenguaje demasiado complejo podría agobiar a las personas que lo utilizaran. En el caso de la unificación de métodos existentes, también debíamos tener en cuenta las bases que se habían sentado. Si se hacían demasiados cambios, confundiríamos a los usuarios existentes; si nos resistíamos a hacer avanzar el lenguaje, perderíamos la oportunidad de involucrar a un conjunto mucho más amplio de usuarios y de hacer un lenguaje más sencillo. La definición de UML procura conseguir el mejor equilibrio en cada una de estas áreas.

El esfuerzo para la definición de UML comenzó en octubre de 1994, cuando Rumbaugh se unió a Booch en Rational. El objetivo inicial de nuestro proyecto fue la unificación de los métodos de Booch y OMT. El borrador de la versión 0.8 del Método Unificado (como fue llamado entonces) se publicó en octubre de 1995. Por esa época, Jacobson se unió a Rational y el alcance del proyecto UML se amplió para incorporar OOSE. Nuestros esfuerzos se plasmaron en los documentos de la versión 0.9 en junio de 1996. Durante 1996, pedimos y recibimos realimentación de la comunidad internacional relacionada con la ingeniería del software. Durante este tiempo, también se vio claro que muchas organizaciones de software veían a UML como un punto estratégico de su negocio. Establecimos un consorcio de UML, con varias organizaciones que querían dedicar recursos para trabajar hacia una definición fuerte y completa de UML.

Las organizaciones que contribuyeron a la definición 1.0 de UML fueron Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments y Unisys. Esta colaboración produjo UML 1.0, un lenguaje de modelado bien definido, expresivo, potente y aplicable a un amplio espectro de dominios de problemas. Mary Loomis fue decisiva para convencer al Object Management Group (OMG) de que emitiera una solicitud de propuestas (RFP, *Request For Proposals*) para un lenguaje de modelado estándar. UML 1.0 se ofreció para su estandarización al Object Management Group (OMG) en enero de 1997, en respuesta a su solicitud.

Entre enero y julio de 1997, el grupo inicial de colaboradores se amplió para incluir prácticamente a todas las demás organizaciones que habían enviado alguna propuesta o habían contribuido en alguna medida en las respuestas iniciales al QMG, incluyendo a Andersen Consulting, Ericsson, ObjectTime Limited, Platinum Technology, PTech, Reich Technologies, Softeam, Sterling Software y Taskon. Se formó un grupo de trabajo para la semántica, liderado por Cris Kobryn de MCI Systemhouse y coordinado por Ed Eykholt de Rational, para formalizar la especificación de UML y para integrar UML con otros esfuerzos de estandarización. Una versión revisada de UML (la versión 1.1) se ofreció al

OMG para su estandarización en julio de 1997. En septiembre de 1997, esta versión fue aceptada por la OMG Analysis and Design Task Force (ADTK) y el OMG Architecture Board y se sometió al voto de todos los miembros del OMG. UML 1.1 fue adoptado por el OMG el 14 de noviembre de 1997.

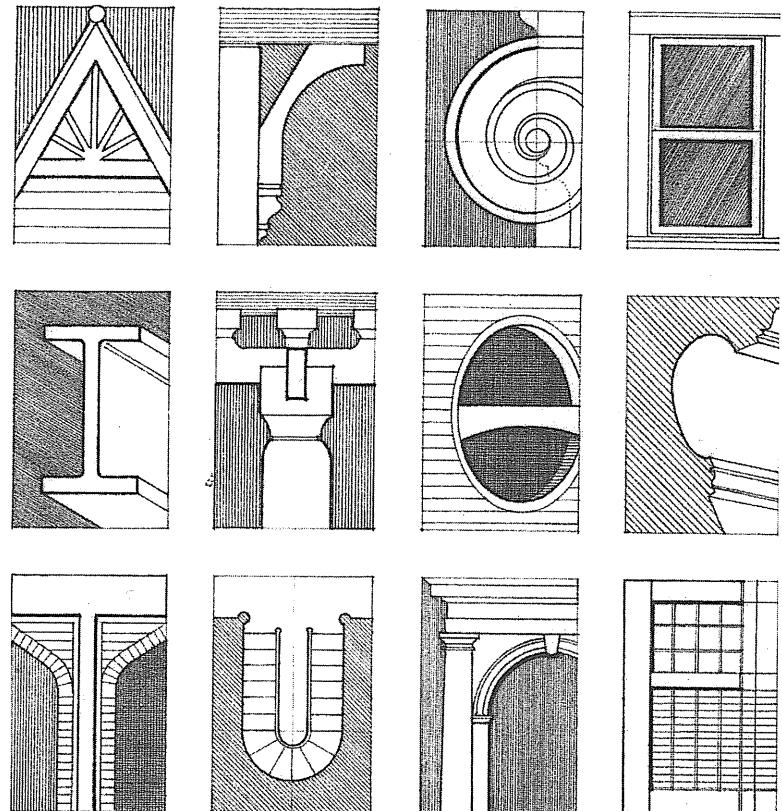
Durante varios años, el mantenimiento de UML fue asumido por la OMG Revision Task Force, que creó las versiones 1.3, 1.4 y 1.5. Del año 2000 al 2003, un conjunto nuevo y ampliado de colaboradores creó una especificación actualizada de UML, versión 2.0. Esta versión fue revisada durante un año por una Finalization Task Force (FTF) liderada por Bran Selic, de IBM, y el OMG adoptó la versión oficial de UML 2.0 a principios de 2005. UML 2.0 es una revisión importante de UML 1, e incluye muchas características adicionales. Además, se hicieron muchos cambios a los elementos previos a partir de la experiencia con la versión anterior. Los documentos de la especificación actual de UML se encuentran en la página web del OMG en www.omg.org.

UML es el fruto del trabajo de muchas personas, y las ideas que hay en él provienen de un amplio espectro de trabajos anteriores. Sería un gran proyecto de investigación histórica reconstruir una lista completa de fuentes, e incluso sería más difícil identificar a todos aquellos que han influido sobre UML en mayor o menor grado. Como ocurre con todo el trabajo de investigación científica y de ingeniería, UML es una pequeña colina situada en la cima de una montaña gigantesca de experiencia previa.

LENGUAJE
UNIFICADO DE
MODELADO

Parte 1

INTRODUCCIÓN





Capítulo 1

POR QUÉ MODELAMOS

En este capítulo

- La importancia de modelar.
- Cuatro principios del modelado.
- Los planos básicos de un sistema software.
- Modelado orientado a objetos.

Una empresa de software con éxito es aquella que produce de una manera consistente software de calidad que satisface las necesidades de sus usuarios. Una empresa que puede desarrollar este software de forma predecible y puntual, con un uso eficiente y efectivo de recursos, tanto humanos como materiales, tiene un negocio sostenible.

Este mensaje conlleva una importante consecuencia: el producto principal de un equipo de desarrollo no son bonitos documentos, reuniones muy importantes, grandes lemas o líneas de código fuente merecedoras del premio Pulitzer. Antes bien, es un buen software que satisfaga las necesidades cambiantes de sus usuarios y la empresa. Todo lo demás es secundario.

Desafortunadamente, muchas empresas de software confunden “secundario” con “irrelevante”. Para producir software que cumpla su propósito, hay que conocer e involucrar a los usuarios de forma disciplinada con el fin de extraer los requisitos reales del sistema. Para desarrollar software de calidad duradera, hay que idear una sólida base arquitectónica que sea flexible al cambio. Para desarrollar software rápida, eficiente y efectivamente, con el mínimo de desechos software y de trabajo repetido, hay que tener la gente apropiada, las herramientas apropiadas y el enfoque apropiado. Para hacer todo esto de forma consistente y predecible, con una estimación de los costes del sistema en cada etapa de su vida, hay que disponer de un proceso de desarrollo sólido que pueda adaptarse a las necesidades cambiantes del problema en cuestión y de la tecnología.

El modelado es una parte central de todas las actividades que conducen a la producción de buen software. Construimos modelos para comunicar la estructura deseada y el comportamiento de nuestro sistema. Construimos modelos para visualizar y controlar la arquitectura del sistema. Construimos modelos para comprender mejor el sistema que estamos construyendo, muchas veces descubriendo oportunidades para la simplificación y la reutilización. Construimos modelos para controlar el riesgo.

La importancia de modelar

Si se quiere construir una caja para un perro, se puede comenzar muy bien con un montón de madera, algunos clavos y unas cuantas herramientas básicas, tales como un martillo, un serrucho y una cinta métrica. En pocas horas, con poca planificación previa, es probable que se acabe con una caja razonablemente funcional, y que probablemente se pueda hacer sin la ayuda de nadie. Mientras sea bastante grande y no tenga goteras, el perro estará contento. Si no sale bien, siempre se puede volver a empezar, o buscar un perro menos exigente.

Si se quiere construir una casa para una familia, se puede comenzar con un montón de madera, algunos clavos, y unas cuantas herramientas básicas, pero va a llevar más tiempo y, con toda seguridad, la familia será más exigente que el perro. En este caso, a menos que se haya hecho antes muchas veces, se obtendrá un resultado mejor haciendo una planificación detallada antes de golpear el primer clavo o echar los cimientos. Como mínimo, se realizarán algunos bocetos del aspecto que se quiere que tenga la casa. Además, si se quiere construir una casa de calidad que satisfaga las necesidades de la familia y que cumpla las leyes de construcción locales, será necesario dibujar algunos planos, de forma que se pueda pensar en el uso pretendido de las habitaciones y los detalles prácticos de la electricidad, calefacción y fontanería. Dados estos planos, se puede empezar a hacer estimaciones razonables de la cantidad de tiempo y materiales que requerirá el trabajo. Aunque es humanamente posible que uno se construya su propia casa, será mucho más eficiente trabajar con otros, posiblemente subcontratando muchos trabajos importantes o comprando materiales preconstruidos. Mientras se cumplan los planes y se permanezca dentro de las limitaciones de tiempo y dinero, es muy probable que la familia esté satisfecha. Si no sale bien, no se puede cambiar precisamente de familia, así que es mejor establecer las expectativas al principio y controlar los cambios cuidadosamente.

Si se quiere construir un gran bloque de oficinas, sería enormemente estúpido comenzar con un montón de madera, algunos clavos y unas cuantas herramientas

básicas. Como es probable que se esté usando dinero de otra gente, estas personas querrán influir en el tamaño, forma y estilo del edificio. Con frecuencia, ellas cambiarán de opinión, incluso después de haber comenzado la construcción del edificio. Será deseable planificar de forma extensiva, porque el coste de fallar es alto. Se formará parte de un grupo numeroso, responsable de planificar y construir el edificio, y los miembros del equipo necesitarán toda clase de planos y modelos para la comunicación interna.

Mientras se consiga la gente y las herramientas apropiadas y se controle de forma activa el proceso de transformar un concepto arquitectónico en realidad, es probable que se llegue a obtener un edificio que satisfaga a sus inquilinos. Si uno quiere seguir construyendo edificios, deberá asegurarse de compaginar los deseos de los inquilinos con la realidad de la tecnología de la construcción, así como tratar al resto del equipo profesionalmente, sin exponerlo a ningún riesgo, ni dirigiéndolo tan duramente que se quemé.

Curiosamente, un montón de empresas de desarrollo de software comienzan queriendo construir rascacielos, pero enfocan el problema como si estuvieran enfrentándose a la caja de un perro.

A veces se tiene suerte. Si uno tiene la gente apropiada en el momento oportuno y si todos los planetas están en conjunción, entonces se podría, sólo se podría, hacer que el equipo produzca un producto software que deslumbre a sus usuarios. Sin embargo, normalmente no se puede conseguir la gente apropiada (los buenos están muy solicitados), nunca es el momento oportuno (ayer hubiera sido mejor), y los planetas no parecen alinearse (en vez de ello siguen moviéndose fuera de nuestro control). Dada la creciente demanda de desarrollo rápido de software, los equipos de desarrollo a menudo recurren a la única cosa que realmente saben hacer bien: triturar más y más líneas de código. Los esfuerzos de programación heroicos son leyenda en esta industria, y a menudo parece que la reacción apropiada para cualquier crisis en el desarrollo es trabajar más duro. Sin embargo, éstas no son necesariamente las líneas de código apropiadas, y algunos proyectos son de tal magnitud que incluso añadir más horas a la jornada laboral no es suficiente para terminar el trabajo.

Si realmente queremos construir el software equivalente a una casa o un rascacielos, el problema es algo más que una cuestión de escribir grandes cantidades de software —de hecho, el truco está en crear el software apropiado y en imaginar cómo escribir menos software—. Esto convierte al desarrollo de software de calidad en una cuestión de arquitectura, procesos y herramientas. Incluso así, muchos proyectos empiezan pareciendo cajas de perro,

pero crecen hasta el tamaño de un rascacielos, simplemente porque son víctimas de su propio éxito. Llega un momento en el que, si no se ha tenido en cuenta la arquitectura, el proceso o las herramientas, la caseta del perro, convertida ahora en rascacielos, se colapsa bajo su propio peso. El derrumamiento de la caseta puede molestar al perro; el fallo de un gran edificio afectará materialmente a sus inquilinos.

Los proyectos software que fracasan lo hacen por circunstancias propias, pero todos los proyectos con éxito se parecen en muchos aspectos. Hay muchos elementos que contribuyen a una empresa de software con éxito; uno en común es el uso del modelado.

El modelado es una técnica de ingeniería probada y bien aceptada. Construimos modelos arquitectónicos de casas y rascacielos para ayudar a sus usuarios a visualizar el producto final. Incluso podemos construir modelos matemáticos para analizar los efectos de vientos o terremotos sobre nuestros edificios.

El modelado no es sólo parte de la industria de la construcción. Sería inconcebible crear una nueva aeronave o automóvil sin construir previamente modelos, desde modelos informáticos a modelos físicos para el túnel de viento y prototipos a escala real. Los nuevos dispositivos eléctricos, desde los microprocesadores a las centralitas telefónicas, requieren algún grado de modelado para comprender mejor el sistema y comunicar las ideas a otros. En la industria cinematográfica, la técnica del *storyboarding* (representación de una película con viñetas), que es una forma de modelado, es fundamental en cualquier producción. En los campos de la sociología, la economía y la gestión empresarial, se construyen modelos para poder validar las teorías o probar otras nuevas con un coste y unos riesgos mínimos.

¿Qué es, entonces, un modelo? Para responder de forma sencilla,

Un modelo es una simplificación de la realidad.

Un modelo proporciona los planos de un sistema. Los modelos pueden involucrar planos detallados, así como planos más generales que ofrecen una visión global del sistema en consideración. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores que no son relevantes para el nivel de abstracción dado. Todos los sistemas pueden ser descritos desde diferentes perspectivas utilizando diferentes modelos, y cada modelo es, por tanto, una abstracción semánticamente cerrada del sistema. Un modelo puede ser estructural, destacando la organización del sistema, o puede ser de comportamiento, resaltando su dinámica.

¿Por qué modelamos? Hay una razón fundamental.

Construimos modelos para comprender mejor el sistema que estamos desarrollando.

A través del modelado, conseguimos cuatro objetivos:

En el Capítulo 2 se discute cómo aborda UML estas cuatro cuestiones.

1. Los modelos nos ayudan a visualizar cómo es o queremos que sea un sistema.
2. Los modelos nos permiten especificar la estructura o el comportamiento de un sistema.
3. Los modelos nos proporcionan plantillas que nos guían en la construcción de un sistema.
4. Los modelos documentan las decisiones que hemos adoptado.

El modelado no es sólo para los grandes sistemas. Incluso el equivalente software de una caseta de perro puede beneficiarse de algo de modelado. Sin embargo, es absolutamente cierto que, cuanto más grande y complejo es el sistema, el modelado se hace más importante, por una simple razón:

Construimos modelos de sistemas complejos porque no podemos comprender el sistema en su totalidad.

Hay límites a la capacidad humana de comprender la complejidad. A través del modelado, reducimos el problema que se está estudiando, centrándonos en un solo aspecto cada vez. Éste es, esencialmente, el enfoque “divide y vencerás” que planteó Edsger Dijkstra años atrás: acometer un problema difícil dividiéndolo en una serie de subproblemas más pequeños que se pueden resolver. Además, a través del modelado, se potencia la mente humana. Un modelo escogido adecuadamente puede permitir al modelador trabajar a mayores niveles de abstracción.

Decir que se debería modelar no significa que necesariamente se esté haciendo así. De hecho, una serie de estudios sugieren que la mayoría de las organizaciones software hacen poco o ningún modelado formal. Si se representara el uso del modelado frente a la complejidad de un problema, se comprobaría que cuanto más sencillo es el proyecto menos probable es que se haya utilizado un modelado formal.

La palabra clave aquí es “formal”. En realidad, incluso en el proyecto más simple, los desarrolladores hacen algo de modelado, si bien muy informalmente. Un desarrollador puede bosquejar una idea sobre una pizarra o un trozo de papel para visualizar una parte de un sistema, o el equipo puede utilizar tarjetas CRC

para trabajar a través de un escenario o el diseño de un mecanismo. No hay nada malo con ninguno de estos modelos. Si funcionan, deben utilizarse por todos los medios. Sin embargo, estos modelos informales son muchas veces *ad hoc*, y no proporcionan un lenguaje común que se pueda compartir fácilmente con otros. Así como existe un lenguaje común para los planos de la industria de la construcción, un lenguaje común para la ingeniería eléctrica y un lenguaje común para el modelado matemático, también una empresa de software puede beneficiarse utilizando un lenguaje común para el modelado de software.

Cualquier proyecto puede beneficiarse de algo de modelado. Incluso en el dominio del software desecharable, donde a veces es más efectivo deshacerse del software inadecuado a causa de la productividad ofrecida por los lenguajes de programación visuales, el modelado puede ayudar al equipo de desarrollo a visualizar mejor el plano de su sistema y a permitirles desarrollar más rápidamente, al ayudarles a construir el producto apropiado. Cuanto más complejo sea un proyecto, más probable es que se fracase o no se construya el producto apropiado si no se hace nada de modelado. Todos los sistemas interesantes y útiles tienen una tendencia natural a hacerse más complejos con el paso del tiempo. Así que, aunque al inicio se pueda pensar que no es necesario modelar, cuando el sistema evolucione se lamentará esa decisión y entonces será demasiado tarde.

Principios del modelado

El uso del modelado tiene una abundante historia en todas las disciplinas de ingeniería. Esa experiencia sugiere cuatro principios básicos de modelado.

Primero:

La elección acerca de qué modelos crear tiene una profunda influencia sobre cómo se acomete un problema y cómo se da forma a una solución.

En otras palabras, hay que elegir bien los modelos. Los modelos adecuados pueden arrojar mucha luz sobre los problemas de desarrollo más horribles, ofreciendo una comprensión que sencillamente no podríamos obtener de otra manera; los modelos erróneos desorientarán, haciendo que uno se centre en cuestiones irrelevantes.

Dejando a un lado el software por un momento, supongamos que se intenta abordar un problema en física cuántica. Ciertos problemas, tales como la interacción

de fotones en el espacio-tiempo, están llenos de unas matemáticas maravillosamente complicadas. Si se elige un modelo diferente al cálculo, de repente toda la complejidad inherente se vuelve tratable. En este campo, éste es precisamente el valor de los diagramas de Feynmann, que proporcionan una visión gráfica de un problema muy complejo. De forma similar, en un dominio totalmente diferente, supongamos que se está construyendo un nuevo edificio e interesa saber cómo podría comportarse frente a fuertes vientos. Si se construye un modelo físico y se lo somete a las pruebas del túnel de viento, se aprenderán algunas cosas interesantes, aunque los materiales en la maqueta no se doblan exactamente como en la realidad. Por ello, si se construye un modelo matemático y luego se lo somete a simulaciones, se aprenderán cosas diferentes, y probablemente se podrá experimentar con más escenarios nuevos que si se utilizara un modelo físico. Probando los modelos rigurosa y continuamente, se alcanzará un alto nivel de confianza en que el sistema que se ha modelado se comportará de hecho como se espera que lo haga en el mundo real.

En el software, los modelos elegidos pueden afectar mucho a nuestra visión del mundo. Si construimos un sistema con la mirada de un desarrollador de bases de datos, probablemente nos centraremos en los modelos entidad-relación que trasladan el comportamiento a disparadores (*triggers*) y procedimientos almacenados. Si construimos un sistema con la mirada de un analista estructurado, probablemente se obtendrán modelos centrados en los algoritmos, con los datos fluyendo de proceso en proceso. Si construimos un sistema con la mirada de un desarrollador orientado a objetos, se obtendrá un sistema cuya arquitectura se centra en un mar de clases y los patrones de interacción que gobiernan el trabajo conjunto de esas clases. Los modelos ejecutables pueden venir muy bien para las pruebas. Cualquiera de estos enfoques podría estar bien para una aplicación y una cultura de desarrollo dadas, aunque la experiencia sugiere que la visión orientada a objetos es superior al proporcionar arquitecturas flexibles, incluso para sistemas que podrían tener una gran base de datos o una gran componente computacional. No obstante, la cuestión es que cada visión del mundo conduce a un tipo de sistema diferente, con distintos costes y beneficios.

Segundo:

Todo modelo puede ser expresado con diferentes niveles de precisión.

Si se está construyendo un rascacielos, a veces es necesaria una vista de pájaro; por ejemplo, para ayudar a que los inversores se imaginen su aspecto exterior. Otras veces, hay que descender al nivel de los remaches; por ejemplo, cuando hay un recorrido de tuberías enmarañado o un elemento estructural poco habitual.

Lo mismo ocurre con los modelos del software. A veces, un pequeño y sencillo modelo ejecutable de la interfaz del usuario es exactamente lo que se necesita; otras veces, hay que bajar y enredarse con los bits, como cuando se están especificando interfaces entre sistemas o luchando con cuellos de botella en redes. En cualquier caso, los mejores tipos de modelos son aquellos que permiten elegir el grado de detalle, dependiendo de quién está viendo el sistema y por qué necesita verlo. Un analista o un usuario final se centrarán en el qué; un desarrollador se centrará en el cómo. Tanto unos como otros querrán visualizar un sistema a distintos niveles de detalle en momentos diferentes.

Tercero:

Los mejores modelos están ligados a la realidad.

Un modelo físico de un edificio que no responda de la misma forma que los materiales reales tan sólo tiene un valor limitado; un modelo matemático de una aeronave que asuma sólo condiciones ideales y una fabricación perfecta puede enmascarar algunas características potencialmente fatales de la aeronave real. Es mejor tener modelos que tengan una clara conexión con la realidad y, donde esta conexión sea débil, saber exactamente cómo se apartan esos modelos del mundo real. Todos los modelos simplifican la realidad; el truco está en asegurarse de que las simplificaciones no enmascaren ningún detalle importante.

En el software, el talón de Aquiles de las técnicas de análisis estructurado es el hecho de que hay una desconexión básica entre el modelo de análisis y el modelo de diseño del sistema. No poder salvar este abismo hace que el sistema concebido y el sistema construido diverjan con el paso del tiempo. En los sistemas orientados a objetos, es posible conectar todas las vistas casi independientes de un sistema en un todo semántico.

Cuarto:

Un único modelo o vista no es suficiente. Cualquier sistema no trivial se aborda mejor a través de un pequeño conjunto de modelos casi independientes con múltiples puntos de vista.

Si se está construyendo un edificio, no hay un único conjunto de planos que revele todos sus detalles. Como mínimo, se necesitarán planos de las plantas, de los alzados, de electricidad, de calefacción y de fontanería. Y dentro de cada tipo de modelo, se necesitan diferentes vistas para capturar toda la extensión del sistema, como los planos de las diferentes plantas.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2.

La expresión clave aquí es “casi independientes”. En este contexto significa tener modelos que podemos construir y estudiar separadamente, pero aun así están interrelacionados. Como en el caso de un edificio, podemos estudiar los planos eléctricos de forma aislada, pero también podemos ver su correspondencia con los planos de la planta y quizás incluso su interacción con los recorridos de las tuberías en el plano de fontanería.

Lo mismo ocurre en los sistemas software orientados a objetos. Para comprender la arquitectura de tales sistemas, se necesitan varias vistas complementarias y entrelazadas: una vista de casos de uso (que muestre los requisitos del sistema), una vista de diseño (que capture el vocabulario del espacio del problema y del espacio de la solución), una vista de interacción (que muestre las interacciones entre las distintas partes del sistema y entre el sistema y el entorno), una vista de implementación (que se ocupe de la materialización física del sistema) y una vista de despliegue (centrada en cuestiones de ingeniería del sistema). Cada una de estas vistas puede tener aspectos tanto estructurales como de comportamiento. En conjunto, estas vistas representan los planos del software.

Según la naturaleza del sistema, algunas vistas pueden ser más importantes que otras. Por ejemplo, en sistemas con grandes cantidades de datos, dominarán las vistas centradas en el diseño estático. En sistemas con uso intensivo de interfaces gráficas de usuario (GUI), las vistas de casos de uso estáticas y dinámicas son bastante importantes. En los sistemas de tiempo real muy exigentes, las vistas de los procesos dinámicos tienden a ser más importantes. Por último, en los sistemas distribuidos, como los encontrados en aplicaciones de uso intensivo de la Web, los modelos de implementación y despliegue son los más importantes.

Modelado orientado a objetos

Los ingenieros civiles construyen muchos tipos de modelos. Lo más frecuente es que usen modelos estructurales que ayudan a la gente a visualizar y especificar partes de los sistemas y la forma en que esas partes se relacionan entre sí. Dependiendo de las cuestiones más importantes del sistema o de la ingeniería que les preocupen, los ingenieros podrían también construir modelos dinámicos; por ejemplo, para ayudarles a estudiar el comportamiento de una estructura en presencia de un terremoto. Cada tipo de modelo se organiza de forma diferente, y cada uno tiene su propio enfoque.

En el software hay varias formas de enfocar un modelo. Las dos formas más comunes son la perspectiva algorítmica y la perspectiva orientada a objetos.

La visión tradicional del desarrollo de software toma una perspectiva algorítmica. En este enfoque, el bloque principal de construcción de todo el software es el procedimiento o función. Esta visión lleva a los desarrolladores a centrarse en cuestiones de control y de descomposición de algoritmos grandes en otros más pequeños. No hay nada inherentemente malo en este punto de vista, salvo que tiende a producir sistemas frágiles. Cuando los requisitos cambian (*¡cambiarán!*) y el sistema crece (*¡crecerá!*), los sistemas construidos con un enfoque algorítmico resultan muy difíciles de mantener.

La visión actual del desarrollo de software toma una perspectiva orientada a objetos. En este enfoque, el principal bloque de construcción de todos los sistemas software es el objeto o clase. Para explicarlo sencillamente, un objeto es una cosa, generalmente extraída del vocabulario del espacio del problema o del espacio de la solución. Una clase es una descripción de un conjunto de objetos que son lo suficientemente similares (desde la perspectiva del modelador) para compartir una especificación. Todo objeto tiene una identidad (puede nombrarse o distinguirse de otra manera de otros objetos), estado (generalmente hay algunos datos asociados a él) y comportamiento (se le pueden hacer cosas al objeto, y él a su vez puede hacer cosas a otros objetos).

Por ejemplo, considérese una arquitectura sencilla de tres capas para un sistema de contabilidad, que involucre una interfaz de usuario, un conjunto de servicios del negocio y una base de datos. En la interfaz de usuario aparecerán objetos concretos tales como botones, menús y cuadros de diálogo. En la base de datos aparecerán objetos concretos, como tablas que representarán entidades del dominio del problema, incluyendo clientes, productos y pedidos. En la capa intermedia aparecerán objetos tales como transacciones y reglas de negocio, así como vistas de más alto nivel de las entidades del problema, tales como clientes, productos y pedidos.

Actualmente, el enfoque orientado a objetos forma parte de la tendencia principal para el desarrollo de software, simplemente porque ha demostrado ser válido en la construcción de sistemas en toda clase de dominios de problemas, abarcando todo el abanico de tamaños y complejidades. Más aún, la mayoría de los lenguajes actuales, sistemas operativos y herramientas son orientados a objetos de alguna manera, lo que ofrece más motivos para ver el mundo en términos de objetos. El desarrollo orientado a objetos proporciona la base fundamental para ensamblar sistemas a partir de componentes utilizando tecnologías como J2EE o .NET.

Estas cuestiones se discuten en el Capítulo 2.

De la elección de ver el mundo de una forma orientada a objetos se derivan varias implicaciones: ¿Cuál es la estructura de una buena arquitectura orientada a objetos? ¿Qué artefactos debería crear el proyecto? ¿Quién debería crearlos? ¿Cómo deberían medirse?

Visualizar, especificar, construir y documentar sistemas orientados a objetos es exactamente el propósito del Lenguaje Unificado de Modelado (*Unified Modeling Language*).



Capítulo 2

PRESENTACIÓN DE UML

En este capítulo

- Visión general de UML.
- Tres pasos para comprender UML.
- Arquitectura del software.
- El proceso de desarrollo de software.

El Lenguaje Unificado de Modelado (*Unified Modeling Language*, UML) es un lenguaje estándar para escribir planos de software. UML puede utilizarse para visualizar, especificar, construir y documentar los artefactos de un sistema que involucre una gran cantidad de software.

UML es apropiado para modelar desde sistemas de información empresariales hasta aplicaciones distribuidas basadas en la Web, e incluso para sistemas embedidos de tiempo real muy exigentes. Es un lenguaje muy expresivo, que cubre todas las vistas necesarias para desarrollar y luego desplegar tales sistemas. Aunque sea expresivo, UML no es difícil de aprender ni de utilizar. Aprender a aplicar UML de modo eficaz comienza por crearse un modelo conceptual del lenguaje, lo cual requiere aprender tres elementos principales: los bloques básicos de construcción de UML, las reglas que dictan cómo pueden combinarse esos bloques y algunos mecanismos comunes que se aplican a lo largo de todo el lenguaje.

UML es sólo un lenguaje y, por tanto, es tan sólo una parte de un método de desarrollo de software. UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Visión general de UML

UML es un lenguaje para

- Visualizar.
- Especificar.
- Construir.
- Documentar.

los artefactos de un sistema con gran cantidad de software.

UML es un lenguaje

Un lenguaje proporciona un vocabulario y las reglas para combinar palabras de ese vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de *modelado* es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física de un sistema. Un lenguaje de modelado como UML es, por tanto, un lenguaje estándar para los planos del software.

Los principios básicos del modelado se discuten en el Capítulo 1.

El modelado proporciona una comprensión de un sistema. Nunca es suficiente un único modelo. Más bien, para comprender cualquier cosa, a menudo se necesitan múltiples modelos conectados entre sí, excepto en los sistemas más triviales. Para sistemas con gran cantidad de software, se requiere un lenguaje que abarque las diferentes vistas de la arquitectura de un sistema conforme evoluciona a través del ciclo de vida del desarrollo de software.

El vocabulario y las reglas de un lenguaje como UML indican cómo crear y leer modelos bien formados, pero no dicen qué modelos se deben crear ni cuándo se deberían crear. Ésta es la tarea del proceso de desarrollo de software. Un proceso bien definido guiará a sus usuarios al decidir qué artefactos producir, qué actividades y qué personal se emplea para crearlos y gestionarlos, y cómo usar esos artefactos para medir y controlar el proyecto de forma global.

UML es un lenguaje para visualizar

Para muchos programadores, la distancia entre pensar en una implementación y transformarla en código es casi cero. Lo piensas, lo codificas. De hecho, algunas cosas se modelan mejor directamente en código. El texto es un medio maravilloso para escribir expresiones y algoritmos de forma concisa y directa.

En tales casos, el programador todavía está haciendo algo de modelado, si bien lo hace de forma completamente mental. Incluso puede bosquejar algunas ideas sobre una pizarra blanca o sobre una servilleta. Sin embargo, esto plantea algunos problemas. Primero, la comunicación de esos modelos conceptuales a otros está sujeta a errores a menos que cualquier persona implicada hable el mismo lenguaje. Normalmente, los proyectos y las organizaciones desarrollan su propio lenguaje, y es difícil comprender lo que está pasando para alguien nuevo o ajeno al grupo. Segundo, hay algunas cuestiones sobre un sistema software que no se pueden entender a menos que se construyan modelos que trasciendan el lenguaje de programación textual. Por ejemplo, el significado de una jerarquía de clases puede inferirse, pero no capturarse completamente, inspeccionando el código de todas las clases en la jerarquía. De forma parecida, la distribución física y posible migración de los objetos en un sistema basado en la Web puede inferirse, pero no capturarse completamente, estudiando el código fuente del sistema. Tercero, si el desarrollador que escribió el código no dejó documentación escrita sobre los modelos que había en su cabeza, esa información se perderá para siempre o, como mucho, será sólo parcialmente reproducible a partir de la implementación, una vez que el desarrollador se haya marchado.

Al escribir modelos en UML se afronta el tercer problema: un modelo explícito facilita la comunicación.

Algunas cosas se modelan mejor textualmente; otras se modelan mejor de forma gráfica. En realidad, en todos los sistemas interesantes hay estructuras que trascienden de lo que puede ser representado en un lenguaje de programación. UML es uno de estos lenguajes gráficos. Así afronta el segundo problema mencionado anteriormente.

La semántica completa de UML se discute en The Unified Modeling Language Reference Manual.

UML es algo más que un simple montón de símbolos gráficos. Detrás de cada símbolo en la notación UML hay una semántica bien definida. De esta manera, un desarrollador puede escribir un modelo en UML, y otro desarrollador, o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad. Así afronta el primer problema mencionado anteriormente.

UML es un lenguaje para especificar

En este contexto, *especificar* significa construir modelos precisos, no ambiguos y completos. En particular, UML cubre la especificación de todas las decisiones de análisis, diseño e implementación que deben realizarse al desarrollar y desplegar un sistema con gran cantidad de software.

UML es un lenguaje para construir

UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse de forma directa a una gran variedad de lenguajes de programación. Esto significa que es posible establecer correspondencias desde un modelo UML a un lenguaje de programación como Java, C++ o Visual Basic, o incluso a tablas en una base de datos relacional o al almacenamiento persistente en una base de datos orientada a objetos. Las cosas que se expresan mejor gráficamente también se representan gráficamente en UML, mientras que las cosas que se expresan mejor textualmente se plasman con el lenguaje de programación.

El modelado de la estructura de un sistema se discute en las Partes 2 y 3.

Esta correspondencia permite ingeniería directa: la generación de código a partir de un modelo UML en un lenguaje de programación. Lo contrario también es posible: se puede reconstruir un modelo en UML a partir de una implementación. La ingeniería inversa no es magia. A menos que se codifique esa información en la implementación, la información se pierde cuando se pasa de los modelos al código. La ingeniería inversa requiere, por tanto, herramientas que la soporten e intervención humana. La combinación de estas dos vías de generación de código y de ingeniería inversa produce una ingeniería “*de ida y vuelta*”, entendiendo por esto la posibilidad de trabajar en una vista gráfica o en una textual, mientras las herramientas mantienen la consistencia entre ambas.

El modelado del comportamiento de un sistema se discute en las Partes 4 y 5.

Además de esta correspondencia directa, UML es lo suficientemente expresivo y no ambiguo para permitir la ejecución directa de modelos, la simulación de sistemas y la coordinación de sistemas en ejecución.

UML es un lenguaje para documentar

Una organización de software que trabaja bien produce toda clase de artefactos, además de código ejecutable. Estos artefactos incluyen (aunque no se limitan a):

- Requisitos.
- Arquitectura.
- Diseño.
- Código fuente.
- Planificación de proyectos.
- Pruebas.
- Prototipos.
- Versiones.

Dependiendo de la cultura de desarrollo, algunos de estos artefactos se tratan más o menos formalmente que otros. Tales artefactos no son tan sólo los entregables de un proyecto, también son críticos para el control, la medición y comunicación que requiere un sistema durante su desarrollo y después de su despliegue.

UML cubre la documentación de la arquitectura de un sistema y todos sus detalles. UML también proporciona un lenguaje para expresar requisitos y pruebas. Por último, UML proporciona un lenguaje para modelar las actividades de planificación de proyectos y gestión de versiones.

¿Dónde puede utilizarse UML?

UML está pensado principalmente para sistemas con gran cantidad de software. Ha sido utilizado de forma efectiva en dominios tales como:

- Sistemas de información empresariales.
- Bancos y servicios financieros.
- Telecomunicaciones.
- Transporte.
- Defensa/industria aeroespacial.
- Comercio.
- Electrónica médica.
- Ámbito científico.
- Servicios distribuidos basados en la Web.

UML no está limitado al modelado de software. De hecho, es lo suficientemente expresivo para modelar sistemas que no son software, como flujos de trabajo (*workflows*) en el sistema jurídico, estructura y comportamiento de un sistema de vigilancia médica de un enfermo, y el diseño de hardware.

Un modelo conceptual de UML

Para comprender UML, se necesita adquirir un modelo conceptual del lenguaje, y esto requiere aprender tres elementos principales: los bloques básicos de construcción de UML, las reglas que dictan cómo se pueden combinar estos bloques básicos y algunos mecanismos comunes que se aplican a través de UML. Una vez comprendidas estas ideas, se pueden leer modelos UML y crear algunos modelos básicos. Conforme se gana más experiencia en la aplicación de UML, se puede ampliar a partir de este modelo conceptual, utilizando características más avanzadas del lenguaje.

Bloques básicos de UML

El vocabulario de UML incluye tres clases de bloques básicos:

1. Elementos.
2. Relaciones.
3. Diagramas.

Los elementos son abstracciones que constituyen los ciudadanos de primera clase en un modelo; las relaciones ligan estos elementos entre sí; los diagramas agrupan colecciones interesantes de elementos.

Elementos en UML. Hay cuatro tipos de elementos en UML:

1. Elementos estructurales.
2. Elementos de comportamiento.
3. Elementos de agrupación.
4. Elementos de anotación.

Estos elementos son los bloques básicos de construcción orientados a objetos de UML. Se utilizan para escribir modelos bien formados.

Elementos estructurales. Los *elementos estructurales* son los nombres de los modelos UML. En su mayoría son las partes estáticas de un modelo, y representan conceptos o cosas materiales. Colectivamente, los elementos estructurales se denominan *clasificadores*.

Las clases se discuten en los Capítulos 4 y 9.

Una *clase* es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces. Gráficamente, una clase se representa como un rectángulo, que normalmente incluye su nombre, atributos y operaciones, como se muestra en la Figura 2.1.



Figura 2.1: Clases.

Las interfaces se discuten en el Capítulo 11.

Una *interfaz* es una colección de operaciones que especifican un servicio de una clase o componente. Por tanto, una interfaz describe el comportamiento visible externamente de ese elemento. Una interfaz puede representar el comportamiento completo de una clase o componente o sólo una parte de ese comportamiento. Una interfaz define un conjunto de especificaciones de operaciones (o sea, sus *signaturas*), pero nunca un conjunto de implementaciones de operaciones. La declaración de una interfaz se asemeja a la de una clase con la palabra «interface» sobre el nombre; los atributos no son relevantes, excepto algunas veces, para mostrar constantes. Una interfaz raramente se encuentra aislada. Cuando una clase proporciona una interfaz al mundo externo, ésta se representa como una pequeña circunferencia unida al rectángulo que representa a la clase por una línea. Una interfaz requerida por una clase, y que será proporcionada por otra clase, se representa como una pequeña semicircunferencia unida al rectángulo de la primera clase por una línea, como se muestra en la Figura 2.2.

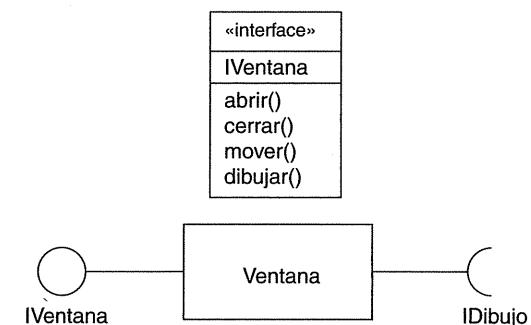


Figura 2.2: Interfaces.

Las colaboraciones se discuten en el Capítulo 28.

Una *colaboración* define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Las colaboraciones tienen tanto una dimensión estructural como una de comportamiento. Una clase dada o un objeto puede participar en varias colaboraciones. Estas colaboraciones representan la implementación de patrones que configuran un sistema. Gráficamente, una colaboración se representa como una elipse de borde discontinuo, que normalmente incluye sólo su nombre, como se muestra en la Figura 2.3.

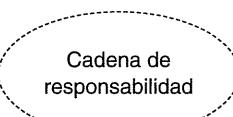


Figura 2.3: Colaboraciones.

Los casos de uso se discuten en el Capítulo 17.

Un *caso de uso* es una descripción de un conjunto de secuencias de acciones que ejecuta un sistema y que produce un resultado observable de interés para un actor particular. Un caso de uso se utiliza para estructurar los aspectos de comportamiento en un modelo. Un caso de uso es realizado por una colaboración. Gráficamente, un caso de uso se representa como una elipse de borde continuo, y normalmente incluye sólo su nombre, como se muestra en la Figura 2.4.

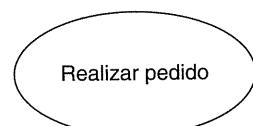


Figura 2.4: Casos de Uso.

Los cuatro elementos restantes (clases activas, componentes, artefactos y nodos) son todos similares a las clases, en cuanto que también describen un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Sin embargo, estos cuatro son suficientemente diferentes y son necesarios para modelar ciertos aspectos de un sistema orientado a objetos; así que está justificado un tratamiento especial.

Las clases activas se discuten en el Capítulo 23.

Una *clase activa* es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución y, por tanto, pueden dar origen a actividades de control. Una clase activa es igual que una clase, excepto en que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos. Gráficamente, una clase activa se representa como una clase, pero con líneas dobles a derecha e izquierda; normalmente incluye su nombre, atributos y operaciones, como se muestra en la Figura 2.5.

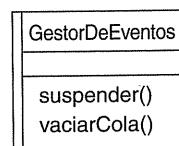


Figura 2.5: Clases Activas.

Los componentes y la estructura interna se discuten en el Capítulo 15.

Un *componente* es una parte modular del diseño del sistema que oculta su implementación tras un conjunto de interfaces externas. En un sistema, los componentes que comparten las mismas interfaces pueden sustituirse siempre y cuando conserven el mismo comportamiento lógico. La implementación de un componente puede expresarse conectando partes y conectores; las partes pueden incluir componentes más pequeños. Gráficamente, un componente se representa

como una clase con un icono especial en la esquina superior derecha, como se muestra en la Figura 2.6.

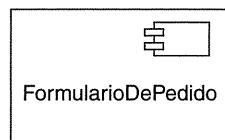


Figura 2.6: Componentes.

Los artefactos se discuten en el Capítulo 26.

Los dos elementos restantes (artefactos y nodos) también son diferentes. Representan elementos físicos, mientras que los seis elementos anteriores representan cosas conceptuales o lógicas.

Un *artefacto* es una parte física y reemplazable de un sistema que contiene información física (“bits”). En un sistema hay diferentes tipos de artefactos de despliegue, como archivos de código fuente, ejecutables y scripts. Un artefacto representa típicamente el empaquetamiento físico de código fuente o información en tiempo de ejecución. Gráficamente, un artefacto se representa como un rectángulo con la palabra clave «artifact» sobre el nombre, como se muestra en la Figura 2.7.

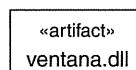


Figura 2.7: Artefactos.

Los nodos se discuten en el Capítulo 27.

Un *nodo* es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que por lo general dispone de algo de memoria y, con frecuencia, capacidad de procesamiento. Un conjunto de artefactos puede residir en un nodo y puede también migrar de un nodo a otro. Gráficamente, un nodo se representa como un cubo, incluyendo normalmente sólo su nombre, como se muestra en la Figura 2.8.

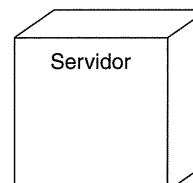


Figura 2.8: Nodos.

Estos elementos (clases, interfaces, colaboraciones, casos de uso, clases activas, componentes, artefactos y nodos) son los elementos estructurales básicos que se pueden incluir en un modelo UML. También existen variaciones de estos elementos, tales como actores, señales, utilidades (un tipo de clases), procesos e hilos (tipos de clases activas), y aplicaciones, documentos, archivos, bibliotecas, páginas y tablas (tipos de artefactos).

Los casos de uso, que se utilizan para estructurar los elementos de comportamiento de un modelo, se discuten en el Capítulo 17; las interacciones se discuten en el Capítulo 16.

Elementos de comportamiento. Los *elementos de comportamiento* son las partes dinámicas de los modelos UML. Éstos son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio. En total hay tres tipos principales de elementos de comportamiento:

En primer lugar, una *interacción* es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un propósito específico. El comportamiento de una sociedad de objetos o una operación individual puede especificarse con una interacción. Una interacción involucra a muchos otros elementos, que incluye mensajes, acciones y enlaces (conexiones entre objetos). Gráficamente, un mensaje se muestra como una línea dirigida, incluyendo casi siempre el nombre de su operación, como se muestra en la Figura 2.9.

dibujar →

Figura 2.9: Mensajes.

Las máquinas de estados se discuten en el Capítulo 22.

En segundo lugar, una *máquina de estados* es un comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos. El comportamiento de una clase individual o una colaboración de clases puede especificarse con una máquina de estados. Una máquina de estados involucra a otros elementos, incluyendo estados, transiciones (el flujo de un estado a otro), eventos (que disparan una transición) y actividades (la respuesta a una transición). Gráficamente, un estado se representa como un rectángulo de esquinas redondeadas, que incluye normalmente su nombre y sus subestados, si los tiene, como se muestra en la Figura 2.10.

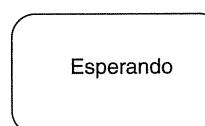


Figura 2.10: Estados.

En tercer lugar, una *actividad* es un comportamiento que especifica la secuencia de pasos que ejecuta un proceso computacional. En una interacción, el énfasis se pone en el conjunto de objetos que interactúan. En una máquina de estados, el énfasis se pone en el ciclo de vida de un objeto cada vez. En una actividad, el énfasis se pone en los flujos entre los pasos, sin mirar qué objeto ejecuta cada paso. Un paso de una actividad se denomina una *acción*. Gráficamente, una acción se representa como un rectángulo con las esquinas redondeadas, con un nombre que indica su propósito. Los estados y las acciones se distinguen por sus diferentes contextos.

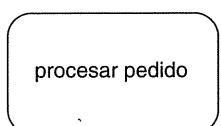


Figura 2.11. Acciones.

Los paquetes se discuten en el Capítulo 12.

Estos tres elementos (interacciones, máquinas de estados y actividades) son los elementos básicos de comportamiento que se pueden incluir en un modelo UML. Semánticamente, estos elementos están conectados normalmente a diversos elementos estructurales, principalmente clases, colaboraciones y objetos.

Elementos de agrupación. Los *elementos de agrupación* son las partes organizativas de los modelos UML. Éstos son las cajas en las que puede descomponerse un modelo. Hay un tipo principal de elementos de agrupación: los paquetes.

Un *paquete* es un mecanismo de propósito general para organizar el propio diseño, en oposición a las clases, que organizan construcciones de implementación. Los elementos estructurales, los elementos de comportamiento e incluso otros elementos de agrupación pueden incluirse en un paquete. Al contrario que los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente, un paquete se visualiza como una carpeta, que normalmente sólo incluye su nombre y, a veces, su contenido, como se muestra en la Figura 2.12.

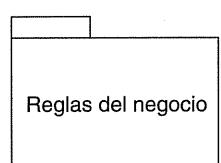


Figura 2.12. Paquetes.

Los paquetes son los elementos de agrupación básicos con los cuales se puede organizar un modelo UML. También hay variaciones, tales como los *frameworks*, los modelos y los subsistemas (tipos de paquetes).

Las notas se discuten en el Capítulo 6.

Elementos de anotación. Los *elementos de anotación* son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento de un modelo. Hay un tipo principal de elemento de anotación llamado nota. Una *nota* es simplemente un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos. Gráficamente, una nota se representa como un rectángulo con una esquina doblada, junto con un comentario textual o gráfico, como se muestra en la Figura 2.13.

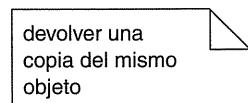


Figura 2.13: Notas.

Este elemento es el objeto básico de anotación que se puede incluir en un modelo UML. Típicamente, las notas se utilizarán para adornar los diagramas con restricciones o comentarios que se expresen mejor en texto informal o formal. También hay variaciones sobre este elemento, tales como los requisitos (que especifican algún comportamiento deseado desde la perspectiva externa del modelo).

Relaciones en UML. Hay cuatro tipos de relaciones en UML:

1. Dependencia.
2. Asociación.
3. Generalización.
4. Realización.

Estas cuatro son los bloques básicos de construcción para relaciones en UML. Se utilizan para escribir modelos bien formados:

Las dependencias se discuten en los Capítulos 5 y 10.

En primer lugar, una *dependencia* es una relación semántica entre dos elementos, en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (el elemento dependiente). Gráficamente, una dependencia se representa como una línea discontinua, posiblemente dirigida, que incluye a veces una etiqueta, como se muestra en la Figura 2.14.



Figura 2.14: Dependencias.

Las asociaciones se discuten en los Capítulos 5 y 10.

En segundo lugar, una *asociación* es una relación estructural entre clases que describe un conjunto de enlaces, los cuales son conexiones entre objetos que son instancias de clases. La agregación es un tipo especial de asociación, que representa una relación estructural entre un todo y sus partes. Gráficamente, una asociación se representa como una línea continua, posiblemente dirigida, que a veces incluye una etiqueta, y a menudo incluye otros adornos, como la multiplicidad y los nombres de rol, como se muestra en la Figura 2.15.

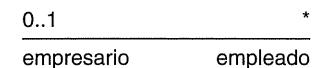


Figura 2.15: Asociaciones.

Las generalizaciones se discuten en los Capítulos 5 y 10.

En tercer lugar, una *generalización* es una relación de especialización/generalización en la cual el elemento especializado (el hijo) se basa en la especificación del elemento generalizado (el padre). El hijo comparte la estructura y el comportamiento del padre. Gráficamente, una relación de generalización se representa como una línea continua con una punta de flecha vacía apuntando al padre, como se muestra en la Figura 2.16.



Figura 2.16: Generalizaciones.

Las realizaciones se discuten en el Capítulo 10.

En cuarto lugar, una *realización* es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. Gráficamente, una relación de realización se representa como una mezcla entre una generalización y una relación de dependencia, como se muestra en la Figura 2.17.



Figura 2.17: Realizaciones.

Estos cuatro elementos son los elementos básicos relacionales que se pueden incluir en un modelo UML. También existen variaciones de estos cuatro, tales como el refinamiento, la traza, la inclusión y la extensión.

Las cinco vistas de una arquitectura se discuten más adelante en este mismo capítulo.

Diagramas en UML. Un *diagrama* es la representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una

proyección de un sistema. En todos los sistemas, excepto en los más triviales, un diagrama representa una vista resumida de los elementos que constituyen un sistema. El mismo elemento puede aparecer en todos los diagramas, sólo en unos pocos diagramas (el caso más común), o en ningún diagrama (un caso muy raro). En teoría, un diagrama puede contener cualquier combinación de elementos y relaciones. En la práctica, sin embargo, sólo surge un pequeño número de combinaciones habituales, las cuales son consistentes con las cinco vistas más útiles que comprenden la arquitectura de un sistema con gran cantidad de software. Por esta razón, UML incluye trece tipos de diagramas:

1. Diagrama de clases.
2. Diagrama de objetos.
3. Diagrama de componentes.
4. Diagrama de estructura compuesta.
5. Diagrama de casos de uso.
6. Diagrama de secuencia.
7. Diagrama de comunicación.
8. Diagrama de estados.
9. Diagrama de actividades.
10. Diagrama de despliegue.
11. Diagrama de paquetes.
12. Diagrama de tiempos.
13. Diagrama de visión global de interacciones.

Los diagramas de clases se discuten en el Capítulo 8.

Un *diagrama de clases* muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los diagramas más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clases abarcan la vista de diseño estática de un sistema. Los diagramas de clases que incluyen clases activas cubren la vista de procesos estática de un sistema. Los diagramas de componentes son una variante de los diagramas de clases.

Los diagramas de objetos se discuten en el Capítulo 14.

Un *diagrama de objetos* muestra un conjunto de objetos y sus relaciones. Los diagramas de objetos representan instantáneas estáticas de instancias de los elementos existentes en los diagramas de clases. Estos diagramas cubren la vista de diseño estática o la vista de procesos estática de un sistema, como lo hacen los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos.

Los diagramas de componentes y la estructura interna se discuten en el Capítulo 15.

Un *diagrama de componentes* representa la encapsulación de una clase, junto con sus interfaces, puertos y estructura interna, la cual está formada por otros componentes anidados y conectores. Los diagramas de componentes cubren la vista de implementación estática del diseño de un sistema. Son importantes para construir sistemas más grandes a partir de partes pequeñas.

(UML distingue entre un *diagrama de estructura compuesta*, aplicable a cualquier clase, y un diagrama de componentes, pero unificaremos la discusión ya que la diferencia entre un componente y una clase estructurada es innecesariamente sutil.)

Los diagramas de casos de uso se discuten en el Capítulo 18.

Un *diagrama de casos de uso* muestra un conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Los diagramas de casos de uso cubren la vista de casos de uso estática de un sistema. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento de un sistema.

Los diagramas de interacción se discuten en el Capítulo 19.

Tanto los diagramas de secuencia como los diagramas de comunicación son tipos de diagramas de interacción. Un *diagrama de interacción* muestra una interacción, que consta de un conjunto de objetos o roles, incluyendo los mensajes que pueden ser enviados entre ellos. Los diagramas de interacción cubren la vista dinámica de un sistema. Un *diagrama de secuencia* es un diagrama de interacción que resalta la ordenación temporal de los mensajes; un *diagrama de comunicación* es un diagrama de interacción que resalta la organización estructural de los objetos o roles que envían y reciben mensajes. Los diagramas de secuencia y los diagramas de comunicación representan conceptos básicos similares, pero cada diagrama resalta una vista diferente de los conceptos. Los diagramas de secuencia resaltan la ordenación temporal, y los diagramas de comunicación resaltan la estructura de datos a través de la cual fluyen los mensajes. Un *diagrama de tiempos* (que no se trata en este libro) muestra los tiempos reales en los que se intercambian los mensajes.

Los diagramas de estados se discuten en el Capítulo 18.

Un *diagrama de estados* muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Un diagrama de estados muestra la vista dinámica de un objeto. Son especialmente importantes en el modelado del comportamiento de una interfaz, una clase o una colaboración y resaltan el comportamiento dirigido por eventos de un objeto, lo cual es especialmente útil en el modelado de sistemas reactivos.

Los diagramas de actividades se discuten en el Capítulo 20.

Un *diagrama de actividades* muestra la estructura de un proceso u otra computación como el flujo de control y datos paso a paso en la computación. Los diagramas de actividades cubren la vista dinámica de un sistema. Son especialmente importantes al modelar el funcionamiento de un sistema y resaltan el flujo de control entre objetos.

Los diagramas de despliegue se discuten en el Capítulo 31.

Un *diagrama de despliegue* muestra la configuración de nodos de procesamiento en tiempo de ejecución y los artefactos que residen en ellos. Los diagramas de despliegue abordan la vista de despliegue estática de una arquitectura. Normalmente, un nodo alberga uno o más artefactos.

Los diagramas de artefactos se discuten en el Capítulo 30.

Un *diagrama de artefactos* muestra los constituyentes físicos de un sistema en el computador. Los artefactos incluyen archivos, bases de datos y colecciones físicas de bits similares. Los artefactos se utilizan a menudo junto con los diagramas de despliegue. Los artefactos también muestran las clases y componentes que implementan. (UML trata a los diagramas de artefactos como una variación de los diagramas de despliegue, pero los discutiremos por separado).

Los diagramas de paquetes se discuten en el Capítulo 12.

Un *diagrama de paquetes* muestra la descomposición del propio modelo en unidades organizativas y sus dependencias.

Un *diagrama de tiempos* es un diagrama de interacción que muestra los tiempos reales entre diferentes objetos o roles, en oposición a la simple secuencia relativa de mensajes. Un *diagrama de visión global de interacciones* es un híbrido entre un diagrama de actividades y un diagrama de secuencia. Estos diagramas tienen usos especializados y, por tanto, no los trataremos en este libro. Para más detalles puede verse el *Manual de Referencia de UML*.

Ésta no es una lista cerrada de diagramas. Las herramientas pueden utilizar UML para proporcionar otros tipos de diagramas, aunque éstos son los que con mayor frecuencia aparecerán en la práctica.

Reglas de UML

Los bloques de construcción de UML no pueden combinarse simplemente de cualquier manera. Como cualquier lenguaje, UML tiene un número de reglas que especifican a qué debe parecerse un modelo bien formado. Un *modelo bien formado* es aquel que es semánticamente autoconsistente y está en armonía con todos sus modelos relacionados.

UML tiene reglas sintácticas y semánticas para:

- Nombres Cómo llamar a los elementos, relaciones y diagramas.
- Alcance El contexto que da un significado específico a un nombre.
- Visibilidad Cómo se pueden ver y utilizar esos nombres por otros.
- Integridad Cómo se relacionan apropiada y consistentemente unos elementos con otros.
- Ejecución Qué significa ejecutar o simular un modelo dinámico.

Los modelos construidos durante el desarrollo de un sistema con gran cantidad de software tienden a evolucionar y pueden ser vistos por muchos usuarios de

formas diferentes y en momentos diferentes. Por esta razón, es habitual que el equipo de desarrollo no sólo construya modelos bien formados, sino también modelos que sean:

- Abreviados Ciertos elementos se ocultan para simplificar la vista.
- Incompletos Pueden estar ausentes ciertos elementos.
- Inconsistentes No se garantiza la integridad del modelo.

Estos modelos que no llegan a ser bien formados son inevitables conforme los detalles de un sistema van apareciendo y mezclándose durante el proceso de desarrollo de software. Las reglas de UML estimulan (pero no obligan) a considerar las cuestiones más importantes de análisis, diseño e implementación que llevan a tales sistemas a convertirse en bien formados con el paso del tiempo.

Mecanismos comunes en UML

Un edificio se hace más simple y más armonioso al ajustarse a un patrón de características comunes. Una casa puede construirse, en su mayor parte, de estilo victoriano o francés utilizando ciertos patrones arquitectónicos que definen esos estilos. Lo mismo es cierto para UML. Éste se simplifica mediante la presencia de cuatro mecanismos comunes que se aplican de forma consistente a través de todo el lenguaje:

1. Especificaciones.
2. Adornos.
3. Divisiones comunes.
4. Mecanismos de extensibilidad.

Especificaciones. UML es algo más que un lenguaje gráfico. Detrás de cada elemento de su notación gráfica hay una especificación que proporciona una explicación textual de la sintaxis y semántica de ese bloque de construcción. Por ejemplo, detrás de un ícono de clase hay una especificación que proporciona el conjunto completo de atributos, operaciones (incluyendo sus signaturas completas) y comportamiento que incluye la clase; visualmente, ese ícono de clase puede mostrar sólo una pequeña parte de su especificación. Aparte de esto, podría haber otra vista de esa clase que presentara un conjunto de elementos totalmente diferente, siendo aún consistente con la especificación subyacente de la clase. La notación gráfica de UML se utiliza para visualizar un sistema; la especificación de UML se utiliza para expresar los detalles del sistema. Hecha esta división, es posible construir un modelo de forma incremental dibujando

primero diagramas y añadiendo después semántica a las especificaciones del modelo, o bien directamente, mediante la creación de una especificación primero, quizás haciendo ingeniería inversa sobre un sistema, y después creando los diagramas que se obtienen como proyecciones de esas especificaciones.

Las especificaciones de UML proporcionan una base semántica que incluye a todas las partes de todos los modelos de un sistema, y cada parte está relacionada con las otras de manera consistente. Los diagramas de UML son así simples proyecciones visuales sobre esa base, y cada diagrama revela un aspecto específico interesante del sistema.

Las notas y otros adornos se discuten en el Capítulo 6.

Adornos. La mayoría de los elementos de UML tienen una notación gráfica única y clara que proporciona una representación visual de los aspectos más importantes del elemento. Por ejemplo, la notación para una clase se ha diseñado intencionadamente de forma que sea fácil de dibujar, porque las clases son los elementos que aparecen con más frecuencia al modelar sistemas orientados a objetos. La notación de la clase también revela los aspectos más importantes de una clase, a saber: su nombre, sus atributos y sus operaciones.

La especificación de una clase puede incluir otros detalles, tales como si es abstracta o la visibilidad de sus atributos y operaciones. Muchos de estos detalles se pueden incluir como adornos gráficos o textuales en la notación rectangular básica de la clase. Por ejemplo, la Figura 2.18 muestra una clase con adornos, que indican que es una clase abstracta con dos operaciones públicas, una protegida y la otra privada.

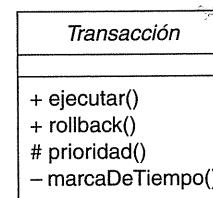


Figura 2.18: Adornos.

Todos los elementos en la notación UML parten de un símbolo básico, al cual pueden añadirse una variedad de adornos específicos de ese símbolo.

Divisiones comunes. Al modelar sistemas orientados a objetos, el mundo a menudo se divide de varias formas.

En primer lugar, está la división entre clase y objeto. Una clase es una abstracción; un objeto es una manifestación concreta de esa abstracción. En UML se pueden modelar tanto clases como objetos, como se muestra en la Figura 2.19.

Los objetos se discuten en el Capítulo 13.

Gráficamente, UML distingue a un objeto utilizando el mismo símbolo que para la clase, y a continuación subrayando el nombre del objeto.

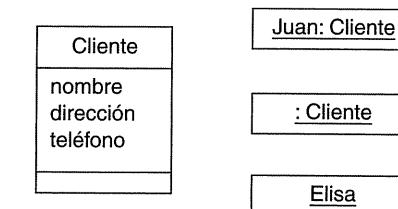


Figura 2.19: Clases y objetos.

En esta figura hay una clase, llamada Cliente, junto a tres objetos: Juan, (del que se indica explícitamente que es un objeto Cliente), : Cliente (un objeto Cliente anónimo) y Elisa (el cual se ha etiquetado en su especificación como un objeto de la clase Cliente, aunque aquí no se muestra de forma explícita).

Casi todos los bloques de construcción de UML presentan este mismo tipo de dicotomía clase/objeto. Por ejemplo, se pueden tener casos de uso y ejecuciones de casos de uso, componentes e instancias de componentes, nodos e instancias de nodos, etcétera.

Las interfaces se discuten en el Capítulo 11.

En segundo lugar, tenemos la separación entre interfaz e implementación. Una interfaz declara un contrato, y una implementación representa una realización concreta de ese contrato, responsable de hacer efectiva la semántica completa de la interfaz de forma fidedigna. En UML se pueden modelar las interfaces y sus implementaciones, como se muestra en la Figura 2.20.

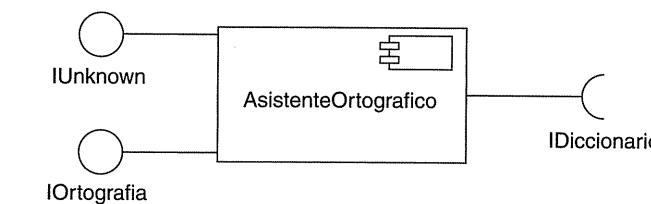


Figura 2.20: Interfaces e implementaciones.

En esta figura hay un componente llamado AsistenteOrtografico.dll que proporciona (implementa) dos interfaces, IUnknown e IOrtografia. También requiere de una interfaz, IDiccionario, que deberá ser proporcionada por otro componente.

Casi todos los bloques de construcción de UML presentan este mismo tipo de dicotomía interfaz/implementación. Por ejemplo, se pueden tener casos de uso

y las colaboraciones que los realizan, así como operaciones y los métodos que las implementan.

En tercer lugar, está la separación entre tipo y rol. El tipo declara la clase de una entidad, como un objeto, un atributo o un parámetro. Un rol describe el significado de una entidad en un contexto, como una clase, un componente o una colaboración. Cualquier entidad que forma parte de la estructura de otra entidad, como un atributo, tiene ambas características: deriva parte de su significado de su tipo inherente, y otra parte del significado de su rol dentro del contexto (Figura 2.21).

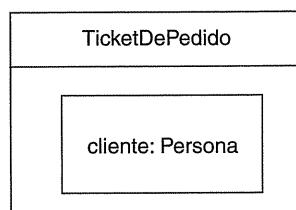


Figura 2.21: Parte con rol y tipo.

Los mecanismos de extensibilidad se discuten en el Capítulo 6.

Mecanismos de extensibilidad. UML proporciona un lenguaje estándar para escribir planos software, pero no es posible que un lenguaje cerrado sea siempre suficiente para expresar todos los matices posibles de todos los modelos en todos los dominios y en todo momento. Por esta razón, UML es abierto-cerrado, siendo posible extender el lenguaje de manera controlada. Los mecanismos de extensión de UML comprenden:

- Estereotipos.
- Valores etiquetados.
- Restricciones.

Un *estereotipo* extiende el vocabulario de UML, permitiendo crear nuevos tipos de bloques de construcción que derivan de los existentes pero que son específicos a un problema. Por ejemplo, si se trabaja en un lenguaje de programación como Java o C++, a menudo habrá que modelar las excepciones. En estos lenguajes, las excepciones son simplemente clases, aunque se tratan de formas muy especiales. Normalmente, sólo se permitirá que sean lanzadas y capturadas, nada más. Se puede hacer que las excepciones sean ciudadanos de primera clase del modelo (lo que significa que serán tratadas como bloques básicos de construcción) marcándolas con un estereotipo apropiado, como se ha hecho con la clase *Overflow* en la Figura 2.22.

Un *valor etiquetado* extiende las propiedades de un estereotipo de UML, permitiendo añadir nueva información en la especificación de ese estereotipo. Por ejemplo, si estamos trabajando en un producto que atraviesa muchas versiones a lo largo del tiempo, a menudo querremos registrar la versión y el autor de ciertas abstracciones críticas. La versión y el autor no son conceptos primitivos de UML. Pueden ser añadidos a cualquier bloque de construcción, como una clase, introduciendo nuevos valores etiquetados en ese bloque de construcción. En la Figura 2.22, por ejemplo, la clase *ColaDeEventos* se extiende indicando explícitamente su versión y su autor.

Una *restricción* extiende la semántica de un bloque de construcción de UML, permitiendo añadir nuevas reglas o modificar las existentes. Por ejemplo, quizás se deseé restringir la clase *ColaDeEventos* para que todas las adiciones se hagan en orden. Como se muestra en la Figura 2.22, se puede añadir una restricción que indique explícitamente esto para la operación *añadir*.

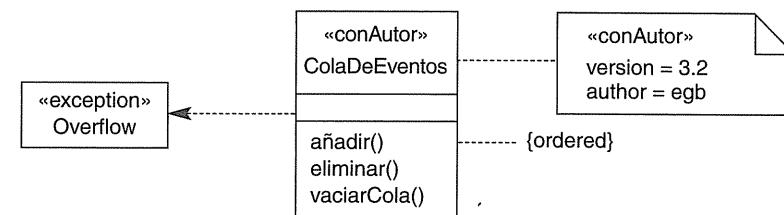


Figura 2.22: Mecanismos de extensibilidad.

En conjunto, estos tres mecanismos de extensibilidad permiten configurar y extender UML para las necesidades de un proyecto. Estos mecanismos también permiten a UML adaptarse a nuevas tecnologías de software, como la probable aparición de lenguajes de programación distribuida más potentes. Es posible añadir nuevos bloques de construcción, modificar la especificación de los existentes, e incluso cambiar su semántica. Naturalmente, es importante hacer esto de forma controlada, para que a través de estas extensiones se siga siendo fiel al propósito de UML, la comunicación de información.

Arquitectura

La necesidad de ver los sistemas complejos desde diferentes puntos de vista se discute en el Capítulo 1.

La visualización, especificación, construcción y documentación de un sistema con gran cantidad de software requiere que el sistema sea visto desde varias perspectivas. Diferentes usuarios (usuarios finales, analistas, desarrolladores, integradores de sistemas, encargados de las pruebas, encargados de la documentación técnica y jefes de proyectos) siguen diferentes agendas en relación con el proyecto, y cada uno mira al sistema de formas diferentes en diversos momen-

tos a lo largo de la vida del proyecto. La arquitectura de un sistema es quizás el artefacto más importante que puede emplearse para manejar estos diferentes puntos de vista y controlar el desarrollo iterativo e incremental de un sistema a lo largo de su ciclo de vida.

La arquitectura es el conjunto de decisiones significativas sobre:

- La organización de un sistema software.
- La selección de elementos estructurales y sus interfaces a través de los cuales se constituye el sistema.
- Su comportamiento, como se especifica en las colaboraciones entre esos elementos.
- La composición de esos elementos estructurales y de comportamiento en subsistemas progresivamente más grandes.
- El estilo arquitectónico que guía esta organización: los elementos estáticos y dinámicos y sus interfaces, sus colaboraciones y su composición.

La arquitectura software no tiene que ver solamente con la estructura y el comportamiento, sino también con el uso, la funcionalidad, el rendimiento, la capacidad de adaptación, la reutilización, la capacidad de ser comprendido, las restricciones económicas y tecnológicas y los compromisos entre alternativas, así como los aspectos estéticos.

El modelado de la arquitectura de un sistema se discute en el Capítulo 32.

Como ilustra la Figura 2.23, la arquitectura de un sistema con gran cantidad de software puede describirse mejor a través de cinco vistas interrelacionadas. Cada vista es una proyección de la organización y la estructura del sistema, centrada en un aspecto particular del mismo.

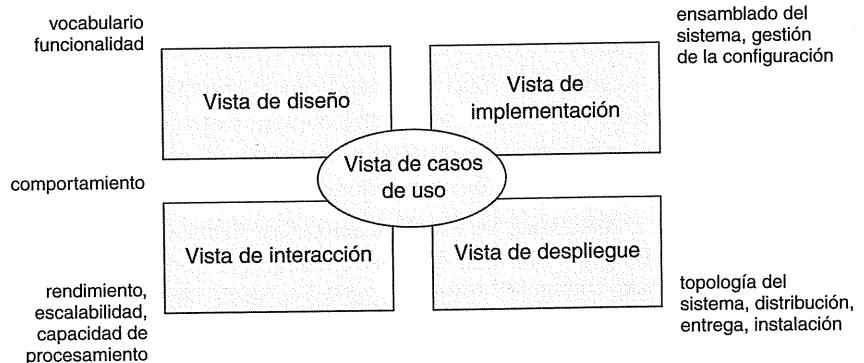


Figura 2.23: Modelo de la arquitectura de un sistema.

La *vista de casos de uso* de un sistema comprende los casos de uso que describen el comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas. Esta vista no especifica realmente la organización de un sistema software. Antes bien, existe para especificar las fuerzas que configuran la arquitectura del sistema. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de casos de uso; los aspectos dinámicos de esta vista se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades.

La *vista de diseño* de un sistema comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y su solución. Esta vista soporta principalmente los requisitos funcionales del sistema, entendiendo por ello los servicios que el sistema debería proporcionar a sus usuarios finales. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de clases y de objetos; los aspectos dinámicos se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades. El diagrama de estructura interna de una clase es particularmente útil.

La *vista de interacción* del sistema muestra el flujo de control entre sus diversas partes, incluyendo los posibles mecanismos de concurrencia y sincronización. Esta vista abarca principalmente el rendimiento, la escalabilidad y la capacidad de procesamiento del sistema. Con UML, los aspectos estáticos y dinámicos de esta vista se capturan con los mismos tipos de diagramas que en la vista de diseño, pero con mayor atención sobre las clases activas que controlan el sistema y los mensajes que fluyen entre ellas.

La *vista de implementación* de un sistema comprende los artefactos que se utilizan para ensamblar y poner en producción el sistema físico. Esta vista se ocupa principalmente de la gestión de configuraciones de las distintas versiones del sistema, a partir de archivos más o menos independientes que pueden ensamblarse de varias formas para producir un sistema en ejecución. También está relacionada con la correspondencia entre clases y componentes lógicos con los artefactos físicos. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de artefactos; los aspectos dinámicos de esta vista se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades.

La *vista de despliegue* de un sistema contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Esta vista se ocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema físico. Con UML, los aspectos estáticos de esta vista se capturan en los diagramas de despliegue; los aspectos dinámicos de esta vista se capturan en los diagramas de interacción, diagramas de estados y diagramas de actividades.

Cada una de estas cinco vistas puede existir por sí misma, de forma que diferentes usuarios puedan centrarse en las cuestiones de la arquitectura del sistema que más les interesen. Estas cinco vistas también interactúan entre sí: los nodos en la vista de despliegue contienen componentes de la vista de implementación que, a su vez, representan la realización física de las clases, interfaces, colaboraciones y clases activas de las vistas de diseño y de procesos. UML permite expresar cada una de estas cinco vistas.

Ciclo de vida del desarrollo de software

El Proceso Unificado de Rational se resume en el Apéndice B; un tratamiento más completo de este proceso se discute en El Proceso Unificado de Desarrollo de Software y en The Rational Unified Process.

UML es bastante independiente del proceso, lo que significa que no está ligado a ningún ciclo de vida de desarrollo de software particular. Sin embargo, para obtener el máximo beneficio de UML, se debería considerar un proceso que fuese:

- Dirigido por los casos de uso.
- Centrado en la arquitectura.
- Iterativo e incremental.

Dirigido por los casos de uso significa que los casos de uso se utilizan como un artefacto básico para establecer el comportamiento deseado del sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para la comunicación entre las personas involucradas en el proyecto.

Centrado en la arquitectura significa que la arquitectura del sistema se utiliza como un artefacto básico para conceptualizar, construir, gestionar y hacer evolucionar el sistema en desarrollo.

Un *proceso iterativo* es aquel que involucra la gestión de un flujo de versiones ejecutables. Un *proceso incremental* es aquel que implica la integración continua de la arquitectura del sistema para producir esas versiones, donde cada nuevo ejecutable incorpora mejoras incrementales sobre los otros. En conjunto, un proceso iterativo e incremental está *dirigido por el riesgo*, lo que significa que cada nueva versión se encarga de atacar y reducir los riesgos más significativos para el éxito del proyecto.

Este proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental puede descomponerse en fases. Una *fase* es el intervalo de tiempo entre dos hitos importantes del proceso, cuando se cumple un conjunto de objetivos bien definidos, se completan los artefactos y se toman las decisiones

sobre si pasar o no a la siguiente fase. Como se muestra en la Figura 2.24, hay cuatro fases en el ciclo de vida del desarrollo de software: concepción, elaboración, construcción y transición. En el diagrama, los flujos de trabajo se representan frente a estas fases, mostrando cómo varía a lo largo del tiempo el nivel de atención que una fase presta a un flujo de trabajo.

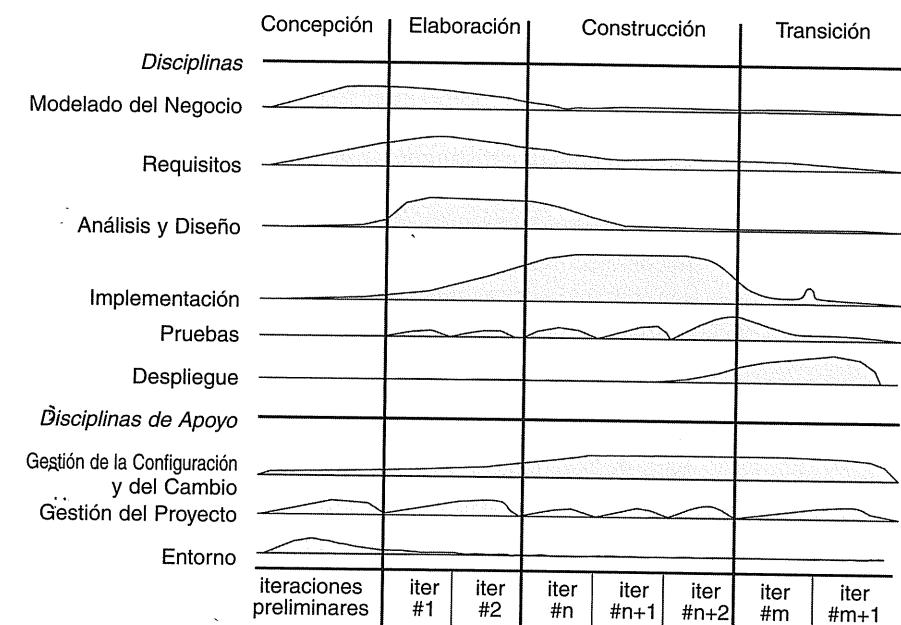


Figura 2.24: Ciclo de vida del desarrollo de software.

La *concepción* es la primera fase del proceso, cuando la idea inicial para el desarrollo se lleva al punto de estar (al menos internamente) suficientemente bien fundamentada para garantizar la entrada en la fase de elaboración.

La *elaboración* es la segunda fase del proceso, cuando se definen los requisitos del producto y su arquitectura. En esta fase se expresan con claridad los requisitos del sistema, son priorizados y se utilizan para crear una sólida base arquitectónica. Los requisitos de un sistema pueden variar desde enunciados de carácter general hasta criterios precisos de evaluación, especificando cada uno un comportamiento funcional o no funcional y proporcionando una referencia para las pruebas.

La *construcción* es la tercera fase del proceso, cuando el software se lleva desde una base arquitectónica ejecutable hasta su disponibilidad para la comunidad de usuarios. Aquí también, los requisitos del sistema y especialmente sus crite-

rios de evaluación son constantemente reexaminados frente a las necesidades del proyecto, y los recursos se asignan al proyecto de forma apropiada para acometer los riesgos.

La *transición* es la cuarta fase del proceso, cuando el software es puesto en las manos de la comunidad de usuarios. El proceso del software raramente termina aquí porque, incluso durante esta fase, el sistema es mejorado continuamente, se erradican errores de programación y se añaden características que no se incluían en las versiones anteriores.

Un elemento que distingue a este proceso y que afecta a las cuatro fases es una iteración. Una *iteración* es un conjunto bien definido de actividades, con un plan y unos criterios de evaluación bien establecidos, que acaba en un sistema que puede ejecutarse, probarse y evaluarse. El sistema ejecutable no tiene por qué ser entregado externamente. Como la iteración produce un producto ejecutable, después de cada iteración se puede juzgar el progreso y se pueden reevaluar los riesgos. Esto significa que el ciclo de vida del desarrollo de software puede caracterizarse por involucrar un flujo continuo de versiones ejecutables de la arquitectura del sistema con correcciones entre ellas, después de cada iteración, para mitigar los riesgos potenciales. Este énfasis en la arquitectura como un artefacto importante es el que conduce a UML a centrarse en el modelado de las diferentes vistas de la arquitectura de un sistema.



Capítulo 3 ¡HOLA, MUNDO!

En este capítulo

- Clases y artefactos.
- Modelos estáticos y modelos dinámicos.
- Conexiones entre modelos.
- Extensión de UML.

Brian Kernighan y Dennis Ritchie, creadores del lenguaje de programación C, señalan que “la única forma de aprender un nuevo lenguaje de programación es programando en él”. Lo mismo es cierto para UML. La única forma de aprender UML es escribiendo modelos en él.

El primer programa que muchos programadores escriben al enfrentarse a un nuevo lenguaje de programación suele ser uno sencillo, que implique poco más que escribir la cadena de caracteres “¡Hola, Mundo!”. Éste es un punto de comienzo razonable, porque el control de esta aplicación trivial proporciona una satisfacción inmediata. También se incluye toda la infraestructura necesaria para poder ejecutar algo.

Aquí es donde comenzamos con UML. El modelado de “¡Hola, Mundo!” será uno de los usos más sencillos de UML que encontraremos. Sin embargo, esta aplicación es engañosamente fácil porque por debajo de ella hay ciertos mecanismos interesantes que la hacen funcionar. Estos mecanismos pueden modelarse fácilmente con UML, lo que proporciona una perspectiva más rica de esta sencilla aplicación.

Abstracciones clave

En Java, el *applet* para imprimir “¡Hola, Mundo!” en un navegador de Internet es bastante sencillo:

```
import java.awt.Graphics;
class HolaMundo extends java.applet.Applet {
    public void paint (Graphics g) {
        g.drawString ("¡Hola, Mundo!", 10, 10);
    }
}
```

La primera línea de código:

```
import java.awt.Graphics;
```

hace que la clase *Graphics* esté disponible directamente para el código que sigue. El prefijo *java.awt* especifica el paquete Java en el cual se encuentra la clase *Graphics*.

La segunda línea de código:

```
class HolaMundo extends java.applet.Applet {
```

introduce una nueva clase llamada *HolaMundo* y especifica que es una subclase de *Applet*, la cual se encuentra en el paquete *java.applet*.

Las tres siguientes líneas de código:

```
public void paint (Graphics g) {
    g.drawString ("¡Hola, Mundo!", 10, 10);
}
```

declaran una operación llamada *paint*, cuya implementación invoca a otra operación, llamada *drawString*, responsable de imprimir la cadena de caracteres “¡Hola, Mundo!” en las coordenadas indicadas. Al estilo habitual orientado a objetos, *drawString* es una operación sobre un parámetro *g*, cuyo tipo es la clase *Graphics*.

Las clases se discuten en los Capítulos 4 y 9.

El modelado de esta aplicación en UML es directo. Como muestra la Figura 3.1, se puede representar la clase *HolaMundo* gráficamente como un ícono rectangular. Ahí se representa su operación *paint*, con los parámetros formales omitidos y su implementación especificada en la nota adjunta.

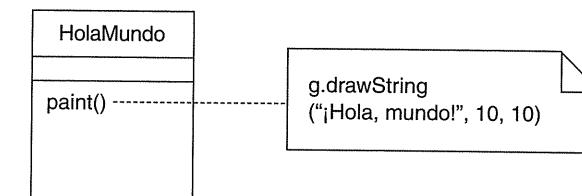


Figura 3.1: Abstracciones clave para *HolaMundo*.

Nota: UML no es un lenguaje de programación visual, aunque, como se muestra en la figura, UML permite (pero no requiere) un alto acoplamiento con varios lenguajes de programación como Java. UML está diseñado para permitir transformar los modelos en código y permitir ingeniería inversa del código a modelos. Algunas cosas se escriben mejor en la sintaxis de un lenguaje de programación textual (por ejemplo, las expresiones matemáticas), mientras que otras se visualizan mejor gráficamente en UML (por ejemplo, jerarquías de clases).

Este diagrama de clases captura los aspectos básicos de la aplicación “¡Hola, Mundo!”, pero deja fuera varias cosas. Como especifica el código precedente, otras dos clases (*Applet* y *Graphics*) intervienen en la aplicación y cada una se utiliza de forma diferente. La clase *Applet* se utiliza como parente de *HolaMundo*, y la clase *Graphics* se utiliza en la firma e implementación de una de sus operaciones, *paint*. Estas clases y sus diferentes relaciones con la clase *HolaMundo* se pueden representar en un diagrama de clases, como se muestra en la Figura 3.2.

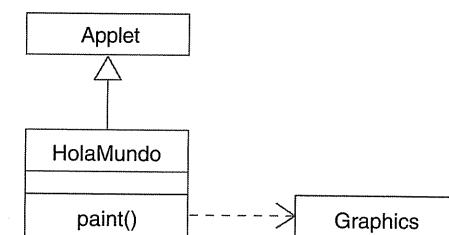


Figura 3.2: Vecinos inmediatos alrededor de *HolaMundo*.

Las relaciones se discuten en los Capítulos 5 y 10.

Las clases *Applet* y *Graphics* se representan gráficamente como rectángulos. No se muestran operaciones para ninguna de ellas y, por ello, sus iconos se omiten. La línea con la punta de flecha vacía dirigida desde *HolaMundo* hasta *Applet* representa una generalización, lo que en este caso significa que

HolaMundo es hija de Applet. La línea discontinua desde HolaMundo hasta Graphics representa una relación de dependencia, lo que significa que HolaMundo utiliza a Graphics.

Aquí no acaba el marco bajo el que se construye HolaMundo. Si estudiamos las bibliotecas de Java para Applet y Graphics, descubriremos que ambas clases son parte de una jerarquía mayor. Siguiendo la pista sólo de las clases que Applet extiende e implementa, se puede generar otro diagrama de clases, que se muestra en la Figura 3.3.

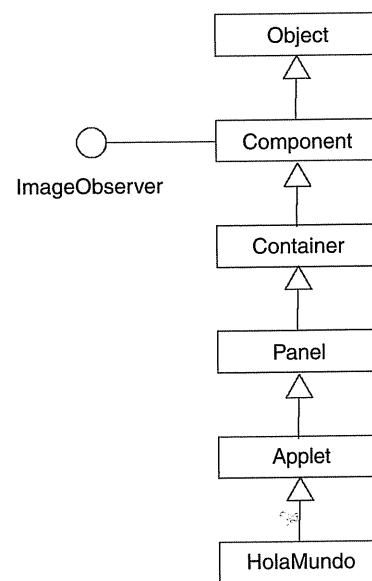


Figura 3.3: Jerarquía de herencia de HolaMundo.

Nota: Esta figura es un buen ejemplo de un diagrama generado por ingeniería inversa sobre un sistema existente. La ingeniería inversa es la creación de un modelo a partir de código.

Esta figura deja claro que HolaMundo es sólo una clase hoja (no tiene subclases) de una jerarquía mayor de clases. HolaMundo es hija de Applet; Applet es hija de Panel; Panel es hija de Container; Container es hija de Component; y Component es hija de Object, la cual es la clase padre de todas las clases en Java. Este modelo se corresponde con la biblioteca Java (cada clase hija extiende a alguna clase padre).

Las interfaces se discuten en el Capítulo 11.

La relación entre ImageObserver y Component es un poco diferente, y el diagrama de clases refleja esta diferencia. En la biblioteca Java, ImageObserver es una interfaz, lo cual, entre otras cosas, significa que no tiene implementación, y que requiere que otras clases la implementen. El hecho de que Component implemente a ImageObserver se representa por la línea continua que conecta el rectángulo (Component) con el círculo que representa a la interfaz implementada (ImageObserver).

Como muestran estas figuras, HolaMundo colabora directamente sólo con dos clases (Applet y Graphics), y estas dos clases sólo son una pequeña parte de una biblioteca mayor de clases Java predefinidas. Para gestionar esta gran colección, Java organiza sus interfaces y clases en diferentes paquetes. El paquete raíz en el entorno Java se llama, como era de esperar, java. Dentro de él hay varios paquetes, que contienen otros paquetes, interfaces y clases. Object se encuentra en el paquete lang y, por ello, su nombre completo es java.lang.Object. Análogamente, Panel, Container y Component están en awt; la clase Applet se encuentra en applet. La interfaz ImageObserver está en el paquete image, el cual a su vez se encuentra en el paquete awt, de forma que su nombre completo es la larga cadena de caracteres java.awt.image.ImageObserver.

Se puede ver esta organización de paquetes en el diagrama de clases de la Figura 3.4. Como se muestra en esta figura, los paquetes se representan en UML como carpetas. Los paquetes se pueden anidar, y las líneas discontinuas dirigidas representan dependencias entre esos paquetes. Por ejemplo, HolaMundo depende del paquete java.applet, y java.applet depende del paquete java.awt.

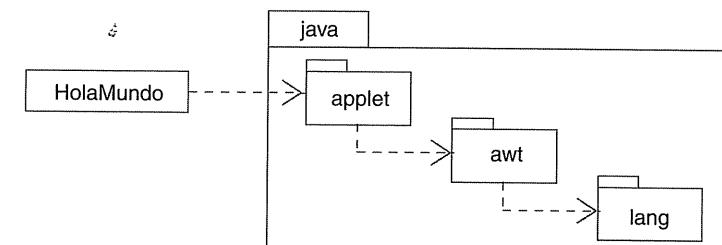


Figura 3.4: Empaquetamiento de HolaMundo.

Mecanismos

Los patrones y los frameworks se discuten en el Capítulo 29.

Lo más difícil para dominar una biblioteca tan rica como la de Java es aprender cómo colaboran sus partes. Por ejemplo, ¿cómo llega a invocarse la operación paint de HolaMundo? ¿Qué operaciones hay que utilizar si se desea cambiar

Los procesos y los hilos se discuten en el Capítulo 23.

el comportamiento de este *applet*, para conseguir, por ejemplo, que imprima la cadena de caracteres en otro color? Para responder a estas y otras cuestiones, hay que tener un modelo conceptual de la forma en que estas clases colaboran dinámicamente.

El estudio de la biblioteca Java revela que la operación *paint* de *HolaMundo* se hereda de *Component*. Esto aún plantea la cuestión de cómo se invoca la operación. La respuesta es que *paint* se llama como parte de la ejecución del hilo que incluye al *applet*, como ilustra la Figura 3.5.

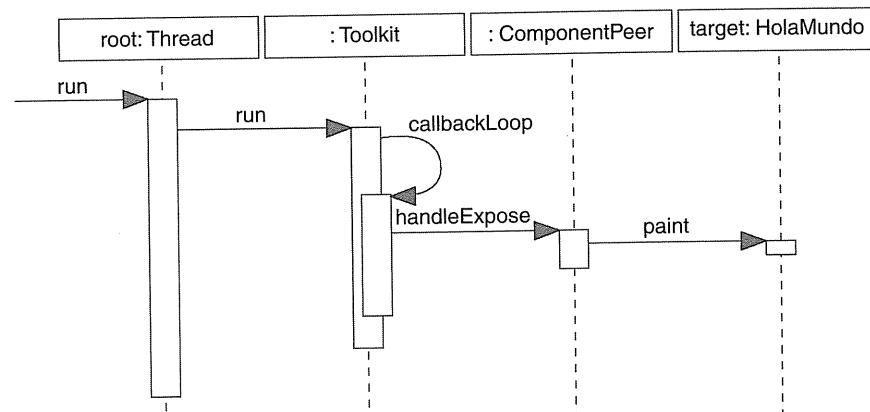


Figura 3.5: Mecanismo de dibujo.

Las instancias se discuten en el Capítulo 13.

Esta figura muestra la colaboración de varios objetos, incluida una instancia de la clase *HolaMundo*. Los otros objetos son parte del entorno Java y, por ello, la mayoría de ellos subyace a los *applets* que creamos. Aquí se muestra una colaboración entre objetos que puede aplicarse muchas veces. Cada columna muestra un rol en la colaboración, es decir, un papel que puede ser desempeñado por un objeto diferente en cada ejecución. En UML, los roles se representan como las clases, excepto en el hecho de que tienen nombres de rol y tipos. Los dos roles intermedios del diagrama son anónimos, porque basta con sus tipos para identificarlos en la colaboración (pero los dos puntos y la ausencia de una línea de subrayado los marca como roles). El Thread inicial se llama *root*, y el rol *HolaMundo* tiene el nombre *target* conocido por el rol *ComponentPeer*.

Los diagramas de secuencia se discuten en el Capítulo 19.

Se puede modelar el orden de los eventos utilizando un diagrama de secuencia, como se muestra en la Figura 3.5. Ahí, la secuencia comienza activando el objeto *Thread*, que a su vez llama a la operación *run* de *Toolkit*. El objeto *Toolkit* entonces llama a una de sus propias operaciones (*callbackLoop*),

la cual invoca a la operación *handleExpose* de *ComponentPeer*. El objeto *ComponentPeer* invoca entonces la operación *paint* de su receptor. El objeto *ComponentPeer* asume que su receptor es un *Component*, pero en este caso el receptor es realmente una subclase de *Component* (llamada *HolaMundo*) y, por ello, la operación *paint* de *HolaMundo* se invoca de forma polimórfica.

Artefactos

“¡Hola, Mundo!” se implementa como un *applet*, así que nunca se encontrará aislada, sino formando parte de una página web. La ejecución del *applet* comienza cuando se abre la página que lo contiene, desencadenada por algún mecanismo del navegador que activa al objeto *Thread* del *applet*. Sin embargo, no es la clase *HolaMundo* la que realmente forma parte de la página web. En vez de ella, encontramos una forma binaria de la clase, creada por un compilador de Java que transforma el código fuente que representa la clase en un artefacto que puede ejecutarse. Esto sugiere una perspectiva muy diferente del sistema. Mientras todos los diagramas anteriores representaban una vista lógica del *applet*, ahora se considera una vista de los componentes físicos del *applet*.

Podemos modelar esta vista física con un diagrama de artefactos, como se muestra en la Figura 3.6.

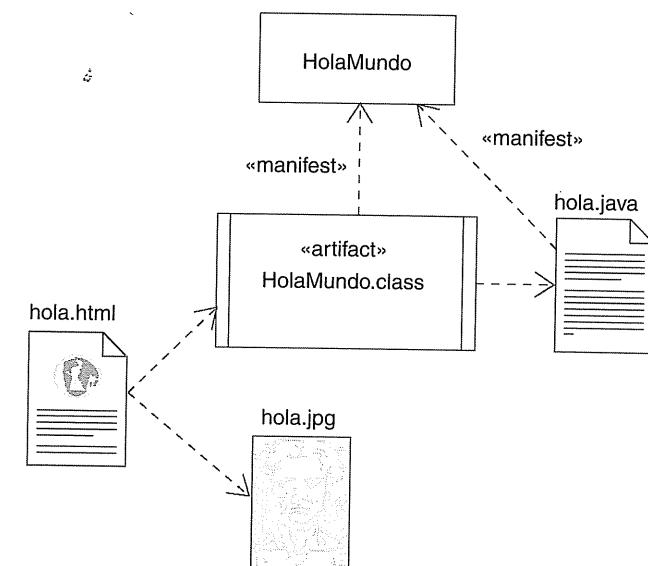


Figura 3.6: Artefactos de *HolaMundo*.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

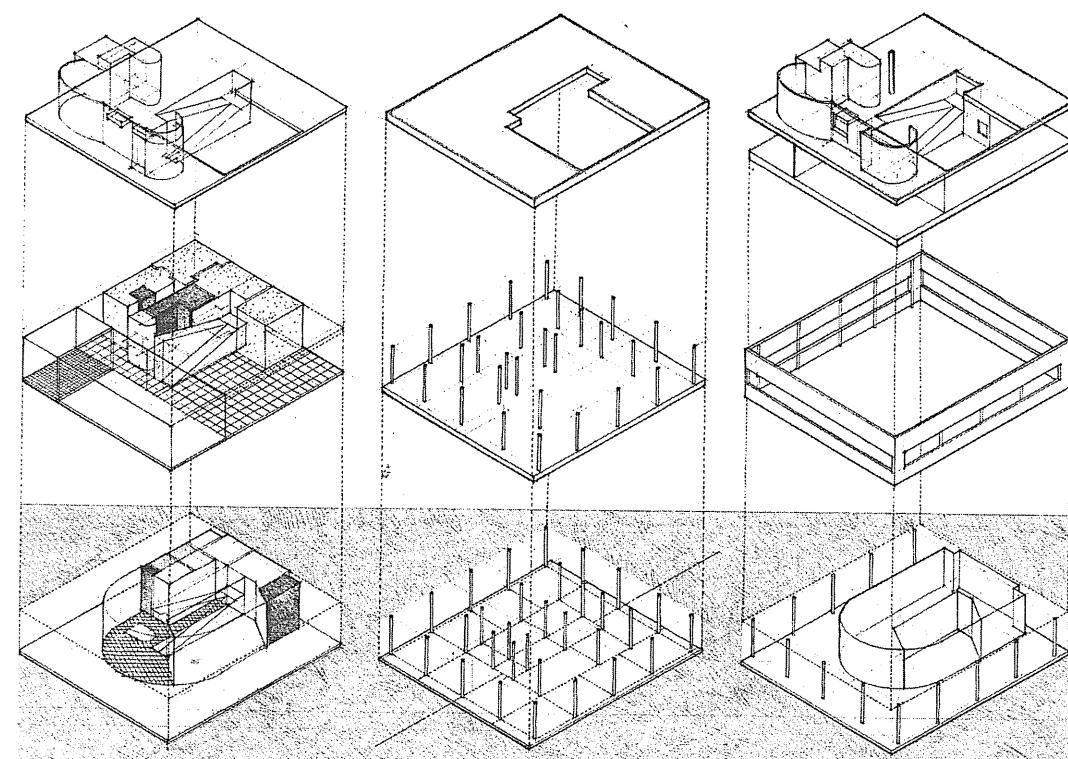
La clase lógica HolaMundo se muestra arriba como un rectángulo de clase. Cada uno de los iconos de esta figura representa un artefacto UML en la vista de implementación del sistema. Un artefacto es una representación física, como un archivo. El artefacto llamado `hola.java` representa el código fuente para la clase HolaMundo, siendo un archivo que puede ser manipulado por los entornos de desarrollo y las herramientas de gestión de configuraciones. Un compilador de Java puede transformar este código fuente en el *applet* binario `hola.class`, que es ejecutable por la máquina virtual Java de un computador. Tanto el código fuente como el applet binario manifiestan (implementan físicamente) la clase lógica. Esto se representa con las flechas discontinuas con la palabra clave `<<manifest>>`.

El ícono para un artefacto es un rectángulo con la palabra clave `<<artifact>>` sobre el nombre. El applet binario `HolaMundo.class` es una variación de este símbolo básico, con las líneas verticales dobles, que indican que es un artefacto ejecutable (como una clase activa). El ícono para el artefacto `hola.java` se ha reemplazado por un ícono definido por el usuario, que representa un archivo de texto. El ícono para la página web `hola.html` se ha personalizado igualmente, extendiendo la notación de UML. Como indica esta figura, esta página web tiene otro artefacto, `hola.jpg`, que se representa por un ícono de artefacto definido por el usuario, en este caso una miniatura de la imagen. Como estos últimos tres artefactos utilizan símbolos gráficos definidos por el usuario, sus nombres se encuentran fuera del ícono. Las dependencias entre los artefactos se muestran con flechas discontinuas.

Nota: Las relaciones entre la clase (HolaMundo), su código fuente (`hola.java`) y su código objeto (`HolaMundo.class`) no se suelen modelar explícitamente, aunque a veces esto es útil para ver la configuración física de un sistema. Por otro lado, es frecuente visualizar la organización de un sistema basado en la Web mediante diagramas de artefactos, para modelar sus páginas y otros artefactos ejecutables.



Parte 2 MODELADO ESTRUCTURAL BÁSICO



En este capítulo

- Clases, atributos, operaciones y responsabilidades.
- Modelado del vocabulario de un sistema.
- Modelado de la distribución de responsabilidades en un sistema.
- Modelado de cosas que no son software.
- Modelado de tipos primitivos.
- Construir abstracciones de calidad.

Las clases son los bloques de construcción más importantes de cualquier sistema orientado a objetos. Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces.

Las características avanzadas de las clases se discuten en el Capítulo 9.

Las clases se utilizan para capturar el vocabulario del sistema en desarrollo. Estas clases pueden incluir abstracciones que formen parte del dominio del problema, así como clases que constituyan una implementación. Las clases se pueden utilizar para representar cosas que sean software, hardware o cosas puramente conceptuales.

Las clases bien estructuradas están bien delimitadas y participan de una distribución equilibrada de responsabilidades en el sistema.

Introducción

El modelado de un sistema implica identificar las cosas que son importantes desde un punto de vista particular. Estas cosas forman el vocabulario del sistema que se está modelando. Por ejemplo, si estamos construyendo una casa, las

paredes, puertas, ventanas, armarios y luces son algunas de las cosas importantes desde la perspectiva de un propietario. Cada una de estas cosas puede ser separada del resto.

Cada una de ellas también tiene un conjunto de propiedades. Las paredes tienen una altura y una anchura y son sólidas. Las puertas también tienen una altura y una anchura y son sólidas, pero tienen el comportamiento adicional que permite abrirse en una dirección. Las ventanas son similares a las puertas en que ambas son aberturas en las paredes, pero las ventanas y las puertas tienen propiedades ligeramente diferentes. Las ventanas se diseñan normalmente (aunque no siempre) para que se pueda mirar a través de ellas en vez de atravesarlas.

Las paredes, puertas y ventanas no suelen encontrarse aisladas, así que también hay que considerar cómo encajan juntas las instancias específicas de ellas. Las cosas que se han identificado y las relaciones que se establecen entre ellas se verán afectadas por el uso que se pretende dar a cada habitación de la casa, cómo se espera que sea la circulación de una habitación a otra y el estilo y el aspecto general que se quiere lograr con esa disposición.

Las personas implicadas estarán interesadas en diferentes cosas. Por ejemplo, los fontaneros que ayudan a construir la casa se interesará por cosas como desagües, sifones y conductos de ventilación. El propietario no estará preocupado necesariamente por estas cuestiones, excepto en el caso de que interactúen con las cosas en las que fija su atención, como dónde se puede colocar un desagüe en un suelo o dónde puede romper la estética del techo un conductor de ventilación.

Los objetos se discuten en el Capítulo 13.

En UML, todas estas cosas se modelan como clases. Una clase es una abstracción de las cosas que forman parte del vocabulario. Una clase no es un objeto individual, sino que representa un conjunto de objetos. Así, se puede pensar conceptualmente en “pared” como una clase de objetos con ciertas propiedades comunes, como altura, longitud, grosor, si es maestra o no, etc. También se puede pensar en instancias individuales de paredes, como “la pared de la esquina sur de mi despacho”.

En el terreno del software, muchos lenguajes de programación soportan directamente el concepto de clase. Esto es excelente, porque ello significa que a menudo las abstracciones que se crean pueden trasladarse directamente a un lenguaje de programación, incluso si éstas lo son de cosas que no son software, como “cliente”, “negocio” o “conversación”.

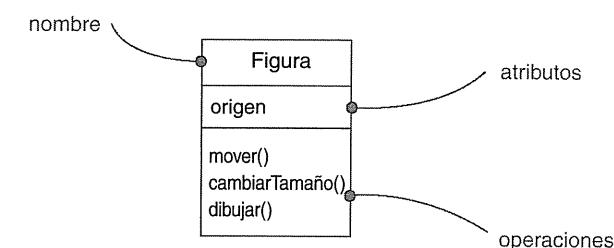


Figura 4.1: Clases.

UML también proporciona una representación gráfica de las clases, como muestra la Figura 4.1. Esta notación permite visualizar una abstracción independientemente de cualquier lenguaje de programación específico y de forma que permite resaltar las partes más importantes de una abstracción: su nombre, sus atributos y sus operaciones.

Términos y conceptos

Una *clase* es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Gráficamente, una clase se representa como un rectángulo.

Nombres

El nombre de una clase debe ser único dentro del paquete que la contiene, como se discute en el Capítulo 12.

Cada clase ha de tener un nombre que la distinga de otras clases. Un *nombre* es una cadena de texto. Ese nombre solo se denomina *nombre simple*; un *nombre calificado* consta del nombre de la clase precedido por el nombre del paquete en el que se encuentra. Una clase puede dibujarse mostrando sólo su nombre, como se muestra en la Figura 4.2.

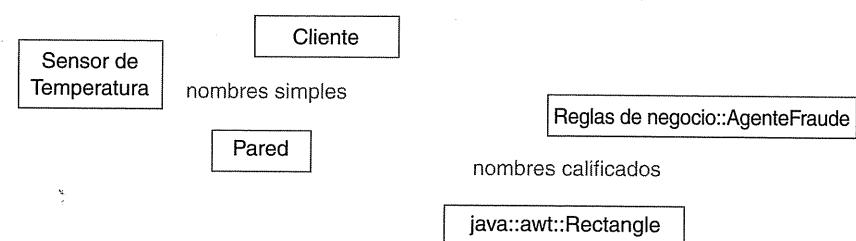


Figura 4.2: Nombres simples y calificados.

Nota: El nombre de una clase puede ser texto formado por cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se utilizan para separar el nombre de una clase y el del paquete que la contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de clase son nombres cortos o expresiones nominales extraídos del vocabulario del sistema que se está modelando. Normalmente, en el nombre de la clase se pone en mayúsculas la primera letra de cada palabra, como en Cliente o SensorDeTemperatura.

Los atributos están relacionados con la semántica de la agregación, como se discute en el Capítulo 10.

Atributos

Un *atributo* es una propiedad de una clase identificada con un nombre, que describe un rango de valores que pueden tomar las instancias de la propiedad. Una clase puede tener cualquier número de atributos o no tener ninguno. Un atributo representa alguna propiedad del elemento que se está modelando que es compartida por todos los objetos de esa clase. Por ejemplo, toda pared tiene una altura, una anchura y un grosor; se podrían modelar los clientes de forma que cada uno tenga un nombre, una dirección, un teléfono y una fecha de nacimiento. Un atributo es, por tanto, una abstracción de un tipo de dato o estado que puede incluir un objeto de la clase. En un momento dado, un objeto de una clase tendrá valores específicos para cada uno de los atributos de su clase. Gráficamente, los atributos se listan en un compartimento justo debajo del nombre de la clase. Los atributos se pueden representar mostrando sólo sus nombres, como se ve en la Figura 4.3.

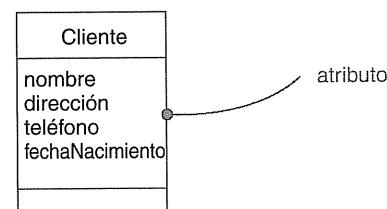


Figura 4.3: Atributos.

Nota: El nombre de un atributo puede ser texto, igual que el nombre de una clase. En la práctica, el nombre de un atributo es un nombre corto o una expresión nominal que representa alguna propiedad de la clase que lo engloba. Normalmente, se pone en mayúsculas la primera letra de cada palabra de un atributo, excepto la primera letra, como en nombre o esMaestra.

También se pueden especificar otras características de un atributo, como indicar si es de sólo lectura o está compartido por todos los objetos de la clase, como se discute en el Capítulo 9.

Un atributo se puede especificar aún más indicando su clase y quizás un valor inicial por defecto, como se muestra en la Figura 4.4.

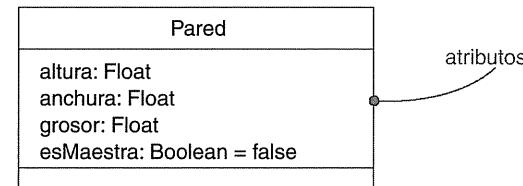


Figura 4.4: Atributos y sus clases.

Operaciones

Se puede especificar aún más la implementación de una operación utilizando una nota, como se describe en el Capítulo 6, o utilizando un diagrama de actividades, como se discute en el Capítulo 20.

Una *operación* es la implementación de un servicio que puede ser requerido a cualquier objeto de la clase para que muestre un comportamiento. En otras palabras, una operación es una abstracción de algo que se puede hacer a un objeto y que es compartido por todos los objetos de la clase. Una clase puede tener cualquier número de operaciones o ninguna. Por ejemplo, en una biblioteca de ventanas como la del paquete awt de Java, todos los objetos de la clase Rectangle pueden ser movidos, redimensionados o consultados sobre sus propiedades. A menudo (pero no siempre), la invocación de una operación sobre un objeto cambia los datos o el estado del objeto. Gráficamente, las operaciones se listan en un compartimento justo debajo de los atributos de la clase. Las operaciones se pueden representar mostrando sólo sus nombres, como se ilustra en la Figura 4.5.

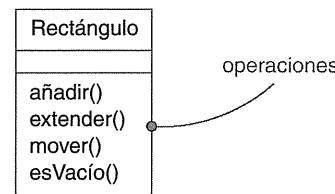


Figura 4.5: Operaciones.

Nota: El nombre de una operación puede ser texto, igual que el nombre de una clase. En la práctica, el nombre de una operación es un verbo corto o una expresión verbal que representa un comportamiento de la clase que la contiene. Normalmente, se pone en mayúsculas la primera letra de cada palabra en el nombre de una operación excepto la primera letra, como en mover o estaVacio.

También se pueden especificar otras características de una operación, como indicar si es polimórfica o constante, o especificar su visibilidad, como se discute en el Capítulo 9.

Una operación se puede especificar indicando su firma, la cual incluye el nombre, tipo y valores por defecto de todos los parámetros y (en el caso de las funciones) un tipo de retorno, como se muestra en la Figura 4.6.

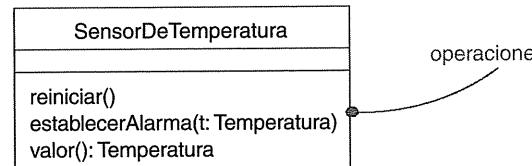


Figura 4.6: Operaciones y sus firmas.

Organización de atributos y operaciones

Cuando se dibuja una clase, no hay por qué mostrar todos los atributos y todas las operaciones de una vez. De hecho, en la mayoría de los casos, no se puede (hay demasiados para ponerlos en una figura) y posiblemente no se debería hacer (probablemente sólo será relevante un subconjunto de esos atributos y operaciones para una vista determinada). Por estas razones, se puede abreviar una clase, es decir, se puede decidir mostrar sólo algunos, o incluso ninguno, de sus atributos y operaciones. Un comportamiento vacío no significa necesariamente que no haya operaciones o atributos, sólo que se ha decidido no mostrarlos. Se puede especificar explícitamente que hay más atributos o propiedades que los mostrados acabando la lista con puntos suspensivos (...). También se puede suprimir por completo el comportamiento, en cuyo caso no se puede afirmar si hay atributos u operaciones, o cuántos hay.

Los estereotipos se discuten en el Capítulo 6.

Para organizar mejor las listas largas de atributos y operaciones, se pueden utilizar estereotipos para anteponer a cada grupo una categoría descriptiva, como se muestra en la Figura 4.7.

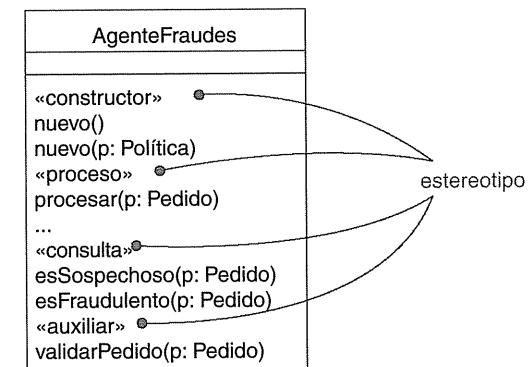


Figura 4.7: Estereotipos para las características de las clases.

Las responsabilidades representan un ejemplo de un estereotipo definido, como se discute en el Capítulo 6.

El modelado de la semántica de una clase se discute en el Capítulo 9.

También se pueden representar las responsabilidades de una clase en una nota, como se discute en el Capítulo 6.

Responsabilidades

Una *responsabilidad* es un contrato o una obligación de una clase. Al crear una clase, se está expresando que todos los objetos de esa clase tienen el mismo tipo de estado y el mismo tipo de comportamiento. A un nivel más abstracto, estos atributos y operaciones son simplemente las características por medio de las cuales se llevan a cabo las responsabilidades de la clase. Una clase Pared es responsable de saber sobre altura, anchura y grosor; una clase InspectorDeFraudes, que se podría encontrar en una aplicación de tarjetas de crédito, es responsable de procesar pedidos y determinar si son legales, sospechosos o fraudulentos; una clase SensorDeTemperatura es responsable de medir la temperatura y disparar una alarma si ésta alcanza un punto determinado.

Al modelar clases, un buen punto de partida consiste en especificar las responsabilidades de los elementos del vocabulario. Técnicas como las tarjetas CRC y el análisis basado en casos de uso son especialmente útiles aquí. Una clase puede tener cualquier número de responsabilidades, aunque, en la práctica, cada clase bien estructurada tiene al menos una responsabilidad y a lo sumo unas pocas. Al ir refinando los modelos, esas responsabilidades se traducirán en el conjunto de atributos y operaciones que mejor satisfagan las responsabilidades de la clase.

Gráficamente, las responsabilidades se pueden expresar en un compartimento separado al final del icono de la clase, como se muestra en la Figura 4.8.

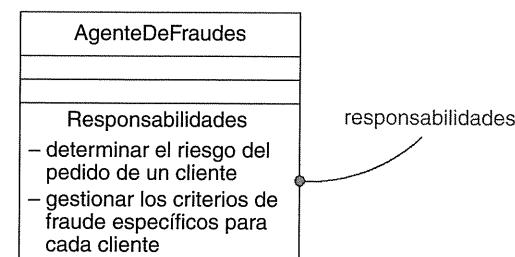


Figura 4.8: Responsabilidades.

Nota: Las responsabilidades son sólo texto libre. En la práctica, una responsabilidad aislada se escribe como una expresión, una frase o (como mucho) un párrafo corto.

Los conceptos avanzados de las clases se discuten en el Capítulo 9.

Otras características

Los atributos, las operaciones y las responsabilidades son las características más comunes que se necesitan cuando se crean abstracciones. De hecho, para la mayoría de los modelos que se construyan, la forma básica de estas tres características será todo lo necesario para expresar la semántica más importante de las clases. A veces, sin embargo, se necesitará visualizar o especificar otras características, como la visibilidad de atributos y operaciones individuales; características específicas del lenguaje para una operación, por ejemplo, si es polimórfica o constante; o incluso las excepciones que pueden producir o manejar los objetos de la clase. Estas y muchas otras características se pueden expresar en UML, pero se tratan como conceptos avanzados.

Las interfaces se discuten en el Capítulo 11.

Al construir modelos, uno descubre pronto que casi todas las abstracciones que crea son algún tipo de clase. A veces se quiere separar la implementación de una clase de su especificación, y esto se puede expresar en UML con interfaces.

La estructura interna se discute en el Capítulo 15.

Cuando se empieza a diseñar la implementación de una clase, se necesita modelar su estructura interna como un conjunto de partes conectadas. Se puede expandir una clase de alto nivel a través de varias capas de estructura interna para obtener el diseño resultante.

Las clases activas, los componentes y los nodos se discuten en los Capítulos 23, 25 y 27, y los artefactos se discuten en el Capítulo 26.

Cuando se empieza a construir modelos más complejos, uno también se tropieza con los mismos tipos de entidades una y otra vez, tales como clases que representan procesos concurrentes e hilos, o clasificadores que representan elementos físicos, como *applets*, Java Beans, archivos, páginas web y hardware. Como estos tipos de entidades son tan frecuentes y representan abstracciones arquitectónicas importantes, UML proporciona las clases activas (para representar procesos e hilos) y clasificadores, como los artefactos (para representar componentes software físicos) y nodos (para representar dispositivos hardware).

Los diagramas de clases se discuten en el Capítulo 8.

Por último, las clases rara vez se encuentran solas. Al construir modelos, uno se centra más bien en grupos de clases que interactúan entre sí. En UML, estas sociedades de clases forman colaboraciones y normalmente se representan en diagramas de clases.

Técnicas comunes de modelado

Modelado del vocabulario de un sistema

Normalmente, las clases se utilizarán para modelar abstracciones extraídas del problema que se intenta resolver o de la tecnología que se utiliza para implementar una solución a ese problema. Cada una de estas abstracciones es parte del vocabulario del sistema, es decir, ellas representan el conjunto de los elementos que son importantes para los usuarios y los implementadores.

Para los usuarios, la mayoría de las abstracciones no son difíciles de identificar, ya que, normalmente, las extraen de los objetos que ellos ya usan para describir su sistema. Técnicas como las tarjetas CRC y el análisis basado en casos de uso son formas excelentes de ayudar a los usuarios a encontrar estas abstracciones. Para los implementadores, estas abstracciones son normalmente los elementos de la tecnología que forman parte de la solución.

Para modelar el vocabulario de un sistema:

- Hay que identificar aquellas cosas que utilizan los usuarios o programadores para describir el problema o la solución. Se pueden utilizar tarjetas CRC y análisis basado en casos de uso para ayudar a encontrar esas abstracciones.
- Para cada abstracción, hay que identificar un conjunto de responsabilidades. Hay que asegurarse de que cada clase está claramente definida y que hay un buen reparto de responsabilidades entre todas las clases.
- Hay que proporcionar los atributos y operaciones necesarios en cada clase para cumplir esas responsabilidades.

La Figura 4.9 representa un conjunto de clases extraídas de un sistema de venta, incluyendo Cliente, Pedido y Producto. Esta figura incluye algunas otras abstracciones relacionadas extraídas del vocabulario del problema, como Envío (para seguir la pista a los pedidos), Factura (para facturar pedidos) y Almacén (donde se encuentran los productos antes del envío). También hay una abstracción relacionada con la solución, Transacción, que se aplica a los pedidos y a los envíos.

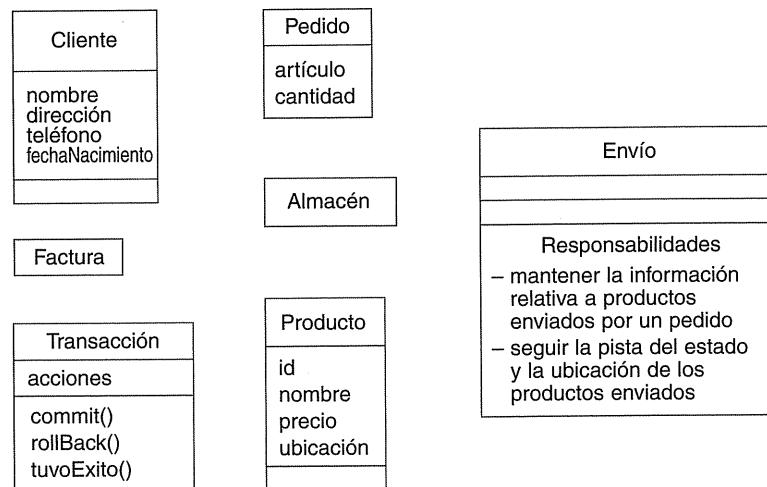


Figura 4.9: Modelado del vocabulario de un sistema.

Los paquetes se discuten en el Capítulo 12.

Al ir aumentando de tamaño los modelos, muchas de las clases que se encuentran tienden a agruparse en grupos relacionados conceptual y semánticamente. En UML se pueden utilizar los paquetes para modelar estos grupos de clases.

El modelado del comportamiento se discute en las Partes 4 y 5.

La mayoría de los modelos rara vez serán completamente estáticos. Más bien, la mayoría de las abstracciones del vocabulario del sistema interactuarán entre sí de forma dinámica. En UML hay varias formas de modelar este comportamiento dinámico.

Modelado de la distribución de responsabilidades en un sistema

Una vez que se ha modelado algo más que un pequeño número de clases, es necesario asegurarse de que las abstracciones proporcionan un conjunto equilibrado de responsabilidades. Esto significa que no deseamos que ninguna clase sea demasiado grande ni demasiado pequeña. Cada clase debería hacer bien una única cosa. Si se abstraen clases demasiado grandes, observaremos que los modelos son difíciles de cambiar y no muy reutilizables. Si se abstraen clases demasiado pequeñas, acabaremos con muchas más abstracciones de las que se pueden manejar o comprender razonablemente. UML se puede utilizar para ayudar a visualizar y especificar este reparto de responsabilidades.

Para modelar la distribución de responsabilidades en un sistema:

- Hay que identificar un conjunto de clases que colaboren entre sí para llevar a cabo algún comportamiento.
- Hay que identificar un conjunto de responsabilidades para cada una de esas clases.
- Hay que mirar este conjunto de clases como un todo, dividir las clases con demasiadas responsabilidades en abstracciones más pequeñas, fusionar las clases pequeñas con responsabilidades triviales en otras mayores, y reubicar las responsabilidades para que cada abstracción se mantenga razonablemente coherente.
- Hay que considerar la forma en que esas clases colaboran entre sí, y redistribuir sus responsabilidades adecuadamente, para que ninguna clase dentro de una colaboración haga demasiado o muy poco.

Las colaboraciones se discuten en el Capítulo 28.

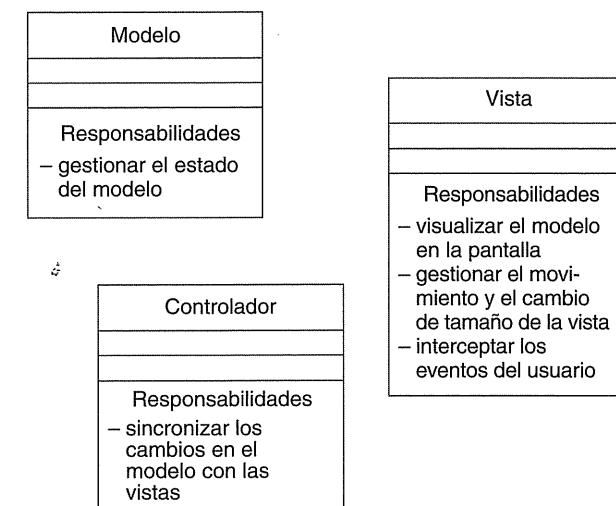


Figura 4.10: Modelado de la distribución de responsabilidades en un sistema.

Este conjunto de clases forma un patrón, como se discute en el Capítulo 29.

Por ejemplo, la Figura 4.10 representa un conjunto de clases obtenidas de Smalltalk, que muestra la distribución de responsabilidades entre las clases Modelo, Vista y Controlador. Obsérvese cómo estas clases colaboran de forma que ninguna clase hace mucho o muy poco.

Modelado de cosas que no son software

A veces, las cosas que se modelan pueden no tener un equivalente en el software. Por ejemplo, la gente que emite facturas y los robots que empaquetan automáticamente los pedidos de envío de un almacén pueden ser parte del flujo de trabajo que se modela en un sistema de ventas. Sin embargo, la aplicación podría no tener ningún software que los representase (al contrario que los clientes en el ejemplo anterior, ya que el sistema probablemente necesitará mantener información sobre ellos).

Para modelar las cosas que no son software:

- Hay que modelar la cosa que se está abstrayendo como una clase.
- Si se quieren distinguir estas cosas de los bloques de construcción de UML, hay que crear un nuevo bloque de construcción utilizando estereotipos para especificar la nueva semántica y proporcionar una representación visual que lo caracterice.
- Si lo que se está modelando es algún tipo de hardware que a su vez contiene software, habría que plantearse modelarlo como un tipo de nodo, de forma que más tarde sea posible completar su estructura.

Nota: UML tiene como objetivo principal el modelado de sistemas con gran cantidad de software, aunque, en conjunción con lenguajes textuales de modelado de hardware, como VHDL, UML puede ser bastante expresivo para modelar sistemas hardware. El OMG también ha producido una extensión de UML llamada SysML destinada al modelado de sistemas.

Los estereotipos se discuten en el Capítulo 6.

Los nodos se discuten en el Capítulo 27.

Los elementos externos se modelan a menudo como actores, como se discute en el Capítulo 17.

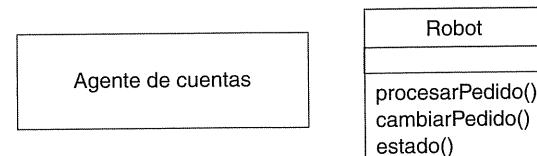


Figura 4.11: Modelado de cosas que no son software.

Modelado de tipos primitivos

Los tipos se discuten en el Capítulo 11.

En el extremo opuesto, las cosas que se modelan pueden extraerse directamente del lenguaje de programación que se utilice para implementar la solución. Normalmente, estas abstracciones involucrarán tipos primitivos, como enteros, caracteres, cadenas e incluso tipos enumerados, que uno mismo podría crear.

Para modelar tipos primitivos:

- Hay que modelar la cosa que se está abstrayendo como un tipo o una enumeración, que se representa como una clase con el estereotipo adecuado.
- Si se necesita especificar el rango de valores asociado al tipo, hay que utilizar restricciones.

Las restricciones se describen en el Capítulo 6.

Los tipos se discuten en el Capítulo 11.

Como se muestra en la Figura 4.12, estas cosas pueden modelarse en UML como tipos o enumeraciones, que se representan del mismo modo que las clases, pero se marcan explícitamente mediante estereotipos. Los tipos primitivos como los enteros (representados por la clase Int) se modelan como tipos, y se puede indicar explícitamente el rango de valores que pueden tomar con una restricción; la semántica de los tipos primitivos debe definirse externamente a UML. Los tipos enumerados, tales como Boolean y Estado, pueden modelarse como enumeraciones, con sus valores literales individuales listados en el compartimento de los atributos (aunque no son atributos). Los tipos enumerados también pueden definir operaciones.

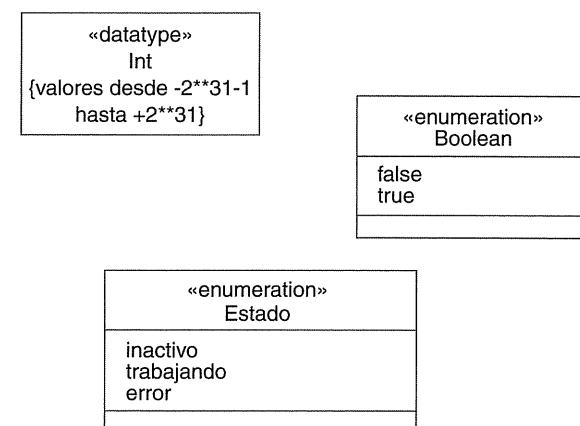


Figura 4.12: Modelado de tipos primitivos.

Nota: Algunos lenguajes, como C y C++, permiten asociar a un enumerado un valor entero equivalente. Esto se puede modelar en UML, asignando una nota al literal de la enumeración como guía para la implementación. Los valores enteros no son necesarios para el modelado lógico.

Sugerencias y consejos

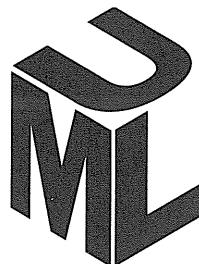
Al modelar clases en UML, hay que recordar que cada clase debe corresponderse con una abstracción tangible o conceptual en el dominio del usuario final o del implementador. Una clase bien estructurada:

- Proporciona una abstracción precisa de algo extraído del vocabulario del dominio del problema o del dominio de la solución.
- Contiene un pequeño conjunto bien definido de responsabilidades, y las lleva a cabo todas ellas muy bien.
- Proporciona una clara distinción entre la especificación de la abstracción y su implementación.
- Es comprensible y sencilla, a la vez que extensible y adaptable.

Cuando se dibuje una clase en UML:

- Hay que mostrar sólo aquellas propiedades de la clase que sean importantes para comprender la abstracción en su contexto.
- Hay que organizar las listas largas de atributos y operaciones, agrupándolos de acuerdo con su categoría.
- Hay que mostrar las clases relacionadas en el mismo diagrama de clases.

LENGUAJE
UNIFICADO DE
MODELADO



Capítulo 5 RELACIONES

En este capítulo

- Relaciones de dependencia, generalización y asociación.
- Modelado de dependencias simples.
- Modelado de la herencia simple.
- Modelado de relaciones estructurales.
- Creación de redes de relaciones.

Al realizar abstracciones, uno se da cuenta de que muy pocas clases se encuentran aisladas. En vez de ello, la mayoría colaboran con otras de varias maneras. Por tanto, al modelar un sistema, no sólo hay que identificar los elementos que conforman el vocabulario del sistema, también hay que modelar cómo se relacionan estos elementos entre sí.

Las características avanzadas de las relaciones se discuten en el Capítulo 10.

En el modelado orientado a objetos hay tres tipos de relaciones especialmente importantes: *dependencias*, que representan relaciones de uso entre clases (incluyendo refinamiento, traza y ligadura); *generalizaciones*, que conectan clases generales con sus especializaciones, y *asociaciones*, que representan relaciones estructurales entre objetos. Cada una de estas relaciones proporciona una forma diferente de combinar las abstracciones.

La construcción de redes de relaciones no es muy diferente de establecer una distribución equilibrada de responsabilidades entre las clases. Si se modela en exceso, se acabará con un lío de relaciones que harán el modelo incomprendible; si se modela insuficientemente, se habrá perdido un montón de la riqueza del sistema, contenida en la forma en que las cosas colaboran entre sí.

Introducción

Al construir una casa, cosas como paredes, puertas, ventanas, armarios y luces formarán parte de nuestro vocabulario. Sin embargo, ninguna de esas cosas se encuentra aislada. Las paredes limitan con otras paredes. Las puertas y las ventanas se colocan en las paredes para hacer aberturas para la gente y la luz. Los muebles y las luces están físicamente adyacentes a las paredes y techos. Las paredes, las puertas, los armarios y las luces se pueden agrupar para formar cosas más complejas, como las habitaciones.

Además de las relaciones estructurales entre estas cosas, también aparecerán otros tipos de relaciones. Por ejemplo, seguro que la casa tiene ventanas, pero probablemente sean de varios tipos diferentes. Puede haber grandes miradores que no se abran, así como pequeñas ventanas en la cocina que sí lo hagan. Algunas ventanas se abrirán hacia arriba y abajo; otras, como las ventanas que dan al patio, se abrirán de izquierda a derecha. Algunas ventanas tendrán una única capa de cristal; otras la tendrán doble. Sin importar sus diferencias, hay algunas cualidades “ventanales” en cada una de ellas: todas son una abertura en una pared, y todas están diseñadas para dejar pasar la luz, el aire y, a veces, gente.

En UML, las formas en que las cosas se conectan entre sí, bien lógica o físicamente, se modelan como relaciones. En el modelado orientado a objetos hay tres clases de relaciones muy importantes: dependencias, generalizaciones y asociaciones.

Las *dependencias* son relaciones de uso. Por ejemplo, las tuberías dependen del calentador para calentar el agua que conducen.

Las *asociaciones* son relaciones estructurales entre instancias. Por ejemplo, las habitaciones constan de paredes y otras cosas; las paredes a su vez pueden tener puertas y ventanas; las tuberías pueden atravesar las paredes.

Las *generalizaciones* conectan clases generales con otras más especializadas en lo que se conoce como relaciones subclase/superclase o hijo/padre. Por ejemplo, un mirador es un tipo de ventana con grandes hojas de cristal fijas; una ventana de patio es un tipo de ventana con hojas de cristal que se abren de lado a lado.

Estos tres tipos de relaciones cubren la mayoría de las formas importantes en que colaboran unas cosas con otras. Como era de esperar, también se corresponden bien con las formas que ofrecen la mayoría de los lenguajes de programación orientados a objetos para conectar objetos.

Otros tipos de relaciones, como la realización y el refinamiento, se discuten en el Capítulo 10.

UML proporciona una representación gráfica para cada uno de estos tipos de relaciones, como se muestra en la Figura 5.1. Esta notación permite visualizar relaciones independientemente de cualquier lenguaje de programación específico, y de forma que permite destacar las partes más importantes de una relación: su nombre, los elementos que conecta y sus propiedades.

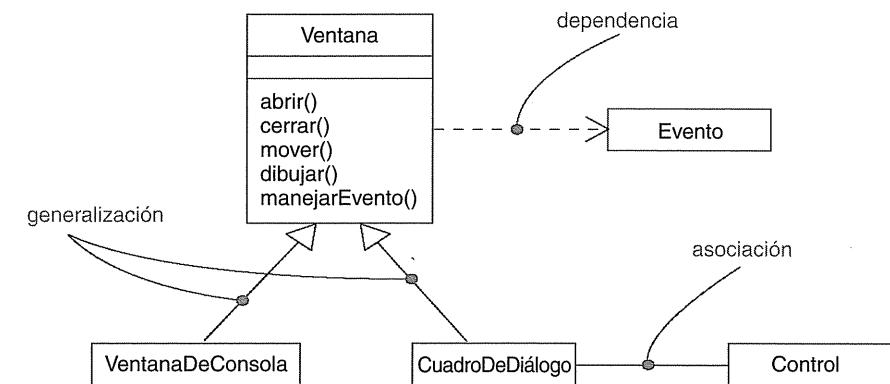


Figura 5.1: Relaciones.

Términos y conceptos

Una *relación* es una conexión entre elementos. En el modelado orientado a objetos, las tres relaciones más importantes son las dependencias, las generalizaciones y las asociaciones. Gráficamente, una relación se representa como una línea, usándose diferentes tipos de línea para diferenciar los tipos de relaciones.

Dependencia

Una *dependencia* es una relación de uso que declara que un elemento (por ejemplo, la clase Ventana) utiliza la información y los servicios de otro elemento (por ejemplo, la clase Evento), pero no necesariamente a la inversa. Gráficamente, una dependencia se representa como una línea discontinua dirigida hacia el elemento del cual se depende. Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro.

La mayoría de las veces, las dependencias se utilizarán en el contexto de las clases, para indicar que una clase utiliza las operaciones de otra o que utiliza variables o parámetros cuyo tipo viene dado por la otra clase; véase la Figura 5.2. Esto es claramente una relación de uso (si la clase utilizada cambia, la operación de la otra clase puede verse también afectada, porque la clase utilizada puede

Las notas se discuten en el Capítulo 6; los paquetes se discuten en el Capítulo 12.

presentar ahora una interfaz o un comportamiento diferentes). En UML también se pueden crear dependencias entre otros muchos elementos, especialmente nodos y paquetes.

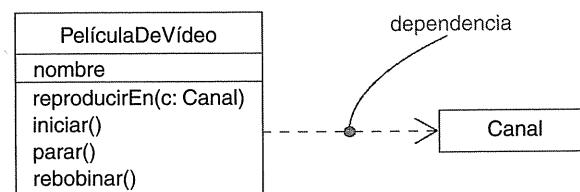


Figura 5.2: Dependencias.

Los diferentes tipos de dependencias se discuten en el Capítulo 10; los estereotipos se discuten en el Capítulo 6.

Nota: Una dependencia puede tener un nombre, aunque es raro que se necesiten los nombres, a menos que se tenga un modelo con muchas dependencias y haya que referirse a ellas o distinguirlas. Normalmente se utilizarán estereotipos para distinguir diferentes variedades de dependencias.

Generalización

Una *generalización* es una relación entre un elemento general (llamado superclase o padre) y un caso más específico de ese elemento (llamado subclase o hijo). La generalización se llama a veces relación “*es-un-tipo-de*”: un elemento (como la clase *Mirador*) *es-un-tipo-de* un elemento más general (por ejemplo, la clase *Ventana*). Un objeto de la clase hija se puede asociar a una variable o un parámetro cuyo tipo venga dado por el padre, pero no a la inversa. En otras palabras, la generalización significa que el hijo puede sustituir a una declaración del padre. Un hijo hereda las propiedades de sus padres, especialmente sus atributos y operaciones. A menudo (pero no siempre) el hijo añade atributos y operaciones a los que hereda de sus padres. Una implementación de una operación en un hijo redefine la implementación de la misma operación en el padre; esto se conoce como *polimorfismo*. Para ser la misma, ambas operaciones han de tener la misma firma (mismo nombre y parámetros). Gráficamente, la generalización se representa como una línea dirigida continua, con una gran punta de flecha vacía, apuntando al padre, como se muestra en la Figura 5.3. Las generalizaciones se utilizarán cuando se quiera mostrar relaciones padre/hijo.

Una clase puede tener ninguno, uno o más padres. Una clase sin padres y uno o más hijos se denomina clase raíz o clase base. Una clase sin hijos se llama

Los paquetes se discuten en el Capítulo 12.

clase hoja. Una clase con un único parente se dice que utiliza herencia simple; una clase con más de un parente se dice que utiliza herencia múltiple.

A menudo se utilizan generalizaciones entre clases e interfaces para representar relaciones de herencia. En UML también se pueden establecer generalizaciones entre otros tipos de clasificadores, como por ejemplo nodos.

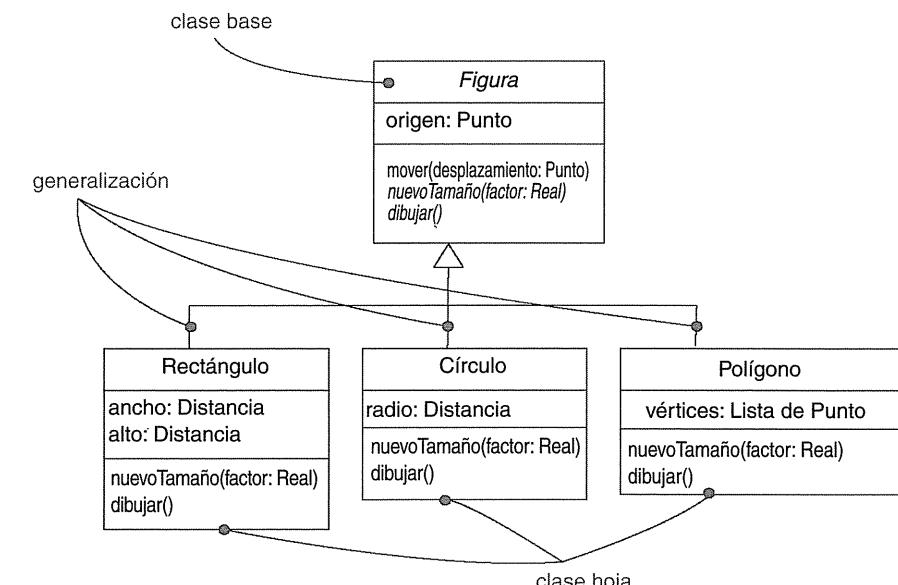


Figura 5.3: Generalización.

Nota: Una generalización con un nombre indica una descomposición de las subclases de una superclase a partir de un aspecto determinado, lo que se denomina un conjunto de generalización. Los conjuntos de generalización múltiples son ortogonales; la idea es que la superclase se especialice mediante herencia múltiple para seleccionar una subclase de cada conjunto. Éste es un tema avanzado que no se trata en este libro.

Asociación

Las asociaciones y las dependencias (pero no las generalizaciones) pueden ser reflexivas, como se discute en el Capítulo 10.

Una *asociación* es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro. Dada una asociación entre dos clases, se puede establecer una relación desde un objeto de una clase hasta algunos objetos de la otra clase. También es válido que ambos extremos de una

asociación estén conectados a la misma clase. Esto significa que un objeto de la clase se puede conectar con otros objetos de la misma clase. Una asociación que conecta exactamente dos clases se dice binaria. Aunque no es frecuente, se pueden tener asociaciones que conecten más de dos clases; éstas se llaman *asociaciones n-arias*. Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases. Las asociaciones se utilizarán cuando se quiera representar relaciones estructurales.

A parte de esta forma básica, hay cuatro adornos que se aplican a las asociaciones.

No debe confundirse la dirección del nombre con la navegación de la asociación, como se discute en el Capítulo 10.

Nombre. Una asociación puede tener un nombre, que se utiliza para describir la naturaleza de la relación. Para que no haya ambigüedad en su significado, se puede dar una dirección al nombre por medio de una flecha que apunte en la dirección en la que se pretende que se lea el nombre, como se muestra en la Figura 5.4.

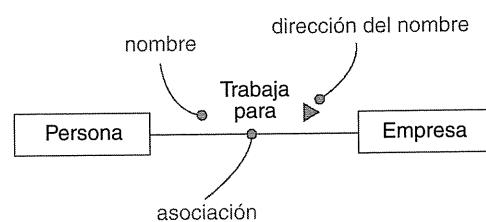


Figura 5.4: Nombres de asociaciones.

Nota: Aunque una asociación puede tener un nombre, normalmente no se necesita incluirlo si se proporcionan explícitamente nombres de rol para la asociación. Si existe más de una asociación entre las mismas clases, es necesario usar nombres de asociación o bien nombres en los extremos de la asociación para distinguirlas. Si una asociación tiene más de un extremo sobre la misma clase, es necesario usar nombres en el extremo de la asociación para distinguirlas. Si sólo hay una asociación entre un par de clases, algunos modeladores omiten los nombres, pero es mejor proporcionarlos para dejar claro el propósito de la asociación.

Los roles se relacionan con la semántica de las interfaces, como se discute en el Capítulo 11.

Rol. Cuando una clase participa en una asociación, tiene un rol específico que juega en esa relación; un rol es simplemente la cara que la clase de un extremo de la asociación presenta a la clase del otro extremo. Se puede dar un nombre explícito al rol que juega una clase en una asociación. El rol que juega una cla-

se en una asociación se denomina *nombre de extremo* (en UML 1 se llamaba *nombre de rol*). En la Figura 5.5, una Persona que juega el rol de empleado está asociada con una Empresa que juega el rol de patrón.

Nota: La misma clase puede jugar el mismo o diferentes roles en otras asociaciones.

Nota: Un atributo puede verse como una asociación de un solo sentido en una clase. El nombre del atributo se corresponde con el nombre de extremo del extremo de la asociación más alejado de la clase.

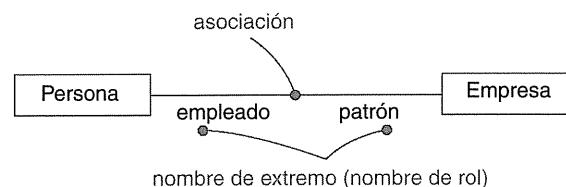


Figura 5.5: Nombres de extremos de asociación (nombres de roles).

Una instancia de una asociación se denomina enlace, como se discute en el Capítulo 16.

Multiplicidad. Una asociación representa una relación estructural entre objetos. En muchas situaciones de modelado, es importante señalar cuántos objetos pueden conectarse a través de una instancia de una asociación. Este “cuántos” se denomina multiplicidad del rol de la asociación, y representa un rango de enteros que especifican el tamaño posible del conjunto de objetos relacionados. La multiplicidad se escribe como una expresión con un valor mínimo y un valor máximo, que pueden ser iguales; se utilizan dos puntos consecutivos para separar ambos valores. Cuando se indica una multiplicidad en un extremo de una asociación, se está especificando cuántos objetos de la clase de ese extremo puede haber para cada objeto de la clase en el otro extremo. Se puede indicar una multiplicidad de exactamente uno (1), cero o uno (0..1), muchos (0..*), o uno o más (1..*). Se puede dar un rango de enteros (como 2..5). Incluso se puede indicar un número exacto (por ejemplo, 3, lo que equivale a 3..3).

Por ejemplo, en la Figura 5.6, cada objeto empresa tiene como empleados a uno o más objetos persona (multiplicidad 1..*); cada objeto persona tiene como patrón a cero o más objetos empresa (multiplicidad *, lo que equivale a 0..*).

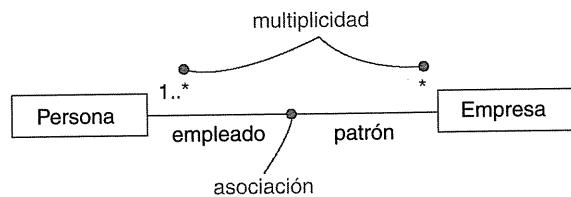


Figura 5.6: Multiplicidad.

La agregación tiene muchas variaciones importantes, como se discute en el Capítulo 10.

Agregación. Una asociación normal entre dos clases representa una relación estructural entre iguales, es decir, ambas clases están conceptualmente en el mismo nivel, sin ser ninguna más importante que la otra. A veces, se desea modelar una relación “todo/parte”, en la cual una clase representa una cosa grande (el “todo”), que consta de elementos más pequeños (las “partes”). Este tipo de relación se denomina agregación, la cual representa una relación del tipo “tiene-un”, o sea, un objeto del todo tiene objetos de la parte. En realidad, la agregación es sólo un tipo especial de asociación y se especifica añadiendo a una asociación normal un rombo vacío en la parte del todo, como se muestra en la Figura 5.7.

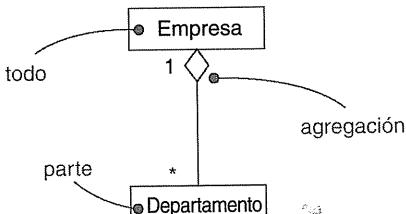


Figura 5.7: Agregación.

Nota: El significado de esta forma simple de agregación es completamente conceptual. El rombo vacío distingue el “todo” de la “parte”, ni más ni menos. Esto significa que la agregación simple no cambia el significado de la navegación a través de la asociación entre el todo y sus partes, ni liga la existencia del todo y sus partes. Para una forma más restrictiva de la agregación, véase la sección sobre la composición en el Capítulo 10.

Los conceptos avanzados de las relaciones se discuten en el Capítulo 10.

Otras características

Las dependencias simples, sin adornos, las generalizaciones y las asociaciones con nombres, multiplicidades y roles son las características más frecuentes que se necesitan al crear abstracciones. De hecho, para la mayoría de los modelos

que uno construya, la forma básica de estas tres relaciones será todo lo necesario para mostrar la semántica principal de las relaciones. Sin embargo, a veces es necesario visualizar o especificar otras características, como la agregación compuesta, la navegación, discriminantes, clases asociación, o tipos especiales de dependencias y generalizaciones. Estas y otras características se pueden expresar en UML, pero se tratan como conceptos avanzados.

Los diagramas de clases se discuten en el Capítulo 8.

Los enlaces se discuten en el Capítulo 16.

Las dependencias, la generalización y las asociaciones son elementos estáticos que se definen al nivel de las clases. En UML estas relaciones se muestran normalmente en los diagramas de clases.

Cuando se empieza a modelar a nivel de objetos, y especialmente cuando se trabaja con colaboraciones dinámicas de estos objetos, aparecerán enlaces, que son instancias de asociaciones que representan conexiones entre objetos a través de las que se pueden enviar mensajes.

Estilos de dibujo

Las relaciones se muestran en los diagramas como líneas que van de un icono a otro. Las líneas tienen varios adornos, como las flechas y los rombos, para distinguir distintos tipos de relaciones. Normalmente, los modeladores eligen uno de entre estos dos estilos para dibujar líneas:

- Líneas oblicuas con cualquier ángulo. Se utiliza un único segmento a menos que sean necesarios varios segmentos para evitar a otros iconos.
- Líneas rectilíneas paralelas a los lados del papel. A menos que una línea conecte a dos iconos alineados, la línea debe dibujarse como una serie de segmentos de línea conectados con ángulos rectos. Éste es el estilo que se utiliza principalmente en este libro.

Con cuidado, la mayoría de los cruces de líneas pueden evitarse. Si es necesario un cruce y hay ambigüedad acerca de cómo están conectadas las líneas, se puede usar un pequeño arco para indicar el cruce.

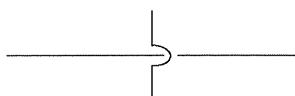


Figura 5.8: Símbolo de cruce de líneas.

Técnicas comunes de modelado

Modelado de dependencias simples

Un tipo común de relación de dependencia es la conexión entre una clase que utiliza a otra clase como parámetro de una operación.

Para modelar esta relación de uso:

- Hay que crear una dependencia que vaya desde la clase con la operación hasta la clase utilizada como parámetro de la operación.

Por ejemplo, la Figura 5.9 muestra un conjunto de clases extraídos de un sistema que gestiona la asignación de estudiantes y profesores a cursos en una universidad. Esta figura muestra una dependencia desde PlanDelCurso hacia Curso, porque Curso se utiliza en las dos operaciones añadir y eliminar de PlanDelCurso.

Si se proporciona la firma completa de la operación, como en esta figura, normalmente no se necesita mostrar la dependencia, porque el uso de la clase ya está explícito en la firma. Sin embargo, a veces se deseará mostrar esta dependencia, especialmente si se han omitido las firmas de las operaciones o si el modelo exhibe otras relaciones con la clase utilizada.

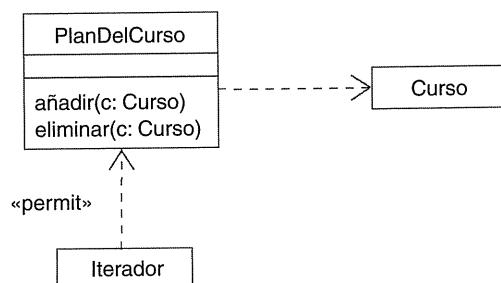


Figura 5.9: Relaciones de dependencia.

Otros estereotipos para las relaciones se discuten en el Capítulo 10.

Esta figura presenta otro tipo de dependencia, la cual no involucra a clases en operaciones, sino que modela una relación entre clases particular de C++. La dependencia que parte de `Iterador` revela que `Iterador` utiliza a `PlanDelCurso`; `PlanDelCurso` no sabe nada sobre `Iterador`. La dependencia está etiquetada con el estereotipo `<<permit>>`, lo que es similar a la instrucción `friend` de C++.

Modelado de la herencia simple

Al modelar el vocabulario de un sistema, se tropieza con clases que son similares a otras en la estructura o en el comportamiento. Cada una de ellas se podría modelar como una abstracción diferenciada e independiente. Sin embargo, una forma mejor consistiría en extraer las características comunes de estructura o comportamiento y colocarlas en clases más generales de las que hereden las clases especializadas.

Para modelar relaciones de herencia:

- Dado un conjunto de clases, hay que buscar responsabilidades, atributos y operaciones comunes a dos o más clases.
- Hay que elevar estas responsabilidades, atributos y operaciones comunes a una clase más general. Si es necesario, hay que crear una nueva clase a la cual se puedan asignar estos elementos (pero con cuidado de no introducir demasiados niveles).
- Hay que especificar que las clases más específicas heredan de la clase más general a través de una relación de generalización desde cada clase especializada a su padre.

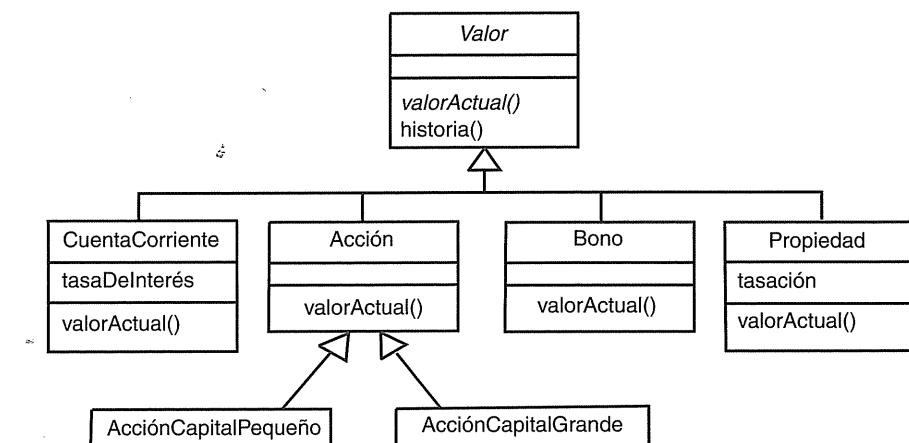


Figura 5.10: Relaciones de herencia.

Por ejemplo, la Figura 5.10 muestra un conjunto de clases extraídas de una aplicación financiera. Podemos encontrar una relación de generalización desde cuatro clases (`CuentaCorriente`, `Acción`, `Bono` y `Propiedad`) hacia la clase más general, `Valor`. `Valor` es el parente, y `CuentaCorriente`,

Acción, Bono y Propiedad son todas hijas. Cada una de estas hijas especializadas es un tipo de Valor. Se puede ver que Valor incluye dos operaciones: valorActual e historia. Como Valor es su padre, CuentaCorriente, Acción, Bono y Propiedad heredan estas dos operaciones, y por la misma razón, cualesquiera otros atributos y operaciones de Valor que puedan estar omitidos en la figura.

Las clases y las operaciones abstractas se discuten en el Capítulo 9.

Podemos ver que los nombres *Valor* y *valorActual* están escritos de forma ligeramente distinta a los otros. Hay un motivo para esto. Cuando se construyen jerarquías como la de la figura, a menudo encontraremos clases no hoja que son incompletas, o simplemente son clases de las que no se desea que haya objetos. Estas clases se llaman *abstractas*. En UML se puede especificar que una clase es abstracta escribiendo su nombre en cursiva, como en la clase *Valor*. Esta convención se aplica a las operaciones como *valorActual* y significa que dicha operación proporciona una firma, pero por lo demás está incompleta y, por tanto, debe ser implementada por algún método a un nivel más bajo de abstracción. De hecho, como se ve en la figura, las cuatro clases hijas inmediatas de *Valor* son concretas (es decir, no son abstractas) y cada una debe proporcionar una implementación concreta de la operación *valorActual*.

Las jerarquías de generalización/especialización no tienen por qué limitarse a dos niveles. De hecho, como se ve en la Figura 5.10, es frecuente tener más de dos niveles de herencia. AcciónCapitalPequeño y AcciónCapitalGrande son ambas hijas de Acción, la cual, a su vez, es hija de Valor. Valor es, por tanto, una clase raíz, ya que no tiene padres. AcciónCapitalPequeño y AcciónCapitalGrande son ambas clases hoja porque no tienen hijas. Acción tiene un parente, así como hijas, y por tanto no es una clase raíz ni una clase hoja.

La herencia múltiple se discute en el Capítulo 10.

Aunque aquí no se muestra, también se pueden crear clases que tengan más de un parente. Esto se denomina herencia múltiple y significa que la clase dada tiene todos los atributos, operaciones y asociaciones de todos sus padres.

Por supuesto, no puede haber ciclos en una jerarquía de herencia; una determinada clase no puede ser su propio antecesor.

Modelado de relaciones estructurales

Al modelar con relaciones de dependencia o generalización, se puede estar modelando clases que representan diferentes niveles de importancia o dife-

rentes niveles de abstracción. Dada una dependencia entre dos clases, una clase depende de la otra, pero la segunda no tiene conocimiento de la primera. En una relación de generalización entre dos clases, la clase hija hereda de su parente, pero el parente no tiene conocimiento específico de sus clases hijas. En definitiva, las relaciones de dependencia y generalización son asimétricas.

Las asociaciones son, por defecto, bidireccionales; se puede limitar su dirección, como se discute en el Capítulo 10.

Cuando se modela con relaciones de asociación, se están modelando clases del mismo nivel. Dada una asociación entre dos clases, ambas dependen de la otra de alguna forma, y se puede navegar en ambas direcciones. Mientras que una dependencia es una relación de uso y la generalización es una relación es-un-tipo-de, una asociación especifica un camino estructural a través del cual interactúan los objetos de las clases implicadas.

Para modelar relaciones estructurales:

- Para cada par de clases, si es necesario navegar desde los objetos de una hacia los objetos de la otra, hay que especificar una asociación entre las dos. Ésta es una vista de las asociaciones dirigida por los datos.
- Para cada par de clases, si los objetos de una clase necesitan interactuar de alguna forma con los objetos de la otra, aparte de como variables locales en un procedimiento o como parámetros de una operación, entonces hay que especificar una asociación entre ambas. Ésta es una vista de las asociaciones dirigida por el comportamiento.
- Para cada una de estas asociaciones, hay que especificar la multiplicidad (especialmente cuando no sea *, que es el valor por defecto), así como los nombres de rol (especialmente si ello ayuda a explicar el modelo).
- Si una de las clases de la asociación es, desde un punto de vista estructural o de organización, un todo comparada con las clases en el otro extremo, que parecen las partes, hay que marcarla como una agregación, añadiendo el rombo a la asociación en el extremo del todo.

Los casos de uso se discuten en el Capítulo 17.

¿Cómo se sabe cuándo deben interactuar los objetos de una clase dada con los objetos de otra clase? La respuesta es que las tarjetas CRC y el análisis basado en casos de uso ayudan enormemente, ya que precisan considerar escenarios estructurales y de comportamiento. Donde se descubra que dos o más clases interactúan mediante relaciones de datos, se debe especificar una asociación.

La Figura 5.11 muestra un conjunto de clases extraído de un sistema de información de una universidad. En la parte inferior del diagrama se encuentran las clases Estudiante, Curso y Profesor. Hay una asociación entre Estudiante y Curso, detallando que los estudiantes asisten a los cursos. Además, cada estudiante puede asistir a cualquier número de cursos y cada curso puede tener cualquier número de estudiantes. Análogamente, existe una asociación entre Curso y Profesor, que especifica que los profesores imparten cursos. Para cada curso hay al menos un profesor y cada profesor puede impartir cero o más cursos. Cada curso pertenece exactamente a un departamento.

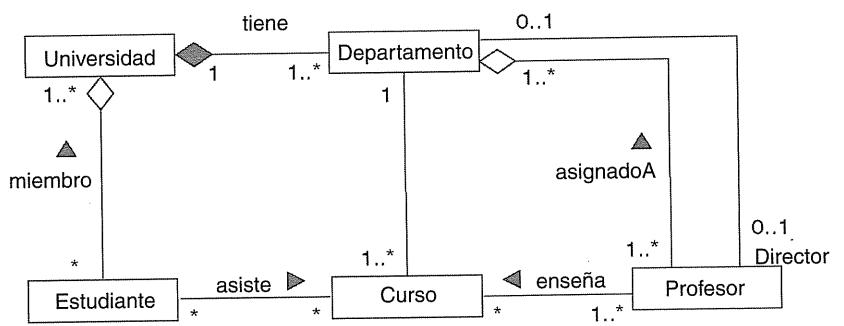


Figura 5.11: Relaciones estructurales.

La relación de agregación entre Universidad y Departamento es una composición, como se discute en el Capítulo 10. La composición es una forma más estricta de agregación, que implica la posesión.

Las relaciones entre Universidad y las clases Estudiante y Departamento son algo diferentes. Ahí se ven relaciones de agregación. Una universidad tiene cero o más estudiantes, cada estudiante puede ser miembro de una o más universidades, una universidad tiene uno o más departamentos, y cada departamento pertenece exactamente a una universidad. Se podrían omitir los adornos de la agregación y utilizar simples asociaciones, pero al especificar que Universidad es un todo y que Estudiante y Departamento son algunas de sus partes, se clarifica cuál es superior a las otras, desde el punto de vista organizativo. Así, las universidades están de algún modo definidas por los estudiantes y los departamentos que tienen. De forma parecida, estudiantes y departamentos realmente no existen solos, fuera de la universidad a la que pertenecen. Más bien, obtienen parte de su identidad a partir de su universidad.

También se ve que hay dos asociaciones entre Departamento y Profesor. Una de esas asociaciones especifica que cada profesor está asignado a uno o más departamentos y que cada departamento tiene uno o más profesores. Esto se modela como una agregación porque, organizativamente, los

departamentos están a un nivel superior que los profesores en la estructura de la universidad. La otra asociación especifica que para cada departamento hay exactamente un profesor que es el director del departamento. Según está especificado este modelo, un profesor puede ser director de un departamento a lo sumo y algunos profesores no son directores de ningún departamento.

Nota: Es posible que este modelo parezca inapropiado porque no refleje la realidad del lector. Puede que otra universidad no tenga departamentos. Puede que haya directores que no sean profesores o incluso se puedan tener estudiantes que sean a su vez profesores. Esto no significa que el modelo aquí presentado sea erróneo, sólo es diferente. No se puede modelar sin estar conectado a una realidad, y cada modelo como éste depende de cómo se pretenda utilizar.

Sugerencias y consejos

Cuando se modelen relaciones en UML:

- Hay que usar dependencias sólo cuando la relación que se esté modelando no sea estructural.
- Hay que usar la generalización sólo cuando se tenga una relación “es-un-tipo-de”; la herencia múltiple se puede reemplazar a menudo por la agregación.
- Hay que tener cuidado de no introducir relaciones de generalización cíclicas.
- En general, hay que mantener las relaciones de generalización equilibradas; las jerarquías de herencia no deberían ser demasiado profundas (más de cinco niveles o así deberían cuestionarse) ni demasiado anchas (en lugar de ello, buscar la posibilidad de clases abstractas intermedias).
- Hay que usar asociaciones principalmente donde existan relaciones estructurales entre objetos. No hay que usarlas para mostrar relaciones transitorias como parámetros o variables locales de los procedimientos.

Cuando se dibuje una relación en UML:

- Hay que usar consistentemente líneas rectas u oblicuas. Las líneas rectas proporcionan una representación gráfica que destaca las conexiones entre elementos relacionados cuando todos apuntan a un elemento común. Las líneas oblicuas suelen aprovechar mejor el espacio de dibujo en diagramas complejos. El empleo de ambos tipos de líneas en el mismo diagrama es útil para llamar la atención sobre diferentes grupos de relaciones.
- Hay que evitar los cruces de líneas a menos que sean estrictamente necesarios.
- Hay que mostrar sólo aquellas relaciones necesarias para comprender una agrupación particular de elementos. Las relaciones superfluas (especialmente las asociaciones redundantes) deberían omitirse.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 6

MECANISMOS COMUNES

En este capítulo

- Notas.
- Estereotipos, valores etiquetados y restricciones.
- Modelado de comentarios.
- Modelado de nuevos bloques de construcción.
- Modelado de nuevas propiedades.
- Modelado de nueva semántica.
- Formas de extender UML.

Estos mecanismos comunes se discuten en el Capítulo 2.

UML se hace más sencillo por la presencia de cuatro mecanismos comunes que se aplican a lo largo y ancho del lenguaje: especificaciones, adornos, divisiones comunes y mecanismos de extensibilidad. Este capítulo explica el uso de dos de estos mecanismos comunes: los adornos y los mecanismos de extensibilidad.

Las notas son el tipo de adorno más importante que puede aparecer aislado. Una nota es un símbolo gráfico para mostrar restricciones o comentarios asociados a un elemento o a una colección de elementos. Las notas se utilizan para añadir a un modelo información como los requisitos, observaciones, revisiones y explicaciones.

Los mecanismos de extensibilidad de UML permiten extender el lenguaje de forma controlada. Estos mecanismos incluyen los estereotipos, los valores etiquetados y las restricciones. Un estereotipo extiende el vocabulario de UML, permitiendo crear nuevos tipos de bloques de construcción derivados de los existentes, pero específicos del problema objeto del modelado. Un valor etiquetado extiende las propiedades de un estereotipo de UML, permitiendo añadir nueva información en la especificación de ese elemento. Una restricción extiende la semántica de un bloque de construcción de UML, permitiendo añadir nuevas reglas o modificar las existentes. Estos mecanismos se utilizan para adaptar UML a las necesidades específicas de un dominio y una cultura de desarrollo.

Introducción

A veces es necesario salir fuera de los límites de una notación. Por ejemplo, en una obra, un arquitecto puede garabatear unas notas sobre los planos del edificio para comunicar algún detalle sutil a los obreros de la construcción. En un estudio de grabación, un compositor podría inventar una nueva notación musical para representar algún efecto inusual que requiere de un guitarrista. En ambos casos, ya existen lenguajes bien definidos (el lenguaje de los planos de arquitectura y el lenguaje de la notación musical), pero a veces, para comunicar una idea, hay que modificar o extender esos lenguajes de forma controlada.

El modelado es pura comunicación. UML ya proporciona las herramientas necesarias para visualizar, especificar, construir y documentar los artefactos de un amplio rango de sistemas con gran cantidad de software. Sin embargo, pueden aparecer circunstancias en las que se desee modificar o extender UML. Esto le pasa al lenguaje humano constantemente (por eso se publican nuevos diccionarios cada año), porque ningún lenguaje estático puede ser nunca suficiente para abarcar todo lo que se deberá comunicar con él a lo largo del tiempo. Cuando se utiliza un lenguaje de modelado como UML, lo cual, como ya hemos dicho, se hace para comunicar, al modelador le interesaría ceñirse al núcleo del lenguaje, sin considerar extensiones, a menos que haya una razón forzosa para desviarse. Cuando uno se encuentre con la necesidad de expresar algo fuera de los límites del lenguaje, debería hacerlo de forma controlada. De otra manera, sería imposible para cualquier otro comprender lo que se ha hecho.

Las notas son el mecanismo que proporciona UML para capturar comentarios arbitrarios y restricciones que ayuden a clarificar los modelos que se han creado. Las notas pueden mostrar artefactos que juegan un importante papel en el ciclo de vida del desarrollo de software, como son los requisitos, o pueden simplemente mostrar observaciones, revisiones o explicaciones en formato libre.

UML proporciona una representación gráfica para los comentarios y las restricciones, llamada nota, como se muestra en la Figura 6.1. Esta notación permite visualizar directamente un comentario. Las notas proporcionan un punto donde enlazar o incluir otros documentos, siempre que se disponga de las herramientas adecuadas.

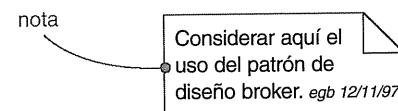


Figura 6.1: Notas.

Los estereotipos, los valores etiquetados y las restricciones son los mecanismos que proporciona UML para añadir nuevos bloques de construcción, crear nuevas propiedades y especificar nueva semántica. Por ejemplo, si se está modelando una red, será deseable tener símbolos para enruteadores y concentradores; se podrían utilizar nodos estereotipados para hacer que estos elementos aparezcan como bloques de construcción primitivos. Del mismo modo, si uno forma parte de un equipo de versiones de un proyecto, y es responsable de ensamblar, probar e instalar las versiones, quizás desee llevar control sobre el número de versión y los resultados de las pruebas para cada subsistema principal. Para ello, se podrían utilizar valores etiquetados para añadir esta información a los modelos. Por último, si se están modelando sistemas de tiempo real muy exigentes, quizás se desee adornar los modelos con información sobre estimaciones y límites de tiempo; se podrían utilizar restricciones para capturar estos requisitos temporales.

UML proporciona una representación textual para los estereotipos, los valores etiquetados y las restricciones, como se muestra en la Figura 6.2. Los estereotipos también permiten introducir nuevos símbolos gráficos, para proporcionar señales visuales en los modelos, relacionadas con el lenguaje del dominio y la cultura de desarrollo.

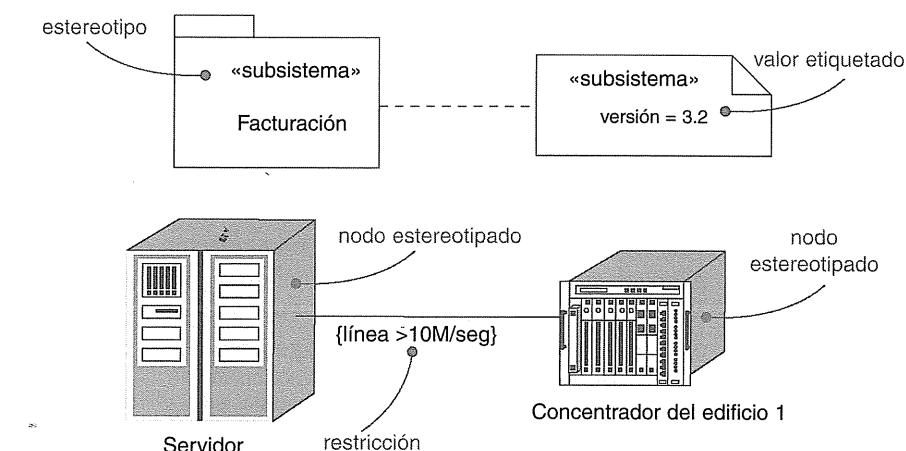


Figura 6.2: Estereotipos, valores etiquetados y restricciones.

Términos y conceptos

Una *nota* es un símbolo gráfico para representar restricciones o comentarios asociados a un elemento o a una colección de elementos. Gráficamente, una nota se representa como un rectángulo con una esquina doblada, junto a un comentario textual o gráfico.

Un *estereotipo* es una extensión del vocabulario de UML que permite crear nuevos tipos de bloques de construcción similares a los existentes, pero específicos del problema que se está modelando. Gráficamente, un estereotipo se representa como un nombre entre comillas francesas (unas marcas como « ») y colocado sobre el nombre de otro elemento. Opcionalmente, el elemento con el estereotipo puede dibujarse con un nuevo ícono asociado a ese estereotipo.

Un *valor etiquetado* es una propiedad de un estereotipo, que permite añadir nueva información en el elemento que lleva ese estereotipo. Gráficamente, un valor etiquetado se representa como una cadena de caracteres de la forma nombre = valor dentro de una nota asociada al objeto.

Una *restricción* es una especificación de la semántica de un elemento de UML, que permite añadir nuevas reglas o modificar las existentes. Gráficamente, una restricción se representa como una cadena de caracteres entre llaves colocada junto al elemento al que está asociada o conectada a ese elemento o elementos por relaciones de dependencia. Como alternativa, una restricción se puede representar en una nota.

Notas

Una nota con un comentario no tiene un efecto semántico, es decir, su contenido no altera el significado del modelo al que está anexa. Por eso las notas se utilizan para especificar cosas como requisitos, observaciones, revisiones y explicaciones, además de representar restricciones.

Las notas pueden asociarse a más de un elemento mediante dependencias, como se discute en el Capítulo 5.

Una nota puede contener cualquier combinación de texto y gráficos. Si la implementación lo permite, se puede incluir en una nota una URL activa, o incluso enlazar o incluir otro documento. De esta forma, UML se puede utilizar para organizar todos los artefactos que se generen o usen durante el desarrollo, como se muestra en la Figura 6.3.

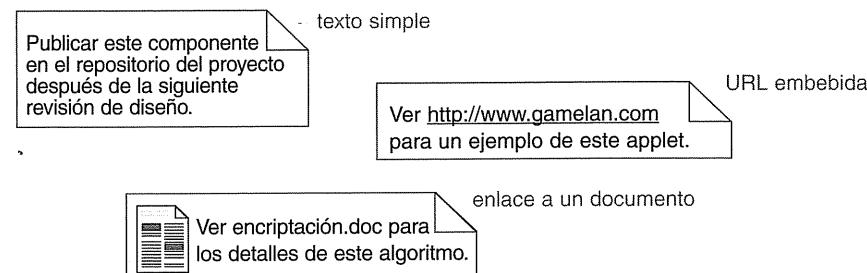


Figura 6.3: Notas.

La notación básica para una asociación, junto con algunos de sus adornos, se discute en los Capítulos 5 y 10.

Otros adornos

Los adornos son complementos gráficos o textuales que se añaden a la notación básica de un elemento para mostrar detalles de su especificación. Por ejemplo, la notación básica para una asociación es una línea, pero ésta se puede adornar con detalles tales como el rol y la multiplicidad de cada extremo. Al utilizar UML, la regla general a seguir es ésta: comenzar con la notación básica de cada elemento y después añadir otros adornos sólo cuando sean necesarios para expresar información específica que sea importante para el modelo.

La mayoría de los adornos se muestran colocando texto junto al elemento de interés o añadiendo un símbolo gráfico a la notación básica. Sin embargo, a veces se quiere adornar un elemento con más detalle del que se puede aportar con simple texto o gráficos. En el caso de elementos como las clases, los componentes y los nodos, se puede añadir un compartimento extra bajo los compartimentos habituales para proporcionar esta información, como se muestra en la Figura 6.4.

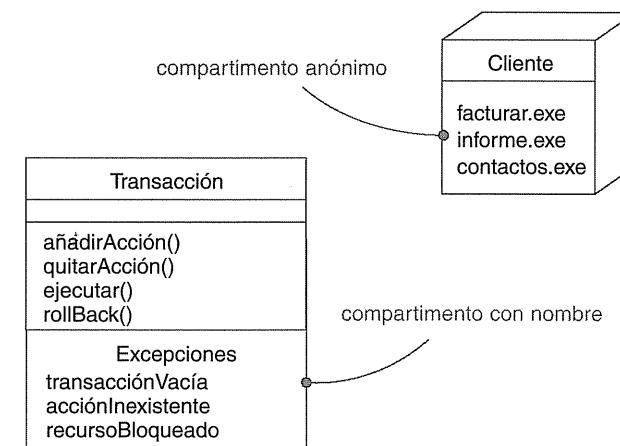


Figura 6.4: Compartimentos extras.

Nota: A menos que sea obvio por su contenido, es buena práctica dar explícitamente un nombre a cualquier compartimento extra, para que no haya confusión sobre su significado. También es buena práctica utilizar de forma moderada los compartimentos extras porque, si se abusa de ellos, los diagramas son más confusos.

Estos cuatro elementos básicos de UML se discuten en el Capítulo 2.

El Proceso Unificado de Rational se resume en el Apéndice B.

Estereotipos

UML proporciona un lenguaje para elementos estructurales, elementos de comportamiento, elementos de agrupación y elementos de anotación. Estos cuatro tipos básicos de elementos son suficientes para la amplia mayoría de los sistemas que será necesario modelar. No obstante, a veces se desea introducir nuevos elementos relacionados con el vocabulario de un dominio y que parezcan bloques de construcción primitivos.

Un estereotipo no es lo mismo que una clase padre en una relación de generalización padre/hijo. Más bien, se puede pensar en un estereotipo como en un metatipo (un tipo que define a otros tipos), porque cada uno crea el equivalente de una nueva clase en el metamodelo de UML. Por ejemplo, si se está modelando un proceso de negocio, quizás se desee introducir elementos como empleados, documentos y políticas. Del mismo modo, si se está siguiendo un proceso de desarrollo, como el Proceso Unificado de Rational, se deseará modelar utilizando clases frontera, control y entidad. Aquí es donde se pone de manifiesto el verdadero valor de los estereotipos. Cuando se aplica un estereotipo sobre un elemento como un nodo o una clase, se está, de hecho, extendiendo UML. Se está creando un nuevo bloque de construcción como cualquiera de los existentes, pero con sus propias características (cada estereotipo puede proporcionar su propio conjunto de valores etiquetados), semántica (cada estereotipo puede proporcionar sus propias restricciones), y notación (cada estereotipo puede proporcionar su propio icono) especiales.

En su forma más sencilla, un estereotipo se representa como un nombre entre comillas francesas (por ejemplo, «nombre»), y se coloca sobre el nombre de otro elemento. Como señal visual, se puede definir un ícono para el estereotipo y mostrar ese ícono a la derecha del nombre (si se utiliza la notación básica para el elemento) o utilizar ese ícono como símbolo básico para el elemento estereotipado. Estos tres enfoques se ilustran en la Figura 6.5.

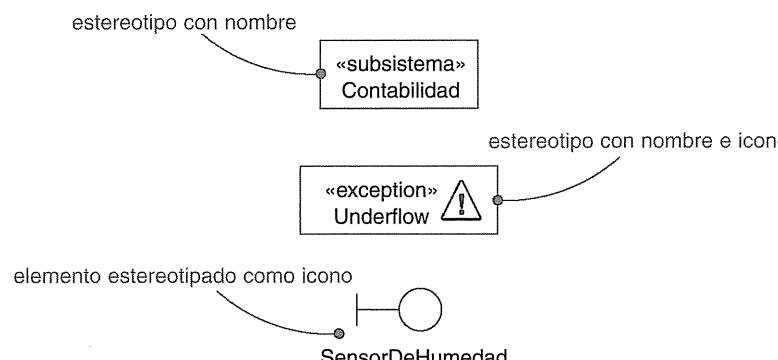


Figura 6.5: Estereotipos.

Nota: Cuando se define un ícono para un estereotipo, debe tenerse en cuenta el empleo del color como apoyo para ofrecer una sutil señal visual (aunque el color debe usarse de forma moderada). UML permite usar cualquier forma en los íconos, y si la implementación lo permite, esos íconos podrían aparecer como herramientas primitivas para que los usuarios que crean diagramas UML tengan una paleta de elementos que son básicos para ellos, relacionados con el vocabulario de su dominio.

Valores etiquetados

Todo elemento de UML tiene su propio conjunto de propiedades: las clases tienen nombres, atributos y operaciones; las asociaciones tienen nombres y dos o más extremos (cada uno con sus propiedades), etc. Con los estereotipos, se pueden añadir nuevos elementos a UML; con los valores etiquetados, se pueden añadir nuevas propiedades a un estereotipo.

Se pueden definir etiquetas que se apliquen a estereotipos individuales, de forma que cualquier elemento con ese estereotipo tenga ese valor etiquetado. Un valor etiquetado no es lo mismo que un atributo de una clase. En vez de ello, se puede ver un valor etiquetado como un metadato porque su valor se aplica a la especificación del elemento, no a sus instancias. Por ejemplo, como se muestra en la Figura 6.6, se podría especificar la capacidad necesaria de un servidor, o requerir que sólo haya un único servidor en un sistema.

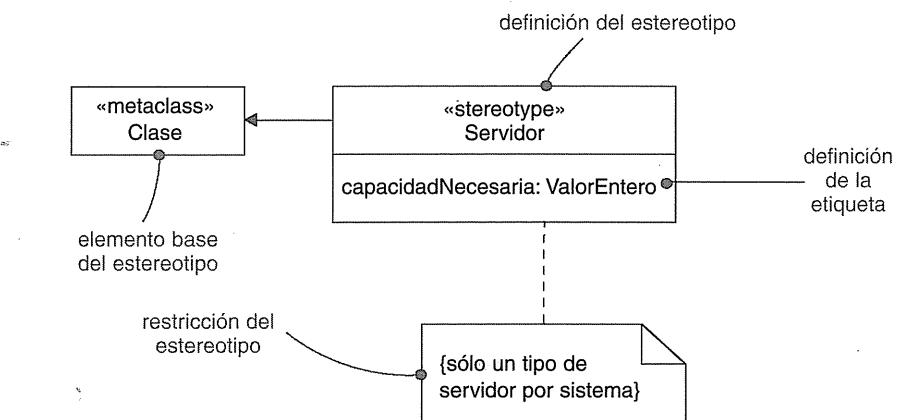


Figura 6.6: Definiciones de estereotipos y etiquetas.

Los valores etiquetados se colocan en notas asociadas al elemento afectado, como se muestra en la Figura 6.7. Cada valor etiquetado incluye una cadena de caracteres con un nombre (la etiqueta), un separador (el símbolo =), y un valor (el de la etiqueta).

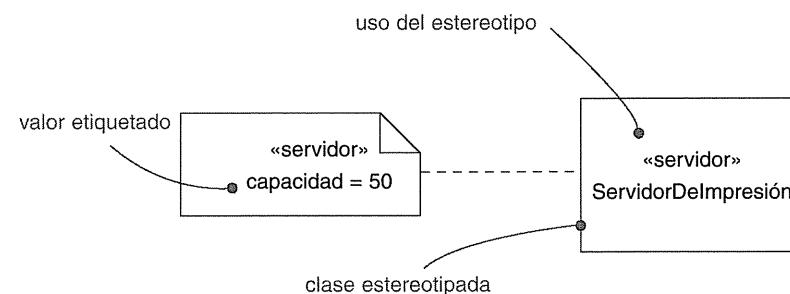


Figura 6.7: Valores etiquetados.

Nota: Uno de los usos más comunes de los valores etiquetados es especificar propiedades relevantes a la generación de código o la gestión de configuraciones. Por ejemplo, se pueden utilizar valores etiquetados para especificar el lenguaje de programación en el cual se implementa una determinada clase. Análogamente, se pueden utilizar valores etiquetados para especificar el autor y la versión de un componente.

Restricciones

Las restricciones de tiempo y espacio, que suelen usarse cuando se modelan sistemas de tiempo real, se discuten en el Capítulo 24.

Las restricciones pueden asociarse a más de un elemento mediante las dependencias, como se discute en el Capítulo 5.

Todo en UML tiene su propia semántica. La generalización (normalmente, si uno sabe lo que le conviene) implica el principio de sustitución de Liskov, y varias asociaciones conectadas a una clase denotan diferentes relaciones. Con las restricciones, se puede añadir nueva semántica o modificar las reglas existentes. Una restricción especifica condiciones que deben cumplirse en cualquier configuración en tiempo de ejecución para que el modelo esté bien formado. Por ejemplo, como se muestra en la Figura 6.8, se podría especificar que la comunicación fuese encriptada a través de una determinada asociación; una configuración que violase esta restricción sería inconsistente con el modelo. Análogamente, se podría especificar que de entre un conjunto de asociaciones conectadas a una clase, una instancia específica sólo pueda tener enlaces correspondientes a una de las asociaciones del conjunto.

Nota: Las restricciones se pueden escribir como texto libre. Si se desea especificar la semántica de forma precisa, se puede utilizar el

Los clasificadores se discuten en el Capítulo 9.

Lenguaje de Restricciones de Objetos (OCL, *Object Constraint Language*) de UML, que se describe con mayor detalle en *The Unified Modeling Language Reference Manual*.

Una restricción se representa como una cadena de caracteres entre llaves junto al elemento asociado. Esta notación también se utiliza como un adorno a la notación básica de un elemento para ver las partes de la especificación de un elemento que no tienen representación gráfica. Por ejemplo, algunas propiedades de las asociaciones (orden y cambios posibles) se muestran con la notación de restricciones.

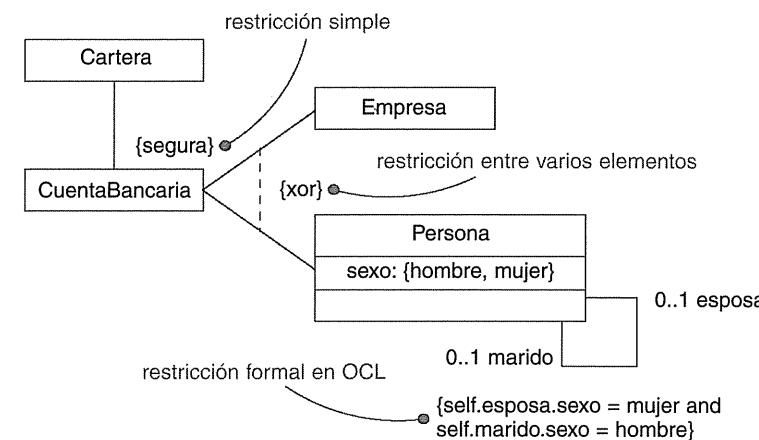


Figura 6.8: Restricciones.

Elementos estándar

UML define varios estereotipos estándar para los clasificadores, componentes, relaciones y otros elementos de modelado. Hay un estereotipo estándar, de interés principalmente para los constructores de herramientas, que permite modelar los propios estereotipos.

- **stereotype** Especifica que el clasificador es un estereotipo que se puede aplicar a otros elementos.

Este estereotipo se utilizará cuando se desee modelar explícitamente los estereotipos definidos para el proyecto.

Perfiles

A menudo es útil definir una versión de UML adaptada a un propósito o a un dominio específico. Por ejemplo, si se desea utilizar UML para generar código

en un lenguaje de programación concreto, es útil definir estereotipos que se puedan aplicar a los elementos para ayudar al generador de código (como por ejemplo la directiva pragma de Ada). Sin embargo, los estereotipos definidos serán distintos para Java que para C++. Otro ejemplo: el uso de UML para modelar bases de datos. Algunas de las capacidades de UML, como el modelado dinámico, son menos importantes, pero harán falta conceptos como claves candidatas e índices. UML se puede adaptar utilizando perfiles.

Un *perfil* (*profile*) es un modelo UML con un conjunto predefinido de estereotipos, valores etiquetados, restricciones y clases básicas. También incluye un subconjunto de tipos de elementos de UML para ser utilizados, de forma que un modelador no se confunda con tipos de elementos innecesarios para un área de aplicación específica. De hecho, un perfil define una versión especializada de UML para un área particular. Como se construye a partir de elementos corrientes de UML, no representa un lenguaje nuevo, y puede ser soportado por herramientas de UML comunes.

La mayoría de los modeladores no construirán sus propios perfiles. La mayoría de los perfiles serán construidos por los creadores de herramientas y *frameworks*, y otros diseñadores de aplicaciones genéricas. Sin embargo, muchos modeladores usarán los perfiles. Ocurre como con las bibliotecas de subrutinas tradicionales: unos pocos expertos las construyen, pero muchos programadores las utilizan. Esperamos que se creen perfiles para lenguajes de programación y para las bases de datos, para diferentes plataformas de implementación, para diversas herramientas de modelado y para varios dominios de aplicación.

Técnicas comunes de modelado

Modelado de comentarios

La mayoría de las veces, las notas se emplearán con el propósito de anotar en formato libre observaciones, revisiones o explicaciones. Incluyendo estos comentarios directamente en los modelos, éstos pueden convertirse en un repositorio común para todos los diferentes artefactos creados durante el desarrollo. Incluso se pueden utilizar las notas para visualizar requisitos e ilustrar explícitamente cómo se ligan a las partes del modelo.

Para modelar un comentario:

- Hay que colocar el comentario como una nota adyacente al elemento al que se refiere. Se puede representar una relación más explícita conectando la nota al elemento con una relación de dependencia.

- Hay que recordar que se pueden ocultar o mostrar los elementos del modelo a voluntad. Esto significa que no se ha de hacer visible un comentario en todos los lugares en los que sean visibles los elementos a los que va asociado. En vez de ello, los comentarios sólo deben incluirse en los diagramas cuando se necesite comunicar esa información en ese contexto.
- Si el comentario es largo o incluye algo más complejo que simple texto, hay que considerar ponerlo en un documento externo y enlazar o incluir ese documento en una nota adjunta al modelo.
- Conforme evoluciona el modelo, hay que mantener aquellos comentarios que recogen decisiones significativas que no se pueden inferir del propio modelo, y se deben descartar los otros a menos que sean de interés histórico.

Por ejemplo, la Figura 6.9 representa un modelo en un punto intermedio del desarrollo de una jerarquía de clases, mostrando algunos requisitos que configuran el modelo, así como algunas notas de una revisión de diseño.

En este ejemplo, la mayoría de los comentarios son texto simple (como la nota para María José), pero uno de ellos (la nota en la parte inferior del diagrama) proporciona un hiperenlace a otro documento.

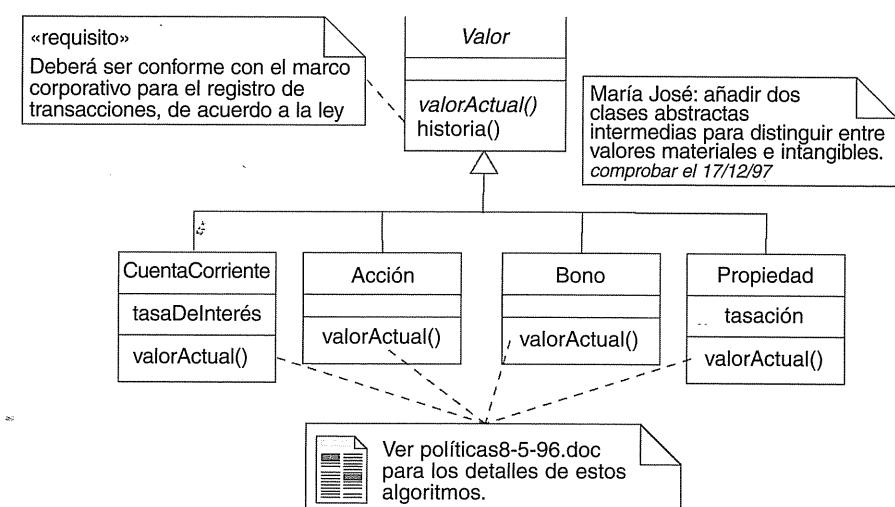


Figura 6.9: Modelado de comentarios.

Modelado de nuevas propiedades

Las propiedades básicas de los bloques de construcción de UML (atributos y operaciones en las clases, el contenido en los paquetes, etcétera) son lo suficiente-

temente genéricos para abarcar la mayoría de los elementos a modelar. Sin embargo, si deseamos extender las propiedades de estos bloques básicos, hay que definir estereotipos o valores etiquetados.

Para modelar nuevas propiedades:

- Hay que asegurarse de que no existe ya en el UML básico una forma de expresar lo que se desea.
- Si se está convencido de que no hay otra forma de expresar la semántica, hay que definir un estereotipo y añadir las nuevas propiedades a éste. Aquí también se aplican las reglas de la generalización: los valores etiquetados definidos para un estereotipo se aplican a sus hijos.

Los subsistemas se discuten en el Capítulo 32.

Por ejemplo, supongamos que se desea asociar los modelos creados al sistema de gestión de configuraciones del proyecto. Entre otras cosas, esto implica llevar el control de los números de versión, del estado actual de las entradas y salidas de modelos, y quizás incluso de las fechas de creación y modificación de cada subsistema. Esta información específica del proceso no es parte básica de UML, aunque se puede añadir como valores etiquetados. Además, esta información no es simplemente un nuevo atributo de la clase. El número de versión de un subsistema es parte de sus metadatos, no parte del modelo.

La Figura 6.10 representa tres subsistemas, cada uno de los cuales ha sido extendido con el estereotipo «versioned» para incluir su número de versión y estado.

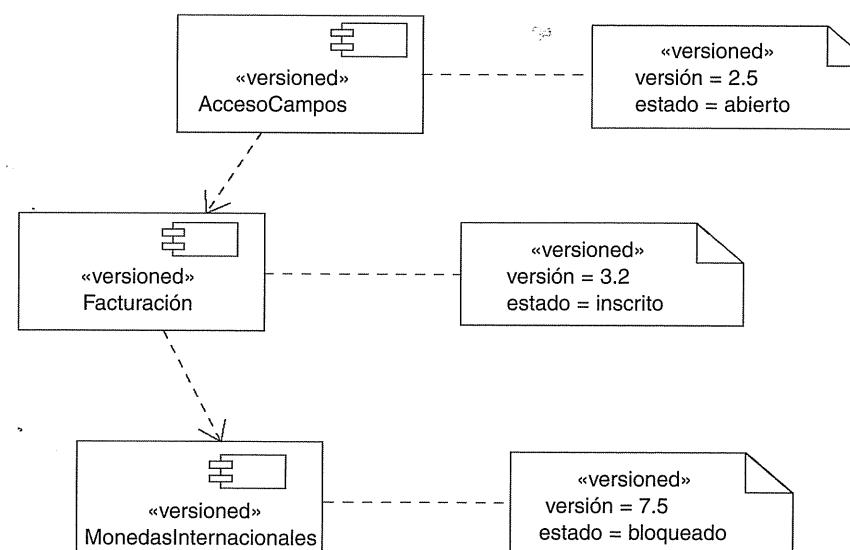


Figura 6.10: Modelado de nuevas propiedades.

Nota: Los valores de etiquetas tales como *versión* y *status* pueden ser establecidos por las herramientas. En vez de introducir a mano estos valores en el modelo, se puede usar un entorno de desarrollo que integre las herramientas de gestión de configuraciones con las herramientas de modelado para obtener estos valores.

Modelado de nueva semántica

Cuando se crea un modelo con UML, se trabaja dentro de las reglas que impone UML. Esto es bueno, porque significa que se puede comunicar una idea sin ambigüedad a cualquiera que sepa cómo leer UML. No obstante, si se necesita expresar nueva semántica sobre la cual UML no dice nada o si se necesitan modificar las reglas de UML, entonces hay que escribir una restricción.

Para modelar nueva semántica:

- Primero, hay que asegurarse de que no existe ya en el UML básico una forma de expresar lo que se quiere.
- Si se está convencido de que no hay otra forma de expresar la semántica, hay que escribir la nueva semántica en una restricción junto al elemento al que se refiere. Se puede mostrar una relación más explícita conectando la restricción al elemento mediante una relación de dependencia.
- Si se necesita especificar la semántica de forma más precisa y formal, hay que escribirla con OCL.

Por ejemplo, la Figura 6.11 modela una pequeña parte de un sistema de recursos humanos de una empresa.

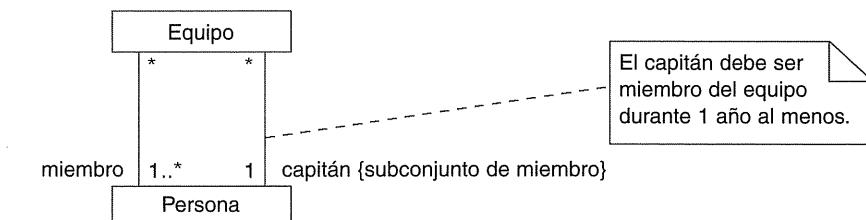


Figura 6.11: Modelado de nueva semántica.

Este diagrama muestra que cada Persona puede ser miembro de cero o más Equipos y que cada Equipo debe tener al menos una Persona como

miembro. El diagrama además indica que cada Equipo debe tener exactamente una Persona como capitán y cada Persona puede ser el capitán de cero o más Equipos. Toda esta semántica puede expresarse con el UML básico. Sin embargo, afirmar que un capitán también debe ser miembro de un equipo es algo que afecta a varias asociaciones y no puede expresarse con el UML básico. Para especificar este invariante, hay que escribir una restricción que muestre al capitán como un subconjunto de los miembros del Equipo, conectando las dos asociaciones con una restricción. También hay una restricción que indica que el capitán debe tener una antigüedad de al menos un año en el equipo.

Sugerencias y consejos

Cuando se adorne un modelo con notas:

- Hay que usar las notas sólo para los requisitos, observaciones, revisiones y explicaciones que no se puedan expresar de un modo simple y significativo con las características existentes de UML.
- Hay que usar las notas como un tipo de notas *post-it* electrónicas, para seguir los pasos del trabajo en desarrollo.

Cuando se dibujen notas:

- No hay que complicar los modelos con grandes bloques de comentarios. En vez de ello, si se necesita realmente un comentario largo, deben usarse las notas como lugar donde enlazar o incluir un documento que contenga el comentario completo.

Cuando se extienda un modelo con estereotipos, valores etiquetados o restricciones:

- Hay que homologar un conjunto pequeño de estereotipos, valores etiquetados y restricciones para usar en el proyecto y evitar que los desarrolladores creen a título individual muchas extensiones nuevas.
- Hay que elegir nombres cortos y significativos para los estereotipos y los valores etiquetados.
- Donde se pueda relajar la precisión, hay que usar texto libre para especificar las restricciones. Si es necesario un mayor rigor, puede usarse OCL para escribir expresiones de restricciones.

Cuando se dibuje un estereotipo, un valor etiquetado o una restricción:

- Hay que usar los estereotipos gráficos de forma moderada. Se puede cambiar totalmente la notación básica de UML con estereotipos, pero así se hará imposible que nadie pueda entender los modelos.
- Hay que considerar el empleo del color y del sombreado para los estereotipos gráficos, así como iconos más complejos. Las notaciones sencillas suelen ser generalmente las mejores, e incluso las señales visuales más sutiles pueden servir muy bien para comunicar el significado.

En este capítulo

- Diagramas, vistas y modelos.
- Modelado de las diferentes vistas de un sistema.
- Modelado de diferentes niveles de abstracción.
- Modelado de vistas complejas.
- Organización de diagramas y otros artefactos.

El modelado se discute en el Capítulo 1.

Cuando se modela algo, se crea una simplificación de la realidad para comprender mejor el sistema que se está desarrollando. Con UML, se construyen modelos a partir de bloques de construcción básicos, tales como clases, interfaces, colaboraciones, componentes, nodos, dependencias, generalizaciones y asociaciones.

Los diagramas son los medios para ver estos bloques de construcción. Un diagrama es una presentación gráfica de un conjunto de elementos, que la mayoría de las veces se dibuja como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se utilizan para visualizar un sistema desde diferentes perspectivas. Como ningún sistema complejo puede ser comprendido completamente desde una única perspectiva, UML define varios diagramas que permiten centrarse en diferentes aspectos del sistema independientemente.

Los buenos diagramas hacen comprensible y accesible el sistema. La elección del conjunto adecuado de diagramas para modelar un sistema obliga a plantearse las cuestiones apropiadas sobre el sistema y ayuda a clarificar las implicaciones de las decisiones.

Introducción

Cuando se trabaja con un arquitecto para diseñar una casa, se comienza con tres cosas: una lista de necesidades (tales como “Quiero una casa con tres dormitorios” y “No quiero pagar más de x”), unos cuantos bocetos o imágenes de otras casas, representando algunas de sus características más significativas (como una imagen de una entrada con una escalera de caracol), y una idea general del estilo (“Nos gustaría un estilo francés con toques del estilo de la costa de California”). El trabajo del arquitecto es recoger estos requisitos incompletos, cambiantes y posiblemente contradictorios y convertirlos en un diseño.

Para hacer esto, el arquitecto probablemente comenzará con un plano de la planta. Este artefacto proporciona un vehículo para que el cliente y el arquitecto puedan imaginar la casa final, especificar detalles y documentar decisiones. En cada revisión será necesario hacer algunos cambios, como cambiar paredes de sitio, reorganizar las habitaciones y colocar ventanas y puertas. Estos planos suelen cambiar en poco tiempo. Conforme madura el diseño y el cliente se siente satisfecho de tener el diseño que mejor se adapta a las restricciones de forma, función, tiempo y dinero, estos planos se estabilizarán hasta el punto en que se pueden emplear para construir la casa. Pero incluso mientras la casa se está construyendo, es probable que se cambien algunos de estos diagramas y se creen otros nuevos.

Más adelante, el cliente deseará tener otras vistas de la casa aparte del plano de la planta. Por ejemplo, querrá ver un plano del alzado, que muestre la casa desde diferentes lados. Mientras el cliente empieza a especificar detalles para que el trabajo pueda ser presupuestado, el arquitecto necesitará realizar planos de electricidad, planos de ventilación y calefacción y planos de fontanería y alcantarillado. Si el diseño requiere alguna característica inusual (como un sótano con una gran superficie libre de pilares) o el cliente desea alguna característica importante (como la ubicación de una chimenea), el arquitecto y el cliente necesitarán algunos bocetos para resaltar estos detalles.

La práctica de realizar diagramas para visualizar sistemas desde diferentes perspectivas no se limita a la industria de la construcción. Aparece en cualquier ingeniería que implique la construcción de sistemas complejos, como sucede en la ingeniería civil, la ingeniería aeronáutica, la ingeniería naval, la ingeniería de manufacturación y la ingeniería del software.

En el contexto del software hay cinco vistas complementarias que son las más importantes para visualizar, especificar, construir y documentar una arquitectura software: la vista de casos de uso, la vista de diseño, la vista de interacción,

Las cinco vistas de una arquitectura se discuten en el Capítulo 2.

El modelado de la arquitectura de un sistema se discute en el Capítulo 32.

Este proceso incremental e iterativo se resume en el Apéndice B.

la vista de implementación y la vista de despliegue. Cada una de estas vistas involucra modelado estructural (modelado de cosas estáticas), así como modelado de comportamiento (modelado de cosas dinámicas). Juntas, estas diferentes vistas capturan las decisiones más importantes sobre el sistema. Individualmente, cada una de estas vistas permite centrar la atención en una perspectiva del sistema para poder razonar con claridad sobre las decisiones.

Cuando se ve un sistema software desde cualquier perspectiva mediante UML, se usan los diagramas para organizar los elementos de interés. UML define diferentes tipos de diagramas, que se pueden mezclar y conectar para componer cada vista. Por ejemplo, los aspectos estáticos de la vista de implementación de un sistema pueden visualizarse con diagramas de clases; los aspectos dinámicos de la misma vista de implementación pueden visualizarse con diagramas de interacción.

Por supuesto, uno no está limitado a los tipos de diagramas predefinidos. En UML, estos tipos de diagramas se han definido porque representan el empaquetamiento más común de los elementos. Para ajustarse a las necesidades de un proyecto u organización, uno puede crearse sus propios diagramas para ver los elementos de UML de diferentes formas.

Los diagramas de UML se utilizarán de dos formas básicas: para especificar modelos a partir de los cuales construir un sistema ejecutable (ingeniería directa) y para reconstruir modelos a partir de partes de un sistema ejecutable (ingeniería inversa). En cualquier caso, al igual que un arquitecto, se tenderá a construir los diagramas incrementalmente (elaborándolos uno a uno) e iterativamente (repitiendo el proceso de diseñar un poco, construir un poco).

Términos y conceptos

Los sistemas, los modelos y las vistas se discuten en el Capítulo 32.

Un *sistema* es una colección de subsistemas organizados para lograr un propósito, descrito por un conjunto de modelos, posiblemente desde diferentes puntos de vista. Un *subsistema* es un grupo de elementos, algunos de los cuales constituyen una especificación del comportamiento ofrecido por los otros. Un *modelo* es una abstracción semánticamente cerrada de un sistema, es decir, representa una simplificación completa y autoconsistente de la realidad, creado para comprender mejor el sistema. En el contexto de la arquitectura, una *vista* es una proyección de la organización y estructura de un modelo del sistema, centrada en un aspecto del sistema. Un *diagrama* es la representación gráfica de un conjunto de elementos, normalmente mostrado como un grafo conexo de nodos (elementos) y arcos (relaciones).

Para decirlo de otra forma, un sistema representa la cosa que se está desarrollando, vista desde diferentes perspectivas mediante distintos modelos, y con esas vistas presentadas en forma de diagramas.

Un diagrama es sólo una proyección gráfica de los elementos que configuran un sistema. Por ejemplo, se podrían tener cientos de clases en el diseño de un sistema de recursos humanos de una empresa. La estructura o el comportamiento de ese sistema no se podría percibir nunca mirando un gran diagrama con todas esas clases y relaciones. En cambio, sería preferible realizar varios diagramas, cada uno centrado en una vista. Por ejemplo, podría crearse un diagrama de clases que incluyese clases como Persona, Departamento y Oficina, agrupadas para formar el esquema de una base de datos. Se podría encontrar alguna de estas clases, junto con otras, en otro diagrama que representase una API usada por aplicaciones clientes. Probablemente, algunas de las clases mencionadas formarán parte de un diagrama de interacción, especificando la semántica de una transacción que reasigne una Persona a un nuevo Departamento.

Como muestra este ejemplo, un mismo elemento de un sistema (como la clase Persona) puede aparecer muchas veces en el mismo diagrama o incluso en diagramas diferentes. En cualquier caso, el elemento es el mismo. Cada diagrama ofrece una vista de los elementos que configuran el sistema.

Cuando se modelan sistemas reales, sea cual sea el dominio del problema, muchas veces se dibujan los mismos tipos de diagramas, porque representan vistas frecuentes de modelos habituales. Normalmente, las partes estáticas de un sistema se representarán mediante uno de los diagramas siguientes:

1. Diagramas de clases.
2. Diagramas de componentes.
3. Diagramas de estructura compuesta.
4. Diagramas de objetos.
5. Diagramas de despliegue.
6. Diagramas de artefactos.

A menudo se emplearán cinco diagramas adicionales para ver las partes dinámicas de un sistema:

1. Diagramas de casos de uso.
2. Diagramas de secuencia.
3. Diagramas de comunicación.
4. Diagramas de estados.
5. Diagramas de actividades.

Los paquetes se discuten en el Capítulo 12.

Cada diagrama que dibujemos pertenecerá probablemente a uno de estos once tipos u ocasionalmente a otro tipo, definido para un determinado proyecto u organización. Cada diagrama debe tener un nombre único en su contexto, para poder referirse a un diagrama específico y distinguir unos de otros. Para cualquier sistema, excepto los más triviales, los diagramas se organizarán en paquetes.

En un mismo diagrama se puede representar cualquier combinación de elementos de UML. Por ejemplo, se pueden mostrar clases y objetos (algo frecuente), o incluso se pueden mostrar clases y componentes en el mismo diagrama (algo legal, pero menos frecuente). Aunque no hay nada que impida que se coloquen elementos de modelado de categorías totalmente dispares en el mismo diagrama, lo más normal es que casi todos los elementos de modelado de un diagrama sean de los mismos tipos. De hecho, los diagramas definidos en UML se denominan según el elemento que aparece en ellos la mayoría de las veces. Por ejemplo, para visualizar un conjunto de clases y sus relaciones se utiliza un diagrama de clases. Análogamente, si se quiere visualizar un conjunto de componentes, se utilizará un diagrama de componentes.

Diagramas estructurales

Los diagramas estructurales de UML existen para visualizar, especificar, construir y documentar los aspectos estáticos de un sistema. Se pueden ver los aspectos estáticos de un sistema como aquellos que representan su esqueleto y su andamiaje, ambos relativamente estables. Así como el aspecto estático de una casa incluye la existencia y ubicación de paredes, puertas, ventanas, tuberías, cables y conductos de ventilación, también los aspectos estáticos de un sistema software incluyen la existencia y ubicación de clases, interfaces, colaboraciones, componentes y nodos.

Los diagramas estructurales de UML se organizan en líneas generales alrededor de los principales grupos de elementos que aparecen al modelar un sistema:

- | | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> 1. Diagramas de clases. 2. Diagramas de componentes. 3. Diagramas de estructura compuesta. 4. Diagramas de objetos. 5. Diagramas de artefactos. 6. Diagramas de despliegue. | Clases, interfaces y colaboraciones.
Componentes.
Estructura interna.
Objetos.
Artefactos.
Nodos. |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|

Los diagramas de clases se discuten en el Capítulo 8.

Diagramas de clases. Un *diagrama de clases* presenta un conjunto de clases, interfaces y colaboraciones, y las relaciones entre ellas. Los diagramas de clases son los diagramas más habituales en el modelado de sistemas orientados a objetos. Los diagramas de clases se utilizan para describir la vista de diseño estática de un sistema. Los diagramas de clases que incluyen clases activas se utilizan para cubrir la vista de procesos estática de un sistema.

Los diagramas de estructura compuesta y los diagramas de componentes se discuten en el Capítulo 15.

Diagramas de componentes. Un *diagrama de componentes* muestra las partes internas, los conectores y los puertos que implementan un componente. Cuando se instancia el componente, también se instancian las copias de sus partes internas.

Los diagramas de objetos se discuten en el Capítulo 14.

Diagramas de objetos. Un *diagrama de objetos* representa un conjunto de objetos y sus relaciones. Se utilizan para describir estructuras de datos, instantáneas estáticas de las instancias de los elementos existentes en los diagramas de clases. Los diagramas de objetos abarcan la vista de diseño estática o la vista de procesos estática de un sistema al igual que los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos.

Los diagramas de artefactos se discuten en el Capítulo 30.

Diagramas de artefactos. Un *diagrama de artefactos* muestra un conjunto de artefactos y sus relaciones con otros artefactos y con las clases a las que implementan. Los diagramas de artefactos se utilizan para mostrar las unidades físicas de implementación del sistema. (Para UML los artefactos son parte de los diagramas de despliegue, pero nosotros los separamos para facilitar la discusión).

Los diagramas de despliegue se discuten en el Capítulo 31.

Diagramas de despliegue. Un *diagrama de despliegue* muestra un conjunto de nodos y sus relaciones. Los diagramas de despliegue se utilizan para describir la vista de despliegue estática de una arquitectura. Los diagramas de despliegue se relacionan con los diagramas de componentes en que un nodo normalmente incluye uno o más componentes.

Nota: Existen algunas variantes frecuentes de estos cinco diagramas, denominados según su propósito principal. Por ejemplo, se podría crear un diagrama de subsistemas para ilustrar la descomposición estructural de un sistema en subsistemas. Un diagrama de subsistemas es simplemente un diagrama de clases que contiene, principalmente, subsistemas.

Los diagramas de secuencia se discuten en el Capítulo 19.

Diagramas de comportamiento

Los diagramas de comportamiento de UML se emplean para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Se pueden ver los aspectos dinámicos de un sistema como aquellos que representan sus partes mutables. Así como los aspectos dinámicos de una casa incluyen flujos de aire y el tránsito entre las habitaciones, los aspectos dinámicos de un sistema software involucran cosas tales como el flujo de mensajes a lo largo del tiempo y el movimiento físico de componentes en una red.

Los diagramas de comportamiento de UML se organizan en líneas generales alrededor de las formas principales en que se puede modelar la dinámica de un sistema:

1. Diagramas de casos de uso. Organiza los comportamientos del sistema.
2. Diagramas de secuencia. Centrados en la ordenación temporal de los mensajes.
3. Diagramas de comunicación. Centrados en la organización estructural de los objetos que envían y reciben mensajes.
4. Diagramas de estados. Centrados en el estado cambiante de un sistema dirigido por eventos.
5. Diagramas de actividades. Centrados en el flujo de control de actividades.

Los diagramas de casos de uso se discuten en el Capítulo 18.

Diagramas de casos de uso. Un *diagrama de casos de uso* representa un conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Los diagramas de casos de uso se utilizan para describir la vista de casos de uso estática de un sistema. Los diagramas de casos de uso son especialmente importantes para organizar y modelar el comportamiento de un sistema.

El nombre colectivo que se da a los diagramas de secuencia y los diagramas de comunicación es el de *diagramas de interacción*. Tanto los diagramas de secuencia como los diagramas de comunicación son diagramas de interacción, y un diagrama de interacción es o bien un diagrama de secuencia o bien un diagrama de colaboración. Estos diagramas comparten el mismo modelo subyacente, aunque en la práctica resaltan cosas diferentes. (Los diagramas de tiempo son otro tipo de diagrama de interacción que no se trata en este libro).

Diagramas de secuencia. Un *diagrama de secuencia* es un diagrama de interacción que resalta la ordenación temporal de los mensajes. Un diagrama

de secuencia presenta un conjunto de roles y los mensajes enviados y recibidos por las instancias que interpretan los roles. Los diagramas de secuencia se utilizan para describir la vista dinámica de un sistema.

Los diagramas de comunicación se discuten en el Capítulo 19.

Diagramas de comunicación. Un *diagrama de comunicación* es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Un diagrama de comunicación muestra un conjunto de roles, enlaces entre ellos y los mensajes enviados y recibidos por las instancias que interpretan esos roles. Los diagramas de comunicación se utilizan para describir la vista dinámica de un sistema.

Los diagramas de estados se discuten en el Capítulo 25.

Diagramas de estados. Un *diagrama de estados* representa una máquina de estados, constituida por estados, transiciones, eventos y actividades. Los diagramas de estados se utilizan para describir la vista dinámica de un sistema. Son especialmente importantes para modelar el comportamiento de una interfaz, una clase o una colaboración. Los diagramas de estados resaltan el comportamiento dirigido por eventos de un objeto, lo que es especialmente útil al modelar sistemas reactivos.

Los diagramas de actividades, un caso especial de los diagramas de estados, se discuten en el Capítulo 20.

Diagramas de actividades. Un *diagrama de actividades* muestra el flujo paso a paso en una computación. Una actividad muestra un conjunto de acciones, el flujo secuencial o ramificado de acción en acción, y los valores que son producidos o consumidos por las acciones. Los diagramas de actividades se utilizan para ilustrar la vista dinámica de un sistema. Además, estos diagramas son especialmente importantes para modelar la función de un sistema, así como para resaltar el flujo de control en la ejecución de un comportamiento.

Nota: Hay limitaciones prácticas obvias para describir algo inherentemente dinámico (el comportamiento de un sistema) a través de diagramas (artefactos inherentemente estáticos, especialmente cuando se dibujan en una hoja de papel, una pizarra o una servilleta). Al dibujarse sobre una pantalla de computador, hay posibilidades de animar los diagramas de comportamiento para que simulen un sistema ejecutable o reproduzcan el comportamiento real de un sistema en ejecución. UML permite crear diagramas dinámicos y usar color u otras señales visuales para “ejecutar” el diagrama. Algunas herramientas ya han demostrado este uso avanzado de UML.

Técnicas comunes de modelado

Modelado de diferentes vistas de un sistema

Cuando se modela un sistema desde diferentes vistas, de hecho se está construyendo el sistema simultáneamente desde múltiples dimensiones. Eligiendo un conjunto apropiado de vistas, se establece un proceso que obliga a plantearse buenas preguntas sobre el sistema y a identificar los riesgos que hay que afrontar. Si se eligen mal las vistas o si uno se concentra en una vista a expensas de las otras, se corre el riesgo de ocultar preguntas y demorar problemas que finalmente acabarán con cualquier posibilidad de éxito.

Para modelar un sistema desde diferentes vistas, es necesario:

- Decidir qué vistas se necesitan para expresar mejor la arquitectura del sistema e identificar los riesgos técnicos del proyecto. Las cinco vistas de una arquitectura descritas anteriormente son un buen punto de partida.
- Para cada una de estas vistas, decidir qué artefactos hay que crear para capturar los detalles esenciales. La mayoría de las veces, estos artefactos consistirán en varios diagramas UML.
- Como parte del proceso de planificación, decidir cuáles de estos diagramas se pondrán bajo algún tipo de control formal o semiformal. Éstos son los diagramas para los que se planificarán revisiones y se conservarán como documentación del proyecto.
- Tener en cuenta que habrá diagramas que se desecharán. Esos diagramas transitorios aún serán útiles para explorar las implicaciones de las decisiones y para experimentar con los cambios.

Por ejemplo, si se modela una simple aplicación monolítica que se ejecuta en una única máquina, se podría necesitar sólo el siguiente grupo de diagramas:

- | | |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> ■ Vista de casos de uso. ■ Vista de diseño. ■ Vista de interacción. ■ Vista de implementación. ■ Vista de despliegue. | <ul style="list-style-type: none"> Diagramas de casos de uso. Diagramas de clases (para modelado estructural). Diagramas de interacción (para modelado del comportamiento). Diagramas de estructura compuesta. No se requiere. |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Si el sistema es reactivo o si se centra en el flujo de procesos, quizás se desee incluir diagramas de estados y de actividades, respectivamente, para modelar el comportamiento del sistema.

De la misma manera, si se trata de un sistema cliente/servidor, quizás se desee incluir diagramas de componentes y diagramas de despliegue para modelar los detalles físicos del sistema.

Por último, si se está modelando un sistema complejo y distribuido, se necesitará emplear el conjunto completo de diagramas de UML para expresar la arquitectura del sistema y los riesgos técnicos del proyecto, como se muestra a continuación:

- Vista de casos de uso. Diagramas de casos de uso.
Diagramas de Secuencia.
- Vista de diseño. Diagramas de clases (para modelado estructural).
Diagramas de interacción (para modelado del comportamiento).
Diagramas de estados (para modelado del comportamiento).
Diagramas de actividades (para modelado del comportamiento).
- Vista de interacción. Diagramas de interacción (para modelado del comportamiento).
- Vista de implementación. Diagramas de clases.
Diagramas de estructura compuesta.
- Vista de despliegue. Diagramas de despliegue.

Modelado a diferentes niveles de abstracción

No sólo es necesario ver un sistema desde varios ángulos, sino que habrá situaciones en las que diferentes personas implicadas en el desarrollo necesiten la misma vista del sistema, pero a diferentes niveles de abstracción. Por ejemplo, dado un conjunto de clases que capturen el vocabulario del espacio del problema, un programador podría necesitar una vista detallada a nivel de atributos, operaciones y relaciones de cada clase. Por otro lado, un analista que esté inspeccionando varios escenarios de casos de uso junto a un usuario final, quizás necesite una vista mucho más escueta de las mismas clases. En este contexto, el programador

está trabajando a un nivel menor de abstracción, y el analista y el usuario final lo hacen a un nivel mayor, pero todos trabajan sobre el mismo modelo. De hecho, ya que los diagramas son sólo una presentación gráfica de los elementos que constituyen un modelo, se pueden crear varios diagramas para el mismo o diferentes modelos, cada uno ocultando o exponiendo diferentes conjuntos de esos elementos, y cada uno mostrando diferentes niveles de detalle.

Básicamente, hay dos formas de modelar un sistema a diferentes niveles de abstracción: presentando diagramas con diferentes niveles de detalle para el mismo modelo o creando modelos a diferentes niveles de abstracción con diagramas que se relacionan de un modelo a otro.

Para modelar un sistema a diferentes niveles de abstracción presentando diagramas con diferentes niveles de detalle:

- Hay que considerar las necesidades de las personas que utilizarán el diagrama, y comenzar con un modelo determinado.
- Si se va a usar el modelo para hacer una implementación, harán falta diagramas a un menor nivel de abstracción, que tendrán que revelar muchos detalles. Si se va a usar para presentar un modelo conceptual a un usuario final, harán falta diagramas a un mayor nivel de abstracción, que tendrán que ocultar muchos detalles.
- Según donde uno se ubique en este espectro de niveles de abstracción, hay que crear un diagrama del nivel de abstracción apropiado, ocultando o revelando las cuatro categorías siguientes de elementos del modelo:
 1. *Bloques de construcción y relaciones*: ocultar los que no sean relevantes para el objetivo del diagrama o las necesidades del usuario.
 2. *Adornos*: revelar sólo los adornos de los bloques y relaciones que sean esenciales para comprender el objetivo.
 3. *Flujo*: en el contexto de los diagramas de comportamiento, considerar sólo aquellos mensajes o transiciones esenciales para comprender el objetivo.
 4. *Estereotipos*: en el contexto de los estereotipos utilizados para clasificar listas de elementos, como atributos y operaciones, revelar sólo aquellos elementos estereotipados esenciales para comprender el objetivo.

Los mensajes se discuten en el Capítulo 16; las transiciones se discuten en el Capítulo 22; los estereotipos se discuten en el Capítulo 6.

La ventaja principal de este enfoque es que siempre se está modelando desde un repositorio semántico común. La principal desventaja de este enfoque es que los cambios de los diagramas a un nivel de abstracción pueden dejar obsoletos los diagramas a otros niveles de abstracción.

Las dependencias de traza se discuten en el Capítulo 32.

Los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27.

Los diagramas de interacción se discuten en el Capítulo 19.

Para modelar un sistema a diferentes niveles de abstracción mediante la creación de modelos a diferentes niveles de abstracción, es necesario:

- Considerar las necesidades de las personas que utilizarán el diagrama y decidir el nivel de abstracción que debería ver cada una, creando un modelo separado para cada nivel.
- En general, poblar los modelos a mayor nivel de abstracción con abstracciones simples, y los modelos a un nivel más bajo de abstracción con abstracciones detalladas. Establecer dependencias de traza entre los elementos relacionados de diferentes modelos.
- En la práctica, si se siguen las cinco vistas de una arquitectura, al modelar un sistema a diferentes niveles de abstracción, se producen cuatro situaciones con frecuencia:
 1. *Casos de uso y su realización*: los casos de uso en un modelo de casos de uso se corresponden con colaboraciones en un modelo de diseño.
 2. *Colaboraciones y su realización*: las colaboraciones se corresponden con una sociedad de clases que trabajan juntas para llevar a cabo la colaboración.
 3. *Componentes y su diseño*: los componentes en un modelo de implementación se corresponden con los elementos en un modelo de diseño.
 4. *Nodos y sus componentes*: los nodos en un modelo de despliegue se corresponden con componentes en un modelo de implementación.

La ventaja principal de este enfoque es que los diagramas a diferentes niveles de abstracción se mantienen poco acoplados. Esto significa que los cambios en un modelo tendrán poco efecto directo sobre los otros modelos. La desventaja principal es que se deben gastar recursos para mantener sincronizados estos modelos y sus diagramas. Esto es especialmente cierto cuando los modelos corren parejos con diferentes fases del ciclo de vida de desarrollo del software, como cuando se decide mantener un modelo de análisis separado de un modelo de diseño.

Por ejemplo, supongamos que estamos modelando un sistema para comercio sobre la Web (uno de los casos de uso principales de ese sistema sería la realización de un pedido). Un analista o un usuario final probablemente crearía algunos diagramas de interacción a un alto nivel de abstracción para mostrar la acción de realizar un pedido, como se muestra en la Figura 7.1.

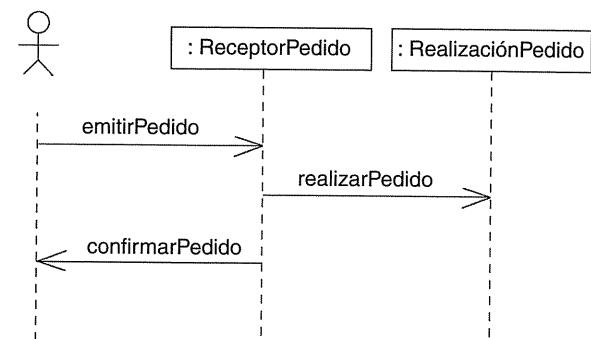


Figura 7.1: Diagrama de interacción a un alto nivel de abstracción.

Por otro lado, un programador responsable de implementar este escenario tendrá que trabajar sobre este diagrama, expandiendo ciertos mensajes y añadiendo otros actores en esta interacción, como se muestra en la Figura 7.2.

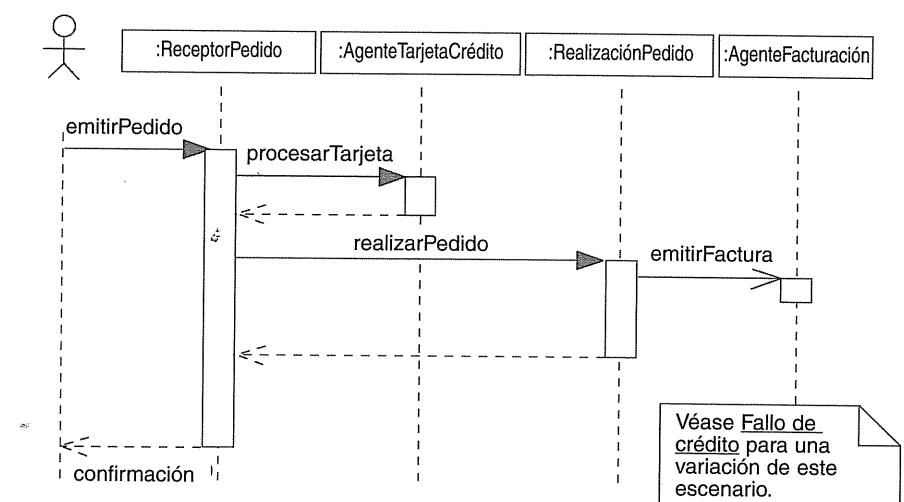


Figura 7.2: Interacción a un nivel de abstracción bajo.

Ambos diagramas corresponden al mismo modelo, pero a diferentes niveles de detalle. El segundo diagrama tiene mensajes y roles adicionales. Es razonable tener muchos diagramas como éstos, especialmente si las herramientas facilitan la navegación de un diagrama a otro.

Modelado de vistas complejas

Independientemente de cómo se descompongan los modelos, a veces se hace necesario crear diagramas grandes y complejos. Por ejemplo, si se desea analizar el esquema completo de una base de datos que contiene cien o más abstracciones, es realmente valioso estudiar un diagrama con todas estas clases y sus asociaciones. Al hacer esto, se podrán identificar patrones comunes de colaboración. Si se mostrase este diagrama a un mayor nivel de abstracción, omitiendo algunos detalles, se perdería la información necesaria para esta comprensión.

Para modelar vistas complejas:

Los paquetes se discuten en el Capítulo 12; las colaboraciones se discuten en el Capítulo 28.

- Primero, hay que asegurarse de que no existe una forma significativa de presentar esta información a mayor nivel de abstracción, quizás omitiendo algunas partes del diagrama y manteniendo los detalles en otras partes.
- Si se han omitido tantos detalles como es posible y el diagrama es aún complejo, hay que considerar la agrupación de algunos elementos en paquetes o en colaboraciones de mayor nivel, y luego mostrar sólo esos paquetes o colaboraciones en el diagrama.
- Si el diagrama es aún complejo, hay que usar notas y colores como señales visuales para atraer la atención del lector hacia los puntos deseados.
- Si el diagrama es aún complejo, hay que imprimirla completamente y colgarla en una gran pared. Se pierde la interactividad que proporciona una herramienta software, pero se puede mirar desde una cierta distancia y estudiarlo en busca de patrones comunes.

Sugerencias y consejos

Cuando se cree un diagrama:

- Hay que recordar que el propósito de un diagrama en UML no es dibujar bonitas imágenes, sino visualizar, especificar, construir y documentar. Los diagramas son un medio para el fin de implantar un sistema ejecutable.
- No hay por qué conservar todos los diagramas. Debe considerarse la construcción de diagramas sobre la marcha, inspeccionando los ele-

mentos en los modelos, y hay que utilizar esos diagramas para razonar sobre el sistema mientras se construye. Muchos de estos diagramas pueden desecharse después de haber servido a su propósito (pero la semántica bajo la que se crearon permanecerá como parte del modelo).

- Hay que evitar diagramas extraños o redundantes. Éstos complican los modelos.
- Hay que revelar sólo el detalle suficiente en cada diagrama para abordar las cuestiones para las que se pensó. La información extraña puede distraer al lector del punto clave que se desea resaltar.
- Por otro lado, no hay que hacer los diagramas minimalistas, a menos que realmente se necesite presentar algo a un nivel muy alto de abstracción. Simplificar en exceso puede ocultar detalles que tal vez sean importantes para razonar sobre los modelos.
- Hay que mantener un equilibrio entre los diagramas de comportamiento y los estructurales en el sistema. Muy pocos sistemas son totalmente estáticos o totalmente dinámicos.
- No hay que hacer diagramas demasiado grandes (los que ocupan varias páginas impresas son difíciles de navegar) ni demasiado pequeños (puede plantearse unir varios diagramas triviales en uno).
- Hay que dar a cada diagrama un nombre significativo que exprese su objetivo claramente.
- Hay que mantener organizados los diagramas. Deben agruparse en paquetes según la vista.
- No hay que obsesionarse con el formato de un diagrama. Hay que dejar que las herramientas ayuden.

Un diagrama bien estructurado:

- Se centra en comunicar un aspecto de la vista de un sistema.
- Contiene sólo aquellos elementos esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con su nivel de abstracción (muestra sólo aquellos adornos esenciales para su comprensión).
- No es tan minimalista que deje de informar al lector sobre la semántica importante.

Cuando se dibuje un diagrama:

- Hay que darle un nombre que comunique su propósito.
- Hay que ordenar sus elementos para minimizar los cruces de líneas.
- Hay que organizar sus elementos espacialmente para que las cosas cercanas semánticamente se coloquen cerca físicamente.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre las características importantes del diagrama. No obstante, el color debe usarse con cuidado, ya que algunas personas son daltónicas. El color sólo debe utilizarse para destacar algo, no para aportar información esencial.



Capítulo 8 DIAGRAMAS DE CLASES

En este capítulo

- Modelado de colaboraciones simples.
- Modelado de un esquema lógico de base de datos.
- Ingeniería directa e inversa.

Los diagramas de clases son los más utilizados en el modelado de sistemas orientados a objetos. Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones.

Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema. Esto incluye, principalmente, modelar el vocabulario del sistema, modelar las colaboraciones o modelar esquemas. Los diagramas de clases también son la base para un par de diagramas relacionados: los diagramas de componentes y los diagramas de despliegue.

Los diagramas de clases son importantes no sólo para visualizar, especificar y documentar modelos estructurales, sino también para construir sistemas ejecutables, aplicando ingeniería directa e inversa.

Introducción

Cuando se construye una casa, se comienza con un vocabulario que incluye bloques de construcción básicos, como paredes, suelos, ventanas, puertas, techos y vigas. Estos elementos son principalmente estructurales (las paredes tienen una altura, una anchura y un grosor), pero también tienen algo de comportamiento (los diferentes tipos de paredes pueden soportar diferentes cargas, las puertas se abren

y cierran, hay restricciones sobre la extensión de un suelo sin apoyo). De hecho, no se pueden considerar independientemente estas características estructurales y de comportamiento. Más bien, cuando uno construye su casa, debe considerar cómo interactúan. El proceso de diseñar una casa por parte de un arquitecto implica ensamblar estos elementos de forma única y satisfactoria que cumpla todos los requisitos funcionales y no funcionales del futuro inquilino. Los diseños que se crean para visualizar la casa y especificar sus detalles a la empresa constructora son representaciones gráficas de estos elementos y sus relaciones.

La construcción de software coincide en muchas de estas características con la construcción de una casa, excepto que, dada la fluidez del software, es posible definir los bloques de construcción básicos desde cero. Con UML, los diagramas de clases se emplean para visualizar el aspecto estático de estos bloques y sus relaciones y para especificar los detalles para construirlos, como se muestra en la Figura 8.1.

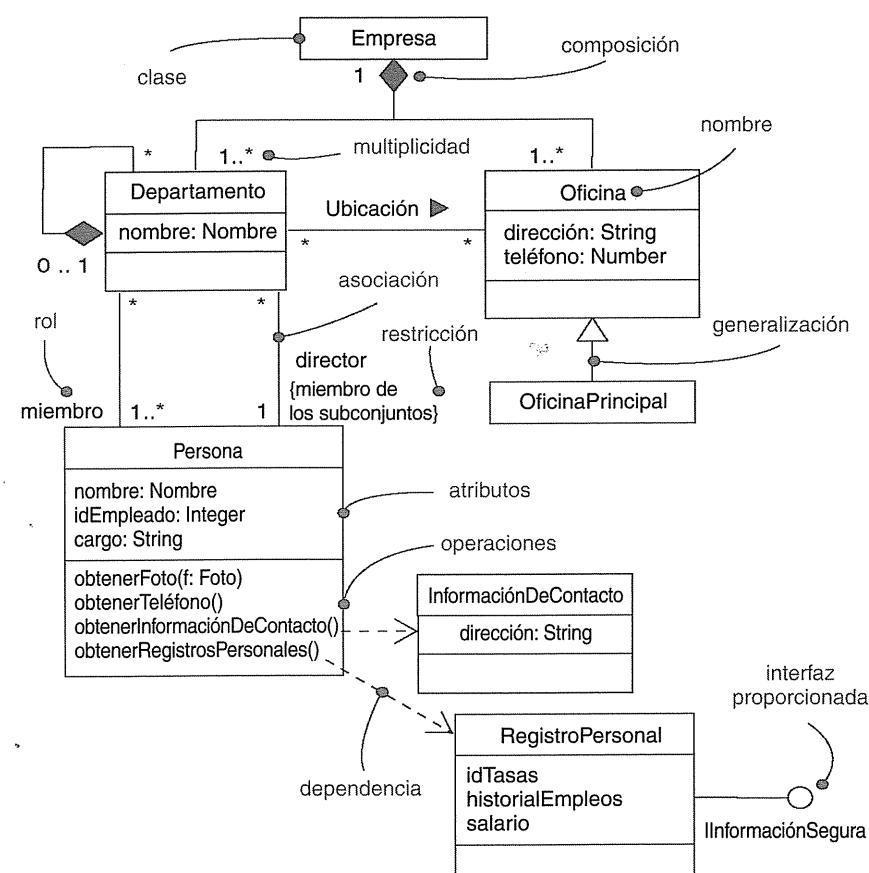


Figura 8.1: Un diagrama de clases.

Términos y conceptos

Un *diagrama de clases* es un diagrama que muestra un conjunto de interfaces, colaboraciones y sus relaciones. Gráficamente, un diagrama de clases es una colección de nodos y arcos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de clases es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de clases de los otros tipos de diagramas es su contenido particular.

Contenido

Los diagramas de clases contienen normalmente los siguientes elementos:

- Clases.
- Interfaces.
- Relaciones de dependencia, generalización y asociación.

Al igual que los demás diagramas, los diagramas de clases pueden contener notas y restricciones.

Los diagramas de clases también pueden contener paquetes o subsistemas, los cuales se usan para agrupar los elementos de un modelo en partes más grandes. A veces se colocarán instancias en los diagramas de clases, especialmente cuando se quiera mostrar el tipo (posiblemente dinámico) de una instancia.

Nota: Los diagramas de componentes y los diagramas de despliegue son similares a los diagramas de clases, excepto que en lugar de clases contienen componentes y nodos, respectivamente.

Las vistas de diseño se discuten en el Capítulo 2.

El modelado del vocabulario de un sistema se discute en el Capítulo 4.

Las colaboraciones se discuten en el Capítulo 28.

La persistencia se discute en el Capítulo 24; el modelado de bases de datos físicas se discute en el Capítulo 30.

Usos comunes

Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema. Esta vista soporta principalmente los requisitos funcionales de un sistema, los servicios que el sistema debe proporcionar a sus usuarios finales.

Cuando se modela la vista de diseño estática de un sistema, normalmente se utilizarán los diagramas de clases de una de estas tres formas:

1. Para modelar el vocabulario de un sistema.

El modelado del vocabulario de un sistema implica tomar decisiones sobre qué abstracciones son parte del sistema en consideración y cuáles caen fuera de sus límites. Los diagramas de clases se utilizan para especificar estas abstracciones y sus responsabilidades.

2. Para modelar colaboraciones simples.

Una colaboración es una sociedad de clases, interfaces y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de todos los elementos. Por ejemplo, cuando se modela la semántica de una transacción en un sistema distribuido, no se puede observar simplemente a una clase aislada para comprender qué ocurre. En vez de ello, la semántica la llevan a cabo un conjunto de clases que colaboran entre sí. Los diagramas de clases se emplean para visualizar y especificar este conjunto de clases y sus relaciones.

3. Para modelar un esquema lógico de base de datos.

Se puede pensar en un esquema como en un plano para el diseño conceptual de una base de datos. En muchos dominios se necesitará almacenar información persistente en una base de datos relacional o en una base de datos orientada a objetos. Se pueden modelar esquemas para estas bases de datos mediante diagramas de clases.

Técnicas comunes de modelado

Modelado de colaboraciones simples

Ninguna clase se encuentra aislada. En vez de ello, cada una trabaja en colaboración con otras para llevar a cabo alguna semántica mayor que la asociada a cada clase individual. Por tanto, aparte de capturar el vocabulario del sistema,

también hay que prestar atención a la visualización, especificación, construcción y documentación de la forma en que estos elementos del vocabulario colaboran entre sí. Estas colaboraciones se representan con los diagramas de clases.

Para modelar una colaboración:

- Hay que identificar los mecanismos que se quieren modelar. Un mecanismo representa una función o comportamiento de la parte del sistema que se está modelando que resulta de la interacción de una sociedad de clases, interfaces y otros elementos.
- Para cada mecanismo, hay que identificar las clases, interfaces y otras colaboraciones que participan en esta colaboración. Asimismo, hay que identificar las relaciones entre estos elementos.
- Hay que usar escenarios para recorrer la interacción entre estos elementos. Durante el recorrido, se descubrirán partes del modelo que faltaban y partes que eran semánticamente incorrectas.
- Hay que asegurarse de llenar estos elementos con su contenido. Para las clases, hay que comenzar obteniendo un reparto equilibrado de responsabilidades. Después, con el tiempo, hay que convertir éstas en atributos y operaciones concretos.

Por ejemplo, la Figura 8.2 muestra un conjunto de clases extraídas de la implementación de un robot autónomo. La figura se centra en las clases implicadas en el mecanismo para mover el robot a través de una trayectoria. Aparece una clase abstracta (`Motor`) con dos hijos concretos, `MotorDireccion` y `MotorPrincipal`. Ambas clases heredan las cinco operaciones de la clase padre, `Motor`. A su vez, las dos clases se muestran como partes de otra clase, `Conductor`. La clase `AgenteTrayectoria` tiene una asociación uno a uno con `Conductor` y una asociación uno a muchos con `SensorColision`. No se muestran atributos ni operaciones para `AgenteTrayectoria`, aunque sí se indican sus responsabilidades.

Hay muchas más clases implicadas en este sistema, pero este diagrama muestra sólo aquellas abstracciones implicadas directamente en mover el robot. Algunas de estas mismas clases aparecerán en otros diagramas. Por ejemplo, aunque no se muestre aquí, la clase `AgenteTrayectoria` colabora al menos con otras dos clases (`Entorno` y `AgenteObjetivo`) en un mecanismo de alto nivel para el manejo de los objetivos contrapuestos que el robot puede tener en un momento determinado. Análogamente, aunque tampoco se muestra aquí, las clases `SensorColision` y `Conductor` (y sus partes) colaboran con otra clase (`AgenteFallos`) en un mecanismo responsable de comprobar

constantemente el hardware del robot en busca de errores. Si cada una de estas colaboraciones es tratada en un diagrama diferente, se proporciona una vista comprensible del sistema desde diferentes perspectivas.

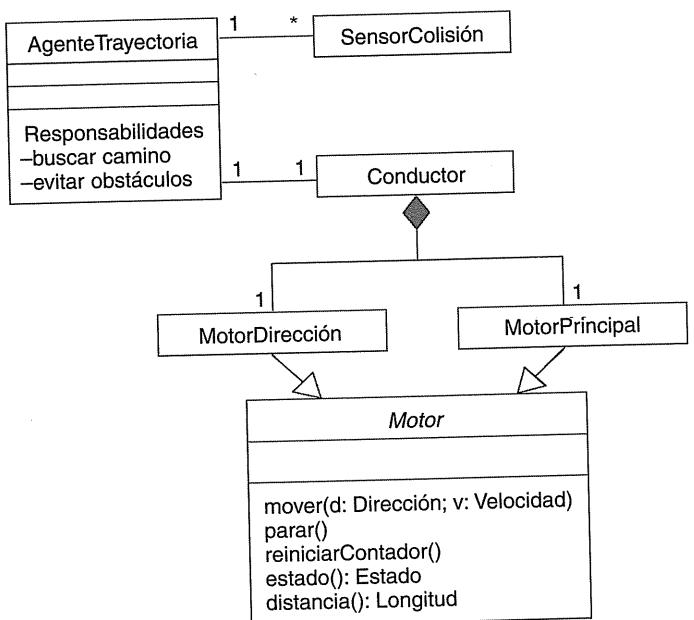


Figura 8.2: Modelado de colaboraciones simples.

El modelado de la distribución y los objetos se discute en el Capítulo 24; el modelado de bases de datos físicas se discute en el Capítulo 30.

Modelado de un esquema lógico de base de datos

Muchos de los sistemas que se modelen tendrán objetos persistentes, lo que significa que estos objetos podrán ser almacenados en una base de datos con el fin de poder recuperarlos posteriormente. La mayoría de las veces se empleará una base de datos relacional, una base de datos orientada a objetos o una base de datos híbrida objeto-relacional para el almacenamiento persistente. UML es apropiado para modelar esquemas lógicos de bases de datos, así como bases de datos físicas.

Los diagramas de clases de UML son un superconjunto de los diagramas entidad-relación (E-R), una herramienta de modelado para el diseño lógico de bases de datos utilizada con mucha frecuencia. Mientras que los diagramas E-R clásicos se centran sólo en los datos, los diagramas de clases van un paso más

Los estereotipos se discuten en el Capítulo 6.

allá, pues permiten el modelado del comportamiento. En la base de datos física, estas operaciones lógicas normalmente se convierten en disparadores (*triggers*) o procedimientos almacenados.

Para modelar un esquema:

- Hay que identificar aquellas clases del modelo cuyo estado debe trascender el tiempo de vida de las aplicaciones.
- Hay que crear un diagrama de clases que contenga estas clases. Se puede definir un conjunto propio de valores etiquetados para cubrir detalles específicos de bases de datos.
- Hay que expandir los detalles estructurales de estas clases. En general, esto significa especificar los detalles de sus atributos y centrar la atención en las asociaciones que estructuran estas clases y en sus cardinalidades.
- Hay que buscar patrones comunes que complican el diseño físico de bases de datos, tales como asociaciones cíclicas y asociaciones uno a uno. Donde sea necesario, deben crearse abstracciones intermedias para simplificar la estructura lógica.
- Hay que considerar también el comportamiento de las clases persistentes expandiendo las operaciones que sean importantes para el acceso a los datos y la integridad de éstos. En general, para proporcionar una mejor separación de intereses, las reglas del negocio relativas a la manipulación de conjuntos de estos objetos deberían encapsularse en una capa por encima de estas clases persistentes.
- Donde sea posible, hay que usar herramientas que ayuden a transformar un diseño lógico en un diseño físico.

Nota: El diseño lógico de bases de datos cae fuera del alcance de este libro. Aquí, el interés radica simplemente en mostrar cómo se pueden modelar esquemas mediante UML. En la práctica, el modelo se realizará con estereotipos adaptados al tipo de base de datos (relacional u orientada a objetos) que se esté utilizando.

La Figura 8.3 muestra un conjunto de clases extraídas de un sistema de información de una universidad. Esta figura es una extensión de un diagrama de clases anterior, y ahora se muestran las clases a un nivel suficientemente detallado para construir una base de datos física. Comenzando por la parte inferior izquierda de este diagrama, se encuentran las clases Estudiante,

Curso y Profesor. Hay una asociación entre Estudiante y Curso, que especifica que los estudiantes asisten a los cursos. Además, cada estudiante puede asistir a cualquier número de cursos y cada curso puede tener cualquier número de estudiantes.

El modelado de tipos primitivos se discute en el Capítulo 4; la agregación se discute en los Capítulos 5 y 10.

Este diagrama muestra los atributos de las seis clases. Todos los atributos son de tipos primitivos. Cuando se modela un esquema, generalmente una relación con cualquier tipo no primitivo se modela mediante agregaciones explícitas en vez de con atributos.

Dos de estas clases (Universidad y Departamento) muestran varias operaciones para manipular sus partes. Estas operaciones se incluyen porque son importantes para mantener la integridad de los datos (añadir o eliminar un Departamento, por ejemplo, tendrá algunos efectos en cadena). Hay otras muchas operaciones que se podrían considerar para estas dos clases y para el resto, como consultar los requisitos de un curso antes de asignarle un estudiante. Éstas son más bien reglas de negocio en vez de operaciones para integridad de la base de datos, y por ello se deberán colocar a un nivel mayor de abstracción que este esquema.

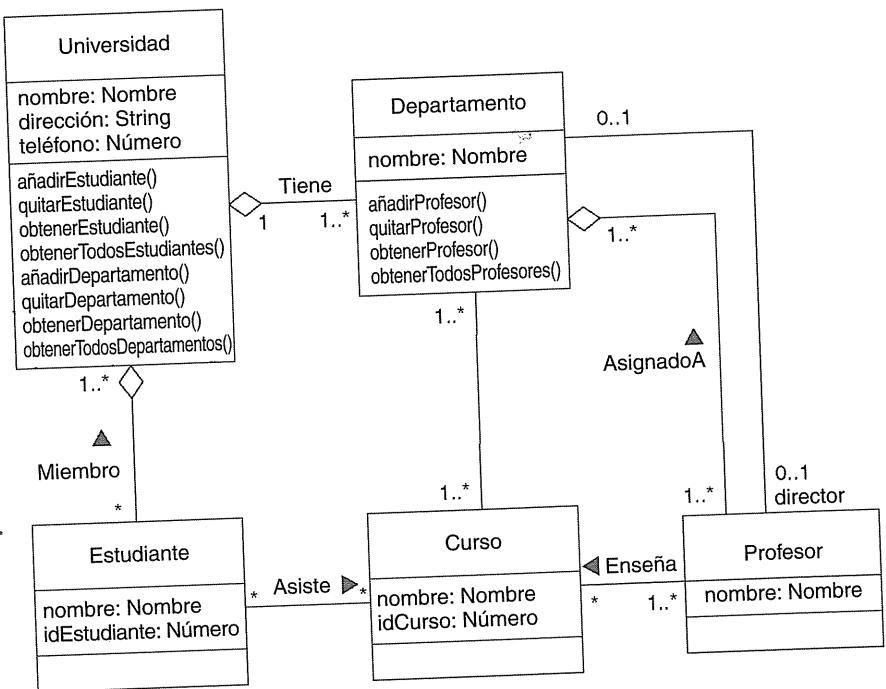


Figura 8.3: Modelado de un Esquema.

Ingeniería directa e inversa

La importancia del modelado se discute en el Capítulo 1.

Los diagramas de actividades se discuten en el Capítulo 20.

El modelado es importante, pero hay que recordar que el producto principal de un equipo de desarrollo es software, no diagramas. Por supuesto, la razón por la que se crean modelos es para entregar, en el momento oportuno, el software adecuado que satisfaga los objetivos siempre cambiantes de los usuarios y la empresa. Por esta razón, es importante que los modelos que se creen y las implementaciones que se desplieguen se correspondan entre sí, de forma que se minimice o incluso se elimine el coste de mantener sincronizados los modelos y las implementaciones.

Para algunos usos de UML, los modelos realizados nunca se corresponderán con un código. Por ejemplo, si se modela un proceso de negocio con diagramas de actividades, muchas de las actividades modeladas involucrarán a gente, no a computadores. En otros casos, se modelarán sistemas cuyas partes sean, desde un nivel dado de abstracción, una pieza de hardware (aunque a otro nivel de abstracción, seguro que este hardware puede contener un computador y software embebido).

En la mayoría de los casos, sin embargo, los modelos creados se corresponderán con código. UML no especifica ninguna correspondencia particular con ningún lenguaje de programación orientado a objetos, pero UML se diseñó con estas correspondencias en mente. Esto es especialmente cierto para los diagramas de clases, cuyos contenidos tienen una clara correspondencia con todos los lenguajes orientados a objetos importantes a nivel industrial, como Java, C++, Smalltalk, Eiffel, Ada, ObjectPascal y Forte. UML también fue diseñado para corresponderse con una variedad de lenguajes comerciales basados en objetos, como Visual Basic.

Los estereotipos y los valores etiquetados se discuten en el Capítulo 6.

Nota: La correspondencia de UML a lenguajes de implementación específicos para realizar ingeniería directa e inversa cae fuera del alcance de este libro. En la práctica, se terminará utilizando estereotipos y valores etiquetados adaptados al lenguaje de programación que se esté empleando.

La *ingeniería directa* es el proceso de transformar un modelo en código a través de una correspondencia con un lenguaje de implementación. La ingeniería directa produce una pérdida de información, porque los modelos escritos en UML son semánticamente más ricos que cualquier lenguaje de programación orientado a objetos actual. De hecho, ésta es una de las razones principales por las que se necesitan modelos además del código. Las características estructurales, como las colaboraciones, y las características de comportamiento, como las interacciones, pueden visualizarse claramente en UML, pero no tan claramente a partir de simple código fuente.

Para hacer ingeniería directa con un diagrama de clases:

- Hay que identificar las reglas para la correspondencia al lenguaje o lenguajes de implementación elegidos. Esto es algo que se hará de forma global para el proyecto o la organización.
- Según la semántica de los lenguajes escogidos, quizás haya que restringir el uso de ciertas características de UML. Por ejemplo, UML permite modelar herencia múltiple, pero Smalltalk sólo permite herencia simple. Se puede optar por prohibir a los desarrolladores el modelado con herencia múltiple (lo que hace a los modelos dependientes del lenguaje) o desarrollar construcciones específicas del lenguaje de implementación para transformar estas características más ricas (lo que hace la correspondencia más compleja).
- Hay que usar valores etiquetados para guiar las decisiones de implementación en el lenguaje destino. Esto se puede hacer a nivel de clases individuales si es necesario un control más preciso. También se puede hacer a un nivel mayor, como colaboraciones o paquetes.
- Hay que usar herramientas para generar código.

Los patrones se discuten en el Capítulo 29.

La Figura 8.4 ilustra un sencillo diagrama de clases que especifica una instancia del patrón cadena de responsabilidad. Esta instancia particular involucra a tres clases: Cliente, GestorEventos y GestorEventosGUI. Cliente y GestorEventos se representan como clases abstractas, mientras que GestorEventosGUI es concreta. GestorEventos tiene la operación que normalmente se espera en este patrón (`gestionarSolicitud()`), aunque se han añadido dos atributos privados para esta instanciaación.

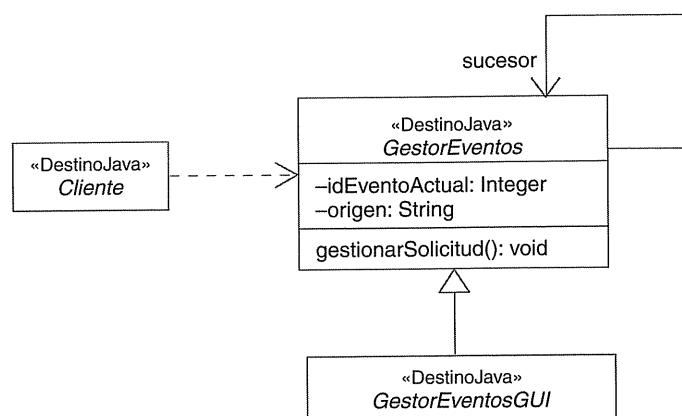


Figura 8.4: Ingeniería directa.

Todas estas clases especifican una correspondencia con Java, como se nota en su estereotipo. Usando una herramienta, la ingeniería directa de las clases de este diagrama a Java es inmediata. La ingeniería directa de la clase GestorEventos produce el siguiente código.

```

public abstract class GestorEventos {
    GestorEventos sucesor;
    private Integer idEventoActual;
    private String origen;

    GestorEventos () {}
    public void gestionarSolicitud() {}

}
  
```

La *ingeniería inversa* es el proceso de transformar código en un modelo a través de una correspondencia con un lenguaje de programación específico. La ingeniería inversa produce un aluvión de información, parte de la cual está a un nivel de detalle más bajo del que se necesita para construir modelos útiles. Al mismo tiempo, la ingeniería inversa es incompleta. Hay una pérdida de información cuando se hace ingeniería directa de modelos a código, así que no se puede recrear completamente un modelo a partir de código a menos que las herramientas incluyan información en los comentarios del código fuente que vaya más allá de la semántica del lenguaje de implementación.

Para hacer ingeniería inversa sobre un diagrama de clases:

- Hay que identificar las reglas para la correspondencia desde el lenguaje o lenguajes de implementación elegidos. Esto es algo que se hará de forma global para el proyecto o la organización.
- Con una herramienta, hay que indicar el código sobre el que se desea aplicar ingeniería inversa. Hay que usar la herramienta para generar un nuevo modelo o modificar uno existente al que se aplicó ingeniería directa previamente. No es razonable esperar que la ingeniería inversa produzca un único modelo conciso a partir de un gran bloque de código. Habrá que seleccionar una parte del código y construir el modelo desde la base.
- Con la herramienta, hay que crear un diagrama de clases inspeccionando el modelo. Por ejemplo, se puede comenzar con una o más clases, después expandir el diagrama, considerando relaciones específicas u otras clases vecinas. Se pueden mostrar u ocultar los detalles del contenido de este diagrama de clases, según sea necesario, para comunicar su propósito.

- La información de diseño se puede añadir manualmente al modelo, para expresar el objetivo del diseño que no se encuentra o está oculto en el código.

Sugerencias y consejos

Al realizar diagramas de clases en UML, debe recordarse que cada diagrama de clases es sólo una representación gráfica de la vista de diseño estática de un sistema. Ningún diagrama de clases individual necesita capturarlo todo sobre la vista de diseño de un sistema. En su conjunto, todos los diagramas de clases de un sistema representan la vista de diseño estática completa del sistema; individualmente, cada uno representa un aspecto.

Un diagrama de clases bien estructurado:

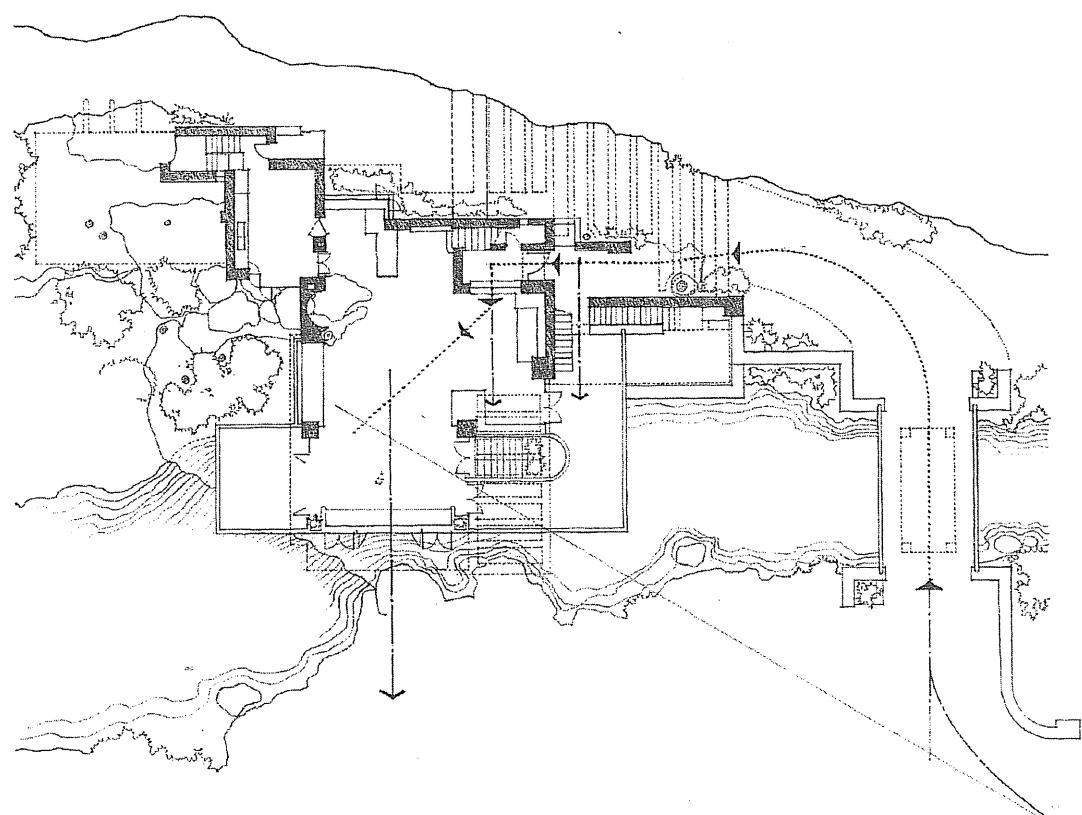
- Se centra en comunicar un aspecto de la vista de diseño estática de un sistema.
- Contiene sólo aquellos elementos que son esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con el nivel de abstracción, mostrando sólo aquellos adornos que sean esenciales para su comprensión.
- No es tan minimalista que deje de informar al lector sobre la semántica importante.

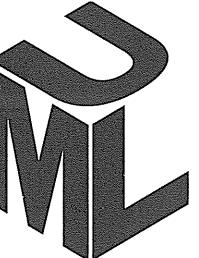
Cuando se dibuje un diagrama de clases:

- Hay que darle un nombre que comunique su propósito.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que organizar sus elementos espacialmente de modo que los que están cercanos semánticamente también lo estén físicamente.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre características importantes del diagrama.
- Hay que intentar no mostrar demasiados tipos de relaciones. En general, un tipo de relación tenderá a prevalecer en cada diagrama de clases.



Parte 3 MODELADO ESTRUCTURAL AVANZADO





Capítulo 9
**CARACTERÍSTICAS
AVANZADAS DE LAS CLASES**

LENGUAJE
UNIFICADO DE
MODELADO



En este capítulo

- Clasificadores, propiedades especiales de los atributos y las operaciones y diferentes tipos de clases.
- Modelado de la semántica de una clase.
- Elección del tipo apropiado de clasificador.

Realmente, las clases son el bloque de construcción más importante de cualquier sistema orientado a objetos. Sin embargo, las clases son sólo un tipo de un bloque de construcción más general de UML, los clasificadores. Un clasificador es un mecanismo que describe características estructurales y de comportamiento. Los clasificadores comprenden clases, interfaces, tipos de datos, señales, componentes, nodos, casos de uso y subsistemas.

Las propiedades básicas de los clasificadores se discuten en el Capítulo 4.

Los clasificadores (y especialmente las clases) tienen varias características avanzadas aparte de los simples atributos y operaciones descritos en la sección anterior. También se puede modelar la multiplicidad, la visibilidad, la firma, el polimorfismo y otras características. En UML se puede modelar la semántica de una clase, de forma que se puede enunciar su significado a cualquier grado de formalismo deseado.

En UML hay varios tipos de clasificadores y clases; es importante elegir el que mejor modele la abstracción del mundo real.

Introducción

La arquitectura se discute en el Capítulo 2.

Cuando se construye una casa, en algún momento del proyecto se toma una decisión sobre los materiales de construcción. Al principio, basta con indicar madera, piedra o acero. Este nivel de detalle es suficiente para continuar adelante. El material será elegido de acuerdo con los requisitos del proyecto (por ejem-

plo, acero y hormigón será una buena elección si se construye en una zona amenazada por huracanes). Según se avanza, el material elegido afectará a las siguientes decisiones de diseño (por ejemplo, elegir madera en vez de acero afectará a la masa que se puede soportar).

Conforme avanza el proyecto, habrá que refinar esas decisiones de diseño básicas y añadir más detalle para que un ingeniero de estructuras pueda validar la seguridad del diseño y para que un constructor pueda proceder a la construcción. Por ejemplo, puede que no sea suficiente especificar simplemente madera, sino indicar que sea madera de un cierto tipo que haya sido tratada para resistir a los insectos.

Las responsabilidades se discuten en el Capítulo 6.

Lo mismo ocurre al construir software. Al principio del proyecto, basta con decir que se incluirá una clase *Cliente* que se encarga de ciertas responsabilidades. Al refinar la arquitectura y pasar a la construcción, habrá que decidir una estructura para la clase (sus atributos) y un comportamiento (sus operaciones) que sean suficientes y necesarios para llevar a cabo esas responsabilidades. Por último, mientras se evoluciona hacia el sistema ejecutable, habrá que modelar detalles, como la visibilidad de los atributos individuales y de las operaciones, la semántica de concurrencia de la clase como un todo y de sus operaciones individuales, y las interfaces que la clase implementa.

Las ingenierías directa e inversa se discuten en los Capítulos 8, 14, 18, 19, 20, 25, 30 y 31.

UML proporciona una representación para varias características avanzadas, como se muestra en la Figura 9.1. Esta notación permite visualizar, especificar, construir y documentar una clase al nivel de detalle que se deseé, incluso el suficiente para soportar las ingenierías directa e inversa de modelos y de código.

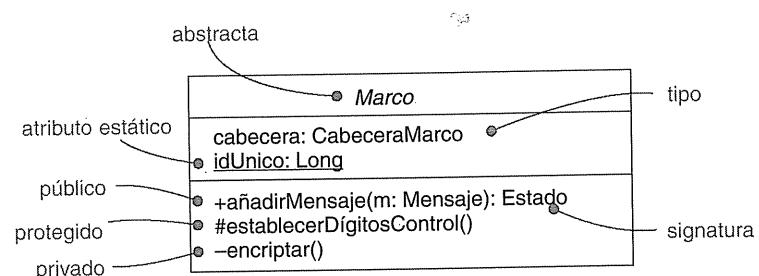


Figura 9.1: Características avanzadas de las clases.

Términos y conceptos

Un *clasificador* es un mecanismo que describe características estructurales y de comportamiento. Los clasificadores comprenden clases, interfaces, tipos de datos, señales, componentes, nodos, casos de uso y subsistemas.

Clasificadores

El modelado del vocabulario de un sistema se discute en el Capítulo 4; la dicotomía clase/objeto se discute en el Capítulo 2.

Las instancias se discuten en el Capítulo 13; los paquetes se discuten en el Capítulo 12; la generalización se discute en los Capítulos 5 y 10; las asociaciones se discuten en los Capítulos 5 y 10; los mensajes se discuten en el Capítulo 16; las interfaces se discuten en el Capítulo 11; los tipos de datos se discuten en los Capítulos 4 y 11; las señales se discuten en el Capítulo 21; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los casos de uso se discuten en el Capítulo 17; los subsistemas se discuten en el Capítulo 32.

Cuando se modela, se descubren abstracciones que representan cosas del mundo real y cosas de la solución. Por ejemplo, si se está construyendo un sistema de pedidos basado en la Web, el vocabulario del proyecto probablemente incluirá una clase *Cliente* (representa a las personas que hacen los pedidos) y una clase *Transacción* (un artefacto de implementación, que representa una acción atómica). En el sistema desarrollado podría haber un componente *Precios*, con instancias en cada nodo cliente. Cada una de estas abstracciones tendrá instancias; separar la esencia de la manifestación de las cosas del mundo real considerado es una parte importante del modelado.

Algunos elementos de UML no tienen instancias (por ejemplo, los paquetes y las relaciones de generalización). En general, aquellos elementos de modelado que pueden tener instancias se llaman clasificadores (las asociaciones y los mensajes también pueden tener instancias, pero sus instancias no son como las de una clase). Incluso más importante, un clasificador tiene características estructurales (en forma de atributos), así como características de comportamiento (en forma de operaciones). Cada instancia de un clasificador determinado comparte la definición de las mismas características, pero cada instancia tiene su propio valor para cada atributo.

El tipo más importante de clasificador en UML es la clase. Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Sin embargo, las clases no son el único tipo de clasificador. UML proporciona otros tipos de clasificadores para ayudar a modelar.

- **Interfaz** Una colección de operaciones que especifican un servicio de una clase o componente.
- **Tipo de datos** Un tipo cuyos valores son inmutables, incluyendo los tipos primitivos predefinidos (como números y cadenas de caracteres), así como los tipos enumerados (como los booleanos).
- **Asociación** Una descripción de un conjunto de enlaces, cada uno de los cuales relaciona a dos o más objetos.
- **Señal** La especificación de un mensaje asíncrono enviado entre instancias.
- **Componente** Una parte modular de un sistema que oculta su implementación tras un conjunto de interfaces externas.
- **Nodo** Un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, generalmente con alguna memoria y a menudo capacidad de procesamiento.

- Caso de uso Descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema y que produce un resultado observable de interés para un actor particular.
- Subsistema Un componente que representa una parte importante de un sistema.

La mayoría de los distintos clasificadores tienen características tanto estructurales como de comportamiento. Además, cuando se modela con cualquiera de estos clasificadores, se pueden usar todas las características avanzadas descritas en este capítulo para proporcionar el nivel de detalle necesario para capturar el significado de la abstracción.

Gráficamente, UML distingue entre estos diferentes clasificadores, como se muestra en la Figura 9.2.

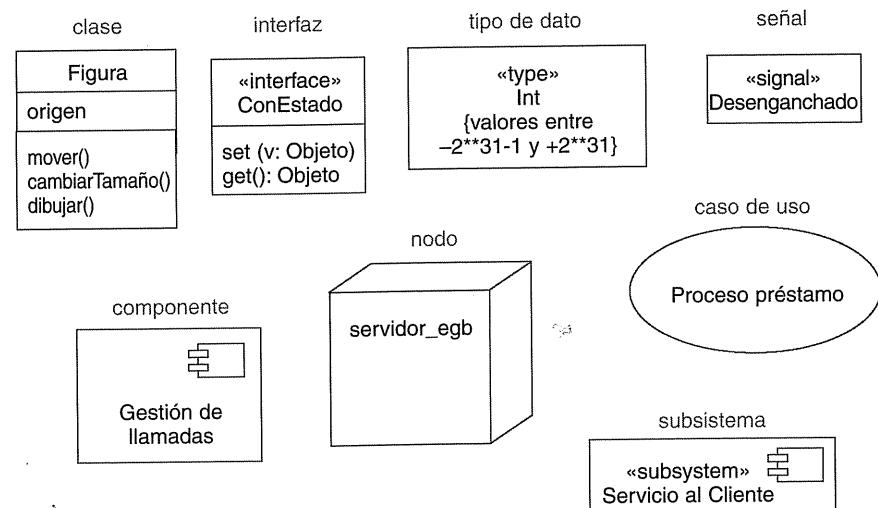


Figura 9.2: Clasificadores.

Nota: Un enfoque minimalista habría empleado un ícono para todos los clasificadores. Sin embargo, se consideró importante disponer de una señal visual. Análogamente, un enfoque maximalista habría utilizado diferentes iconos para cada tipo de clasificador. Esto tampoco tiene sentido porque, por ejemplo, las clases y los tipos de datos no son tan diferentes. El diseño de UML mantiene un equilibrio: unos clasificadores tienen su propio ícono, y otros utilizan palabras clave (como `type`, `signal` y `subsystem`).

Visibilidad

Uno de los detalles de diseño que se puede especificar para un atributo y operación es su visibilidad. La visibilidad de una característica especifica si puede ser utilizada por otros clasificadores. En UML se pueden especificar cuatro niveles de visibilidad.

- Un clasificador puede ver a otro clasificador si éste está en su alcance, y si existe una relación implícita o explícita hacia él; las relaciones se discuten en los Capítulos 5 y 10; los descendientes provienen de las relaciones de generalización, como se discute en el Capítulo 5; los permisos permiten a un clasificador compartir sus características privadas, como se discute en el Capítulo 10.
1. public Cualquier clasificador externo con visibilidad hacia el clasificador dado puede utilizar la característica; se especifica precediéndola del símbolo +.
 2. protected Cualquier descendiente del clasificador puede utilizar la característica; se especifica precediéndola del símbolo #.
 3. private Sólo el propio clasificador puede utilizar la característica; se especifica precediéndola del símbolo -.
 4. package Sólo los clasificadores declarados en el mismo paquete pueden utilizar la característica; se especifica precediéndola del símbolo ~.

La Figura 9.3 muestra una mezcla de características públicas, protegidas y privadas para la clase BarraHerramientas.

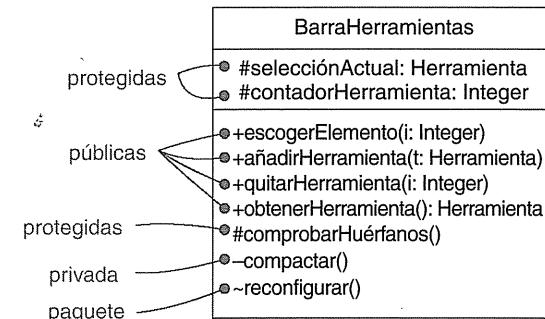


Figura 9.3: Visibilidad.

Cuando se especifica la visibilidad de las características de un clasificador, normalmente se ocultan los detalles de implementación y se muestran sólo aquellas características necesarias para llevar a cabo las responsabilidades de la abstracción. Ésta es la base del ocultamiento de información, esencial para construir sistemas sólidos y flexibles. Si no se adorna explícitamente una característica con un símbolo de visibilidad, por lo general se puede asumir que es pública.

Nota: La propiedad de visibilidad de UML tiene la semántica común en la mayoría de los lenguajes de programación orientados a objetos, incluyendo a C++, Java, Ada y Eiffel. Sin embargo, los lenguajes difieren ligeramente en su semántica de la visibilidad.

Alcance de instancia y estático

Las instancias se discuten en el Capítulo 13.

Otro detalle importante que se puede especificar para los atributos y operaciones de un clasificador es el alcance. El alcance de una característica especifica si cada instancia del clasificador tiene su propio valor de la característica, o si sólo hay un valor de la característica para todas las instancias del clasificador. En UML se pueden especificar dos tipos de alcances.

1. **instance** Cada instancia del clasificador tiene su propio valor para la característica. Éste es el valor por defecto y no requiere una notación adicional.
2. **static** Sólo hay un valor de la característica para todas las instancias del clasificador. También ha sido llamado *alcance de clase*. Esto se denota subrayando el nombre de la característica.

Como se ve en la Figura 9.4 (una simplificación de la primera figura), una característica con alcance estático se muestra subrayando su nombre. La ausencia de adorno significa que la característica tiene alcance de instancia.

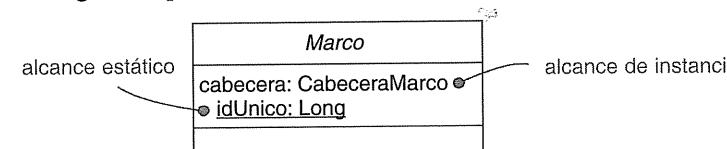


Figura 9.4: Alcance de propiedad.

En general, la mayoría de las características de los clasificadores modelados tendrán alcance de instancia. El uso más común de las características con alcance estático es utilizarlas para atributos privados que deben compartirse entre un conjunto de instancias, como sucede cuando es necesario generar identificadores únicos entre todas las instancias de una clase.

Nota: El alcance estático se corresponde con lo que en C++ y Java se llaman atributos y operaciones estáticos.

El alcance estático funciona de manera ligeramente distinta para las operaciones. Una operación de instancia tiene un parámetro implícito que se correspon-

de con el objeto que se está manipulando. Una operación estática no tiene ese parámetro; se comporta como un procedimiento global tradicional, sin un objeto destinatario. Las operaciones estáticas se utilizan para operaciones que crean instancias u operaciones que manipulan atributos estáticos.

Elementos abstractos, hojas y polimórficos

La generalización se discute en los Capítulos 5 y 10; las instancias se discuten en el Capítulo 13.

Las relaciones de generalización se utilizan para modelar jerarquías de clases, con las abstracciones más generales en la cima y las más específicas en el fondo. Dentro de estas jerarquías es frecuente especificar que ciertas clases son abstractas (es decir, que no pueden tener instancias directas). En UML se especifica que una clase es abstracta escribiendo su nombre en cursiva. Por ejemplo, como se muestra en la Figura 9.5, *Icono*, *IconoRectangular* e *IconoArbitrario* son todas ellas clases abstractas. En contraste, una clase concreta (como *Botón* y *BotónOK*) puede tener instancias directas.

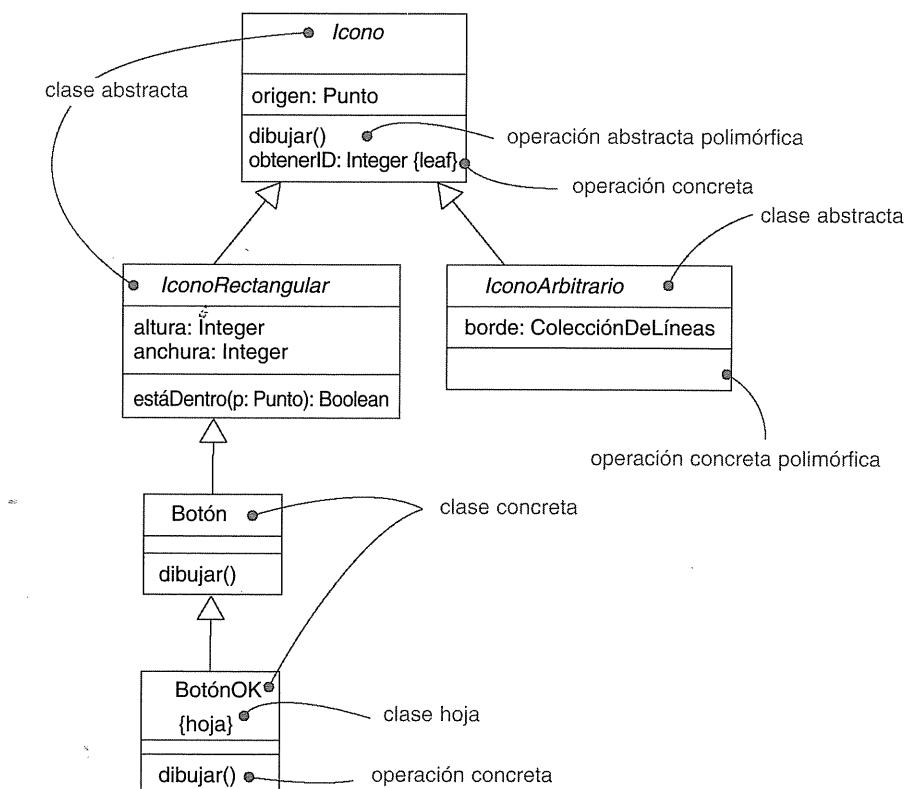


Figura 9.5: Clases y operaciones abstractas y concretas.

Los mensajes se discuten en el Capítulo 16.

Cuando se usa una clase, es probable que se desee heredar características de otras clases más generales, y también permitir que otras clases más específicas hereden características de ella. Ésta es la semántica normal que se tiene con las clases en UML. Sin embargo, también se puede especificar que una clase no puede tener hijos. Ese elemento se llama clase hoja y se especifica en UML escribiendo la propiedad `leaf` bajo el nombre de la clase. Por ejemplo, en la figura, `BotónOK` es una clase hoja, así que no puede tener hijos.

Las operaciones tienen propiedades similares. Normalmente, una operación es polimórfica, lo que significa que, en una jerarquía de clases, se pueden especificar operaciones con la misma firma en diferentes puntos de la jerarquía. Las operaciones de las clases hijas redefinen el comportamiento de las operaciones similares en las clases padres. Cuando se envía un mensaje en tiempo de ejecución, la operación de la jerarquía que se invoca se elige polimórficamente (es decir, el tipo del objeto receptor en tiempo de ejecución determina la elección). Por ejemplo, `dibujar` y `estáDentro` son ambas operaciones polimórficas. Además, la operación `Icono::dibujar()` es abstracta, lo que significa que es incompleta y necesita que una clase hija proporcione una implementación. En UML se especifica una operación abstracta escribiendo su nombre en cursiva, igual que se hace con una clase. Por el contrario, `Icono::obtenerId()` es una operación hoja, señalada así por la propiedad `leaf`. Esto significa que la operación no es polimórfica y no puede ser redefinida (esto es similar a una operación final en Java).

Nota: Las operaciones abstractas se corresponden con lo que en C++ se llaman operaciones virtuales puras; las operaciones hoja de UML se corresponden con las operaciones no virtuales de C++.

Multiplicidad

Las instancias se discuten en el Capítulo 13.

Cuando se utiliza una clase, es razonable asumir que puede haber cualquier número de instancias de ella (a menos que, por supuesto, sea una clase abstracta y, por tanto, no pueda tener instancias directas, aunque podría haber cualquier número de instancias de sus clases hijas concretas). A veces, no obstante, se desea restringir el número de instancias que puede tener una clase. Lo más frecuente es que se desee especificar cero instancias (en cuyo caso la clase es una clase utilidad que hace públicos sólo atributos y operaciones con alcance estático), una única instancia (una clase unitaria o *singleton*), un número específico de instancias o muchas instancias (el valor por omisión).

La multiplicidad se aplica también a las asociaciones, como se discute en los Capítulos 5 y 10.

Los atributos están relacionados con la semántica de la asociación, como se discute en el Capítulo 10.

Los clasificadores estructurados se discuten en el Capítulo 15.

El número de instancias que puede tener una clase es su multiplicidad. Ésta consiste en una especificación del rango de cardinalidades permitidas que puede asumir una entidad. En UML se puede especificar la multiplicidad de una clase con una expresión en la esquina superior derecha del icono de la clase. Por ejemplo, en la Figura 9.6, `ControladorRed` es una clase singleton. Del mismo modo, hay exactamente tres instancias de la clase `BarraControl` en el sistema.

La multiplicidad también se aplica a los atributos. Se puede especificar la multiplicidad de un atributo mediante una expresión adecuada encerrada entre corchetes tras el nombre del atributo. Por ejemplo, en la figura, hay dos o más instancias de `puertoConsola` en la instancia de `ControladorRed`.

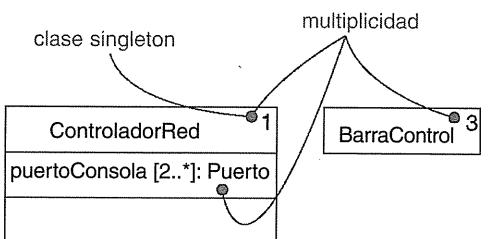


Figura 9.6: Multiplicidad.

Nota: La multiplicidad de una clase se aplica en un contexto determinado. Hay un contexto implícito para el sistema completo en el nivel superior. El sistema completo puede ser visto como un clasificador estructurado.

Atributos

Al nivel más abstracto, al modelar las características estructurales de una clase (es decir, sus atributos), simplemente se escribe el nombre de cada atributo. Normalmente esta información es suficiente para que el lector medio pueda comprender el propósito del modelo. No obstante, como se ha descrito en las secciones anteriores, también se puede especificar la visibilidad, el alcance y la multiplicidad de cada atributo. Aún hay más. También se puede especificar el tipo, el valor inicial y los cambios posibles de cada uno de ellos.

En su forma completa, la sintaxis de un atributo en UML es

```

[visibilidad] nombre
[':' tipo] ['[' multiplicidad '']']
['=' valor inicial]
[propiedad { ',' propiedad }]
  
```

Por ejemplo, las siguientes son declaraciones legales de atributos:

- **origen** Sólo el nombre.
- **+ origen** Visibilidad y nombre.
- **origen: Punto** Nombre y tipo.
- **nombre: String[0..1]** Nombre, multiplicidad y tipo.
- **origen: Punto = (0, 0)** Nombre, tipo y valor inicial.
- **id: Integer {readonly}** Nombre, tipo y propiedad.

A menos que se indique lo contrario, los atributos se pueden modificar siempre. Se puede utilizar la propiedad `readonly` para indicar que el valor del atributo no puede cambiar una vez que el objeto tome un valor inicial. Principalmente, se utilizará `readonly` para modelar constantes o atributos que se inicializan en el momento de creación de una instancia y no cambian a partir de ahí.

Nota: La propiedad `readonly` se corresponde con `const` en C++.

Operaciones

Las señales se discuten en el Capítulo 21.

Al nivel más abstracto, para modelar las características de comportamiento de una clase (sus operaciones y sus señales), simplemente se escribirá el nombre de cada operación. Normalmente, esta información es suficiente para que el lector medio pueda comprender el propósito del modelo. No obstante, como se ha descrito en las secciones anteriores, también se puede especificar la visibilidad y el alcance de cada operación. Aún hay más. También se pueden especificar los parámetros, el tipo de retorno, la semántica de concurrencia y otras propiedades de cada operación. El nombre de una operación junto a sus parámetros (incluido el tipo de retorno, si lo hay) se conoce como *signatura de la operación*.

Nota: UML distingue entre operación y método. Una operación especifica un servicio que se puede requerir de cualquier objeto de la clase para influir en su comportamiento; un método es una implementación de una operación. Cada operación no abstracta de una clase debe tener un método, el cual proporciona un algoritmo ejecutable como cuerpo (normalmente en algún lenguaje de programación o como texto estructurado). En una jerarquía de herencia, puede haber varios métodos para la misma operación, y el polimorfismo selecciona qué método de la jerarquía se ejecuta en tiempo de ejecución.

También se pueden utilizar los estereotipos para designar conjuntos de operaciones relacionadas, como las funciones auxiliares, como se discute en el Capítulo 6.

En su forma completa, la sintaxis de una operación en UML es

```
[visibilidad] nombre [(''lista de parametros '')]
[': tipo de retorno]
[propiedad {,' propriedad}]
```

Por ejemplo, las siguientes son declaraciones legales de operaciones:

- **mostrar** Sólo el nombre.
- **+ mostrar** Visibilidad y nombre.
- **set(n: Nombre, s: String)** Nombre y parámetros.
- **obtenerID(): Integer** Nombre y tipo de retorno.
- **reiniciar() {guarded}** Nombre y propiedad.

En la *signatura* de una operación se pueden proporcionar cero o más parámetros, donde cada uno sigue la siguiente sintaxis:

```
[direccion] nombre : tipo [= valor por defecto]
```

Dirección puede tomar uno de los siguientes valores:

- **in** Parámetro de entrada; no se puede modificar.
- **out** Parámetro de salida; puede modificarse para comunicar información al invocador.
- **inout** Parámetro de entrada; puede modificarse para comunicar información al invocador.

Nota: Un parámetro `out` o `inout` es equivalente a un parámetro de retorno y a un parámetro `in`. `Out` e `inout` se proporcionan por compatibilidad con lenguajes de programación antiguos. Es mejor usar parámetros de retorno explícitos.

Además de la propiedad `leaf` descrita antes, hay cuatro propiedades definidas que se pueden utilizar con las operaciones.

1. **query** La ejecución de la operación no cambia el estado del sistema. En otras palabras, la operación es una función pura sin efectos laterales.
2. **sequential** Los invocadores deben coordinarse para que en el objeto sólo haya un único flujo al mismo tiempo. En presencia de múltiples flujos de control, no se pueden garantizar ni la semántica ni la integridad del objeto.

3. guarded La semántica e integridad del objeto se garantizan en presencia de múltiples flujos de control por medio de la secuenciación de todas las llamadas a todas las operaciones *guarded* del objeto. Así, se puede invocar exactamente una operación a un mismo tiempo sobre el objeto, reduciendo esto a una semántica secuencial.
4. concurrent La semántica e integridad del objeto se garantizan en presencia de múltiples flujos de control, tratando la operación como atómica. Pueden ocurrir simultáneamente múltiples llamadas desde diferentes flujos de control a cualquier operación concurrente, y todas pueden ejecutarse concurrentemente con semántica correcta; las operaciones concurrentes deben diseñarse para ejecutarse correctamente en caso de que haya una operación secuencial concurrente o con guarda sobre el mismo objeto.
5. static La operación no tiene un parámetro implícito para el objeto destino; se comporta como un procedimiento global.

Los objetos activos, los procesos y los hilos se discuten en el Capítulo 23.

Las propiedades de concurrencia (*sequential*, *guarded*, *concurrent*) abarcan la semántica de concurrencia de una operación. Estas propiedades son relevantes sólo en presencia de objetos activos, procesos o hilos.

Clases plantilla (*Template*)

Las propiedades básicas de las clases se discuten en el Capítulo 4.

Una plantilla es un elemento parametrizado. En lenguajes como C++ y Ada, se pueden escribir clases plantilla, cada una de las cuales define una familia de clases (también se pueden escribir funciones plantilla, cada una de las cuales define una familia de funciones). Una plantilla incluye huecos para clases, objetos y valores, y estos huecos sirven como parámetros de la plantilla. Una plantilla no se puede utilizar directamente; antes hay que instanciarla. La instanciación implica ligar los parámetros formales con los reales. Para una clase plantilla, el resultado es una clase concreta que se puede emplear como cualquier otra clase.

El uso más frecuente de las plantillas es especificar contenedores que se pueden instanciar para elementos específicos, asegurándose de que sólo contendrá ele-

mentos del tipo indicado (*type-safe*). Por ejemplo, el siguiente fragmento de código C++ declara una clase parametrizada Map.

```
template<class Item, class Valor, int Cubeta>
class Map {
public:
    virtual Boolean map(const Item&, const VType&);
    virtual Boolean isMapped(const Item&) const;
    ...
};
```

Se podría entonces instanciar esta plantilla para hacer corresponder objetos Cliente con objetos Pedido.

```
m : Map<Cliente, Pedido, 3>;
```

También se pueden modelar las clases plantilla en UML. Como se muestra en la Figura 9.7, una clase plantilla se representa como una clase normal, pero con un recuadro discontinuo en la esquina superior derecha del icono de la clase, donde se listan los parámetros de la plantilla.

Las dependencias se discuten en los Capítulos 5 y 10; los estereotípos se discuten en el Capítulo 6.

Como se muestra en la figura, se puede modelar la instanciación de una plantilla de dos formas. La primera, de forma implícita, declarando una clase cuyo nombre proporcione la ligadura. La segunda, explícitamente, utilizando una dependencia estereotipada, como bind, que especifica que el origen instancia a la plantilla destino, usando los parámetros reales.

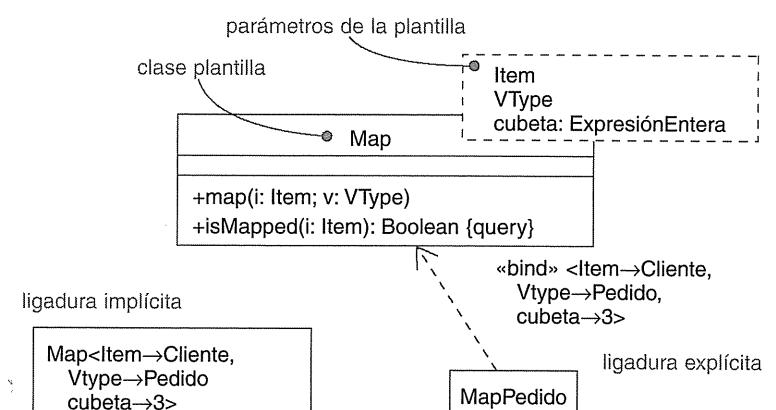


Figura 9.7: Clases plantilla.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Elementos estándar

Todos los mecanismos de extensibilidad de UML se aplican a las clases. Lo más frecuente es que se utilicen valores etiquetados para extender las propiedades de la clase (como especificar la versión de la clase) y estereotipos para especificar nuevos tipos de componentes (como los específicos del modelo).

UML define cuatro estereotipos estándar que se aplican a las clases:

1. **metaclass** Especifica un clasificador cuyos objetos son todos clases.
2. **powertype** Especifica un clasificador cuyos objetos son clases hijas de una clase padre específica.
3. **stereotype** Especifica que el clasificador es un estereotipo que se puede aplicar a otros elementos.
4. **utility** Especifica una clase cuyos atributos y operaciones tienen alcance estático.

Nota: Varios estereotipos estándar o palabras clave que se aplican a las clases se discuten en otras partes del texto.

Técnicas comunes de modelado

Modelado de la semántica de una clase

Los usos frecuentes de las clases se discuten en el Capítulo 4.

El modelado se discute en el Capítulo 1; también podemos modelar la semántica de una operación mediante un diagrama de actividades, como se discute en el Capítulo 20.

La mayoría de las veces, las clases se utilizan para modelar abstracciones extraídas del problema que se intenta resolver o de la tecnología utilizada para implementar una solución. Una vez identificadas estas abstracciones, lo siguiente que hay que hacer es especificar su semántica.

En UML se dispone de un amplio espectro de formas de hacer modelado, desde la más informal (responsabilidades) a la más formal (OCL, Lenguaje de Restricciones de Objetos). Dadas las alternativas, hay que decidir el nivel de detalle apropiado para comunicar el objetivo del modelo. Si el propósito es comunicarse con los usuarios finales y expertos en el dominio, se tenderá hacia la menos formal. Si el propósito es soportar la ingeniería “de ida y vuelta”, moviéndose entre los modelos y el código, se tenderá hacia algo más formal. Si el propósito es razonar rigurosa y matemáticamente sobre los modelos y demostrar su corrección, se tenderá a la más formal de todas.

Nota: Menos formal no significa menos precisa. Significa menos completa y detallada. Para ser práctico, habrá que mantener un equilibrio entre informal y muy formal. Esto significa que habrá que proporcionar suficiente detalle para soportar la creación de artefactos ejecutables, pero habrá que ocultar ciertos detalles para no sobrecargar al lector de los modelos.

Para modelar la semántica de una clase, se puede elegir entre las siguientes posibilidades, ordenadas de informal a formal.

Las responsabilidades se discuten en el Capítulo 4.

La especificación del cuerpo de un método se discute en el Capítulo 3.

La especificación de la semántica de una operación se discute en el Capítulo 20. Las máquinas de estados se discuten en el Capítulo 22; las colaboraciones se discuten en el Capítulo 28; las estructuras internas se discuten en el Capítulo 15; OCL se discute en The Unified Modeling Language Reference Manual.

- Especificar las responsabilidades de la clase. Una responsabilidad es un contrato u obligación de un tipo o clase y se representa en una nota junto a la clase, o en un compartimento extra en el icono de la clase.
- Especificar la semántica de la clase como un todo con texto estructurado, representado en una nota (con estereotipo **semantics**) junto a la clase.
- Especificar el cuerpo de cada método usando texto estructurado o un lenguaje de programación, representado en una nota, unida a la operación por una relación de dependencia.
- Especificar las pre y postcondiciones de cada operación, junto a los invariantes de la clase como un todo, usando texto estructurado. Estos elementos se representan en notas (con los estereotipos **precondition**, **postcondition** e **invariant**) unidas a la operación o clase por una relación de dependencia.
- Especificar una máquina de estados para la clase. Una máquina de estados es un comportamiento que especifica la secuencia de estados por los que pasa un objeto durante su vida en respuesta a eventos, junto con las respuestas a estos eventos.
- Especificar la estructura interna de la clase.
- Especificar una colaboración que represente a la clase. Una colaboración es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Una colaboración tiene una parte estructural y otra dinámica, así que se puede emplear para especificar todas las dimensiones de la semántica de una clase.
- Especificar las pre y poscondiciones de cada operación, además de los invariantes de la clase como un todo, con un lenguaje formal como OCL.

En la práctica, se suele terminar haciendo una combinación de estos enfoques para las diferentes abstracciones en el sistema.

Nota: Cuando se especifica la semántica de una clase, hay que tener presente si se pretende especificar lo que hace la clase o cómo lo hace. La especificación de la semántica de lo que hace la clase representa su vista pública, externa; especificar la semántica de cómo lo hace representa su vista privada, interna. Se utilizará una mezcla de ambas vistas, destacando la vista externa para los clientes de la clase y la vista interna para los implementadores.

Sugerencias y consejos

Al modelar clasificadores en UML, debe recordarse que hay un amplio rango de bloques de construcción disponibles, desde las interfaces y las clases hasta los componentes, etc. Se debe elegir, pues, el que mejor se adapte a la abstracción. Un clasificador bien estructurado:

- Tiene aspectos tanto estructurales como de comportamiento.
- Es fuertemente cohesivo y débilmente acoplado.
- Muestra sólo las características necesarias para que los clientes utilicen la clase, y oculta las demás.
- No debe ser ambiguo en su objetivo ni en su semántica.
- No debe estar tan sobreespecificado que elimine por completo la libertad de los implementadores.
- No debe estar tan poco especificado que deje ambigüedad en su significado.

Cuando se represente un clasificador en UML:

- Hay que mostrar sólo aquellas propiedades importantes para comprender la abstracción en su contexto.
- Hay que elegir una versión con estereotipo que proporcione la mejor señal visual de su propósito.



Capítulo 10

CARACTERÍSTICAS AVANZADAS DE LAS RELACIONES

En este capítulo

- Relaciones avanzadas: dependencia, generalización, asociación, realización y refinamiento.
- Modelado de redes de relaciones.
- Creación de redes de relaciones.

Las propiedades básicas de las relaciones se discuten en el Capítulo 5; las interfaces se discuten en el Capítulo 11; los componentes se discuten en el Capítulo 15; los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Al modelar los elementos que constituyen el vocabulario de un sistema, también hay que modelar cómo se relacionan entre sí estos elementos. Pero las relaciones pueden ser complejas. La visualización, la especificación y la documentación de complicadas redes de relaciones requieren varias características avanzadas.

Las dependencias, las generalizaciones y las asociaciones son los tres bloques de construcción de relaciones más importantes de UML. Estas relaciones tienen varias propiedades aparte de las descritas en la sección anterior. También se puede modelar herencia múltiple, navegación, composición, refinamiento y otras características. Un cuarto tipo de relación (la realización) permite modelar la conexión entre una interfaz y una clase o componente, o entre un caso de uso y una colaboración. En UML se puede modelar la semántica de las relaciones con cualquier grado de formalismo.

El manejo de redes complejas requiere usar las relaciones que sean apropiadas al nivel de detalle necesario, de modo que no se modele el sistema en exceso o por defecto.

Introducción

Los casos de uso y los escenarios se discuten en el Capítulo 17.

Al construir una casa, la decisión de dónde colocar cada habitación en relación con las demás es algo crítico. A cierto nivel de abstracción, se podría decidir colocar el dormitorio principal en la planta más baja, lejos de la fachada de la casa. A continuación, se podrían idear escenarios frecuentes para ayudar a razonar sobre esa disposición de la habitación. Por ejemplo, se consideraría la entrada de comestibles desde el garaje. No tendría sentido caminar desde el garaje a través del dormitorio hasta llegar a la cocina, y por tanto esta disposición se rechazaría.

Se puede formar una imagen bastante completa del plano de la casa pensando en función de estas relaciones básicas y casos de uso. Sin embargo, no es suficiente. Se puede acabar con verdaderos fallos de diseño si no se tienen en cuenta algunas relaciones más complejas.

Por ejemplo, puede que la disposición de habitaciones en cada planta parezca adecuada, pero las habitaciones de plantas distintas podrían interaccionar de formas imprevistas. Supongamos que se ha colocado la habitación de una hija adolescente justo encima del dormitorio de los padres, y que la hija decide aprender a tocar la batería. Este plano también sería rechazado, obviamente.

De la misma forma, hay que considerar cómo podrían interaccionar los mecanismos subyacentes de la casa con el plano de la planta. Por ejemplo, el coste de construcción se incrementará si no se organizan las habitaciones para tener paredes comunes en las que colocar las tuberías y desagües.

Las ingenierías directa e inversa se discuten en los Capítulos 8, 14, 18, 19, 20, 25, 30 y 31.

Lo mismo ocurre al construir software. Las dependencias, las generalizaciones y las asociaciones son las relaciones más comunes que aparecen al modelar sistemas con gran cantidad de software. Sin embargo, se necesitan varias características avanzadas de estas relaciones para capturar los detalles de muchos sistemas, detalles importantes a tener en cuenta para evitar verdaderos fallos en el diseño.

UML proporciona una representación para varias características avanzadas, como se muestra en la Figura 10.1. Esta notación permite visualizar, especificar, construir y documentar redes de relaciones al nivel de detalle que se deseé, incluso el necesario para soportar las ingenierías directa e inversa entre modelos y código.

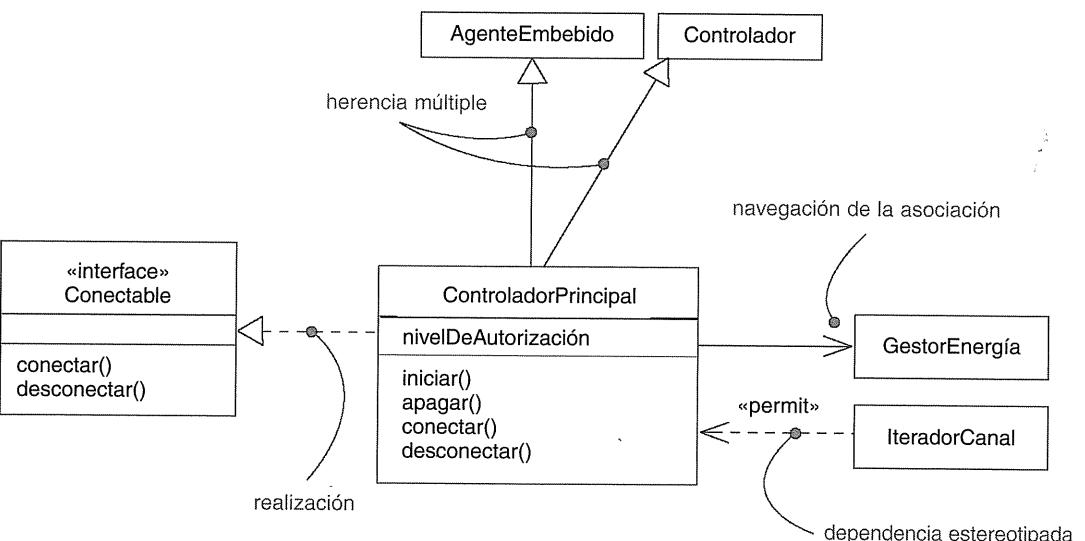


Figura 10.1: Características avanzadas de las relaciones.

Términos y conceptos

Una *relación* es una conexión entre elementos. En el modelado orientado a objetos, los cuatro tipos más importantes de relaciones son las dependencias, las generalizaciones, las asociaciones y las realizaciones. Gráficamente, una relación se dibuja como una línea, con diferentes tipos de líneas para distinguir las diferentes relaciones.

Dependencia

Una *dependencia* es una relación de uso, la cual especifica que un cambio en la especificación de un elemento (por ejemplo, la clase ControladorPrincipal) puede afectar a otro elemento que lo utiliza (por ejemplo, la clase IteradorCanal), pero no necesariamente a la inversa. Gráficamente, una dependencia se representa como una línea discontinua, dirigida hacia el elemento del que se depende. Las dependencias se deben aplicar cuando se quiera representar que un elemento utiliza a otro.

Las propiedades básicas de las dependencias se discuten en el Capítulo 5.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Una dependencia simple sin adornos suele bastar para la mayoría de las relaciones de uso que aparecen al modelar. Sin embargo, si se quieren especificar ciertos matices, UML define varios estereotipos que pueden aplicarse a las dependencias. Hay muchos de estos estereotipos, que se pueden organizar en varios grupos.

Los diagramas de clases se discuten en el Capítulo 8.

En primer lugar, hay estereotipos que se aplican a las dependencias entre clases y objetos en los diagramas de clases.

1. bind

Especifica que el origen de la dependencia instancia a la plantilla destino con los parámetros reales dados.

Las plantillas y las dependencias bind se discuten en el Capítulo 9.

Se utilizará bind para modelar los detalles de las clases plantilla. Por ejemplo, las relaciones entre una clase plantilla contenedora y una instancia de esa clase se modelarán como una dependencia bind. Bind incluye una lista de argumentos reales que se corresponden con los argumentos formales de la plantilla.

2. derive

Especifica que el origen puede calcularse a partir del destino.

Los atributos se discuten en los Capítulos 4 y 9; las asociaciones se discuten en el Capítulo 5 y más adelante en este mismo capítulo.

Se utilizará derive cuando se desee modelar la relaciones entre dos atributos o dos asociaciones, uno de los cuales sea concreto y el otro sea conceptual. Por ejemplo, una clase Persona podría tener el atributo FechaNacimiento (concreto), así como el atributo Edad (que se puede derivar de FechaNacimiento, de modo que no se manifiesta de forma separada en la clase). Se podría mostrar la relación entre Edad y FechaNacimiento con una dependencia derived, donde Edad derive de FechaNacimiento.

3. permit

Especifica que el origen tiene una visibilidad especial en el destino.

Las dependencias permit se discuten en el Capítulo 5.

Se utilizará permit para permitir a una clase acceder a características privadas de otra, como ocurre con las clases friend de C++.

4. instanceof

Especifica que el objeto origen es una instancia del clasificador destino. Normalmente se muestra de la forma origen: Destino.

5. instantiate

Especifica que el origen crea instancias del destino.

La dicotomía clase/objeto se discute en el Capítulo 2.

Estos dos últimos estereotipos permiten modelar relaciones clase/objeto explícitamente. Se utilizará instanceof para modelar la relación entre una clase y un objeto en el mismo diagrama, o entre una clase y su metaclasé; sin embargo, esto se muestra usando una sintaxis textual normalmente. Se empleará instantiate para especificar qué elemento crea objetos de otro.

6. powertype

Especifica que el destino es un supratípo (*powertype*) del origen; un supratípo es un clasificador cuyos objetos son todos los hijos de un padre dado.

El modelado lógico de bases de datos se discute en el Capítulo 8; el modelado físico de bases de datos se discute en el Capítulo 30.

Se utilizará powertype cuando se quiera modelar clases que clasifican a otras clases, como las que se encuentran al modelar bases de datos.

7. refine

Especifica que el origen está a un grado de abstracción más detallado que el destino.

Se utilizará refine cuando se quiera modelar clases que sean esencialmente la misma, pero a diferentes niveles de abstracción. Por ejemplo, durante el análisis, se podría encontrar una clase Cliente que durante el diseño se refinase en una clase Cliente más detallada, completada con su implementación.

8. use

Especifica que la semántica del elemento origen depende de la semántica de la parte pública del destino.

Los paquetes se discuten en el Capítulo 12.

Se aplicará use cuando se quiera etiquetar explícitamente una dependencia como una relación de uso, en contraste con otras formas de dependencia que proporcionan los otros estereotipos.

Hay dos estereotipos que se aplican a las dependencias entre paquetes:

1. import

Especifica que los contenidos públicos del paquete destino entran en el espacio de nombres público del origen, como si hubiesen sido declarados en el origen.

2. access

Especifica que los contenidos públicos del paquete destino entran en el espacio de nombres privado del origen. Los nombres sin calificar pueden usarse en el origen, pero no pueden volver a exportarse.

Los casos de uso se discuten en el Capítulo 17.

Se utilizarán access e import cuando se quiera usar elementos declarados en otros paquetes. Importar elementos evita la necesidad de usar un nombre completamente calificado para referenciar a un elemento desde otro paquete dentro de una expresión de texto.

Dos estereotipos se aplican a las relaciones de dependencia entre casos de uso:

1. extend

Especifica que el caso de uso destino extiende el comportamiento del origen.

2. include

Especifica que el caso de uso origen incorpora explícitamente el comportamiento de otro caso de uso en la posición especificada por el origen.

Se utilizarán **extend** e **include** (y generalización simple) para descomponer los casos de uso en partes reutilizables.

Las interacciones se discuten en el Capítulo 16.

En el contexto de la interacción entre objetos aparece un estereotipo:

- **send** Especifica que la clase origen envía el evento destino.

Las máquinas de estados se discuten en el Capítulo 22.

Se utilizará **send** cuando se quiera modelar una operación (como la que puede aparecer en la acción asociada a una transición de estado) que envía un evento dado a un objeto destino (que a su vez puede tener una máquina de estados asociada). De hecho, la dependencia **send** permite ligar máquinas de estados independientes.

Los sistemas y los modelos se discuten en el Capítulo 2.

Por último, hay un estereotipo que aparecerá en el contexto de la organización de los elementos de un sistema en subsistemas y modelos:

- **trace** Especifica que el destino es un antecesor histórico del origen, de una etapa previa del desarrollo.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2.

Se utilizará **trace** cuando se quiera modelar las relaciones entre elementos de diferentes modelos. Por ejemplo, en el contexto de la arquitectura de un sistema, un caso de uso en un modelo de casos de uso (que representa un requisito funcional) podría ser el origen de un paquete en el correspondiente modelo de diseño (que representa los artefactos que realizan el caso de uso).

Nota: Conceptualmente, todas las relaciones, incluidas la generalización, la asociación y la realización, son tipos de dependencias. La generalización, la asociación y la realización tienen una semántica lo bastante importante para justificar que se traten como distintos tipos de relaciones en UML. Los estereotipos anteriores representan matrizes de dependencias, cada uno de los cuales tiene su propia semántica, pero no están tan distantes semánticamente de las dependencias simples como para justificar que se traten como distintos tipos de relaciones. Ésta es una decisión de diseño no basada en reglas o principios por parte de UML, pero la experiencia muestra que este enfoque mantiene un equilibrio entre dar importancia a los tipos importantes de relaciones y no sobrecargar al modelador con demasiadas opciones. Uno no se equivocará si modela la generalización, la asociación y la realización en primer lugar, y luego ve las demás relaciones como tipos de dependencias.

Generalización

Las propiedades básicas de las generalizaciones se discuten en el Capítulo 5.

Una **generalización** es una relación entre un elemento general (llamado superclase o padre) y un tipo más específico de ese elemento (llamado subclase o hijo). Por ejemplo, se puede encontrar la clase general **Ventana** junto a un tipo más específico, **VentanaConPaneles**. Con una relación de generalización del hijo al padre, el hijo (**VentanaConPaneles**) heredará la estructura y comportamiento del padre (**Ventana**). El hijo puede añadir nueva estructura y comportamiento, o modificar el comportamiento del padre. En una generalización, las instancias del hijo pueden usarse donde quiera que se puedan usar las instancias del padre (o sea, el hijo es un sustituto del padre).

La herencia simple es suficiente la mayoría de las veces. Una clase que tenga únicamente un parente utiliza herencia simple. Hay veces, no obstante, en las que una clase incorpora aspectos de varias clases. En estos casos, la herencia múltiple modela mejor estas relaciones. Por ejemplo, la Figura 10.2 muestra un conjunto de clases de una aplicación de servicios financieros. Se muestra la clase **Activo** con tres hijos: **CuentaBancaria**, **Inmueble** y **Valor**. Dos de estos hijos (**CuentaBancaria** y **Valor**) tienen sus propios hijos. Por ejemplo, **Acción** y **Bono** son ambos hijos de **Valor**.

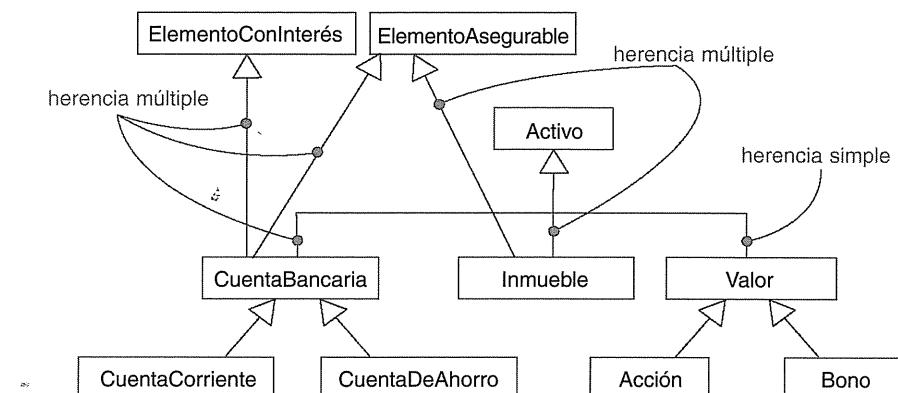


Figura 10.2: Herencia múltiple.

Dos de estos hijos (**CuentaBancaria** e **Inmueble**) heredan de varios padres. **Inmueble**, por ejemplo, es un tipo de **Activo**, así como un tipo de **Elemento-Asegurable**, y **CuentaBancaria** es un tipo de **Activo**, así como un **ElementoConInterés** y un **ElementoAsegurable**.

Algunas superclases se utilizan solamente para añadir comportamiento (normalmente) y estructura (ocasionalmente) a clases que heredan la estructura princi-

pal de superclases normales. Estas clases aditivas se denominan *mixins*; no se encuentran aisladas, sino que se utilizan como superclases suplementarias en una relación de herencia múltiple. Por ejemplo, *ElementoConInterés* y *ElementoAsegurable* son *mixins* en la Figura 10.2.

Nota: La herencia múltiple debe usarse con precaución. Si un hijo tiene varios padres cuya estructura o comportamiento se solapan habrá problemas. De hecho, en la mayoría de los casos, la herencia múltiple puede reemplazarse por la delegación, en la cual un hijo hereda de un único parente y emplea la agregación para obtener la estructura y comportamiento de los demás padres subordinados. Por ejemplo, en vez de especializar *Vehículo* en *VehículoTerrestre*, *VehículoAcuático* y *VehículoAéreo* por un lado, y en *PropulsiónGasolina*, *PropulsiónEólica* y *PropulsiónAnimal* por otro lado, se puede hacer que contenga una parte *medioDePropulsión*. El principal inconveniente de este enfoque es que se pierde la semántica de sustitución con estos padres subordinados.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Una generalización simple, sin adornos, es suficiente para la mayoría de las relaciones de herencia que aparecen al modelar. Pero si se quiere especificar ciertos matices, UML define cuatro restricciones que pueden aplicarse a las generalizaciones:

1. *complete* Especifica que todos los hijos en la generalización se han especificado en el modelo (aunque puede que algunos se omitan en el diagrama) y no se permiten hijos adicionales.
2. *incomplete* Especifica que no se han especificado todos los hijos en la generalización (incluso aunque se omitan algunos) y que se permiten hijos adicionales.

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

A menos que se indique lo contrario, se puede asumir que cualquier diagrama sólo muestra una vista parcial de una jerarquía de herencia y, por tanto, tiene omisiones. No obstante, la omisión es diferente de la completitud de un modelo. De forma específica, la restricción *complete* se usará para mostrar explícitamente que se ha especificado al completo una jerarquía en el modelo (aunque puede que ningún diagrama muestre esa jerarquía); se usará *incomplete* para mostrar de forma explícita que no se ha establecido la especificación completa de la jerarquía en el modelo (aunque un diagrama puede mostrar todo lo existente en el modelo).

3. *disjoint* Especifica que los objetos del parente no pueden tener más de uno de los hijos como tipo. Por ejemplo, la clase *Persona* puede especializarse en clases disjuntas *Hombre* y *Mujer*.
4. *overlapping* Especifica que los objetos del parente pueden tener más de uno de los hijos como tipo. Por ejemplo, la clase *Vehículo* puede especializarse en las subclases solapadas *VehículoTerrestre* y *VehículoAcuático* (un vehículo anfibio es ambas cosas a la vez).

Los tipos y las interfaces se discuten en el Capítulo 11; las interacciones se discuten en el Capítulo 16.

Estas dos restricciones sólo se aplican en el contexto de la herencia múltiple. Se utilizará *disjoint* para mostrar que las clases en un conjunto son mutuamente incompatibles; una subclase no puede heredar de más de una clase. Se utilizará *overlapping* cuando se quiera indicar que una clase puede realizar herencia múltiple de más de una clase del conjunto.

Nota: En la mayoría de los casos, un objeto tiene un tipo en tiempo de ejecución; éste es el caso de la clasificación estática. Si un objeto puede cambiar su tipo en tiempo de ejecución, es el caso de la clasificación dinámica. El modelado de la clasificación dinámica es complejo. Pero en UML se puede usar una combinación de herencia múltiple (para mostrar los tipos potenciales de un objeto) y tipos e interacciones (para mostrar cómo cambia de tipo un objeto en tiempo de ejecución).

Asociación

Las propiedades básicas de las asociaciones se discuten en el Capítulo 5.

Una *asociación* es una relación estructural que especifica que los objetos de un elemento se conectan con los objetos de otro. Por ejemplo, una clase *Biblioteca* podría tener una asociación uno-a-muchos con una clase *Libro*, indicando que cada instancia de *Libro* pertenece a una instancia de *Biblioteca*. Además, dado un *Libro*, se puede encontrar su *Biblioteca*, y dada una *Biblioteca* se puede navegar hacia todos sus *Libros*. Gráficamente, una asociación se representa con una línea continua entre la misma o diferentes clases. Las asociaciones se utilizan para mostrar relaciones estructurales.

Hay cuatro adornos básicos que se aplican a las asociaciones: nombre, rol en cada extremo de la asociación, multiplicidad en cada extremo y agregación. Para usos avanzados, hay otras muchas propiedades que permiten modelar de-

tales sutiles, como la navegación, la calificación y algunas variantes de la agregación.

Navegación. Dada una asociación simple, sin adornos, entre dos clases, como *Libro* y *Biblioteca*, es posible navegar de los objetos de un tipo a los del otro tipo. A menos que se indique lo contrario, la navegación a través de una asociación es bidireccional. Pero hay ciertas circunstancias en las que se desea limitar la navegación a una sola dirección. Por ejemplo, como se puede ver en la Figura 10.3, al modelar los servicios de un sistema operativo, se encuentra una asociación entre objetos *Usuario* y *Clave*. Dado un *Usuario*, se pueden encontrar las correspondientes *Claves*; pero, dada una *Clave*, no se podrá identificar al *Usuario* correspondiente. Se puede representar de forma explícita la dirección de la navegación con una flecha que apunte en la dirección de recorrido.

Nota: Especificar una dirección de recorrido no significa necesariamente que no se pueda llegar nunca de los objetos de un extremo a los del otro. En vez de ello, la navegación es un enunciado del conocimiento de una clase por parte de otra. Por ejemplo, en la figura anterior, aún sería posible encontrar los objetos *Usuario* asociados con una *Clave* a través de otras asociaciones que involucren a otras clases no mostradas en la figura. Especificar que una asociación es navegable es un enunciado de que, dado un objeto en un extremo, se puede llegar fácil y directamente a los objetos del otro extremo, normalmente debido a que el objeto inicial almacena algunas referencias a ellos.

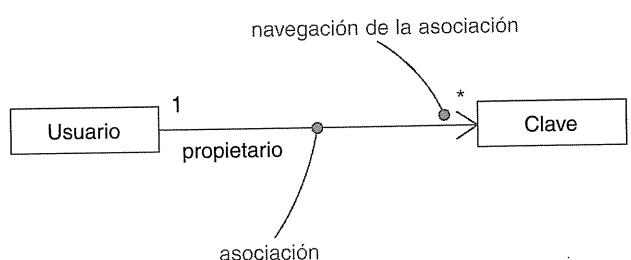


Figura 10.3: Navegación.

Las visibilidades públicas, protegidas, privadas y de paquete se discuten en el Capítulo 9.

Visibilidad. Dada una asociación entre dos clases, los objetos de una clase pueden ver y navegar hasta los objetos de la otra, a menos que se restrinja explícitamente por un enunciado explícito de navegación. Sin embargo, hay circunstancias en las que se quiere limitar la visibilidad a través de esta asociación relativa a los objetos externos a ella. Por ejemplo, como se muestra en la Figura

ra 10.4, hay una asociación entre *GrupoUsuarios* y *Usuario* y otra entre *Usuario* y *Clave*. Dado un objeto *Usuario*, es posible identificar sus correspondientes objetos *Clave*. Sin embargo, una *Clave* es privada a un *Usuario*, así que no debería ser accesible desde el exterior (a menos que el *Usuario* ofrezca acceso explícito a la *Clave*, quizás a través de alguna operación pública). Por lo tanto, como muestra la figura, dado un objeto *GrupoUsuarios*, se puede navegar a sus objetos *Usuario* (y viceversa), pero en cambio no se pueden ver los objetos *Clave* del objeto *Usuario*; son privados al *Usuario*. En UML se pueden especificar tres niveles de visibilidad para el extremo de una asociación, igual que para las características de una clase, adjuntando un símbolo de visibilidad al nombre de un rol. A menos que se indique lo contrario, la visibilidad de un rol es pública. La visibilidad privada indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación; la visibilidad protegida indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación, excepto los hijos del otro extremo. La visibilidad de paquete significa que las clases declaradas en el mismo paquete pueden ver al elemento dado; esto no se aplica a los extremos de asociación.

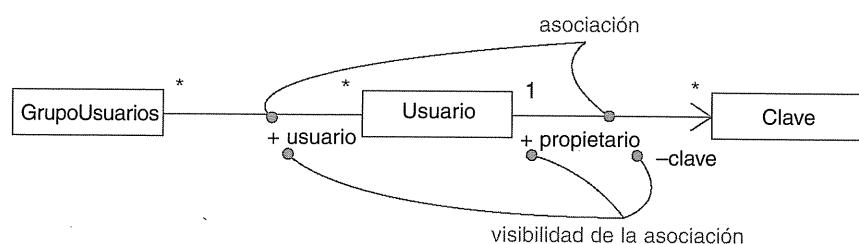


Figura 10.4: Visibilidad.

Los atributos se discuten en los Capítulos 4 y 9.

Calificación. En el contexto de una asociación, uno de los esquemas de modelado más frecuentes tiene que ver con el problema de las búsquedas. Dado un objeto en un extremo de una asociación, ¿cómo identificar un objeto o conjunto de objetos en el otro extremo? Por ejemplo, considérese el problema de modelar una mesa de trabajo en una fábrica, en la cual se arreglan los artículos devueltos. Como se representa en la Figura 10.5, se podría modelar una asociación entre dos clases, *MesaDeTrabajo* y *ArticuloDevuelto*. En el contexto de la *MesaDeTrabajo*, se tendrá un *IdTrabajo* que identificará un *ArticuloDevuelto* particular. En este sentido, *IdTrabajo* es un atributo de la asociación. No es una característica de *ArticuloDevuelto* porque los artículos realmente no tienen conocimiento de cosas como reparaciones o trabajos. Entonces, dado un objeto *MesaDeTrabajo* y dado un *IdTrabajo* particular, se puede navegar hasta cero o un objeto *ArticuloDevuelto*.

En UML se modela este esquema con un calificador, que es un atributo de una asociación cuyos valores identifican un subconjunto de objetos (normalmente un único objeto) relacionados con un objeto a través de una asociación. Un calificador se dibuja como un pequeño rectángulo junto al extremo de la asociación, con los atributos dentro, como se muestra en la figura. El objeto origen, junto a los valores de los atributos del calificador, devuelven un objeto destino (si la multiplicidad del destino es como máximo uno) o un conjunto de objetos (si la multiplicidad del destino es muchos).

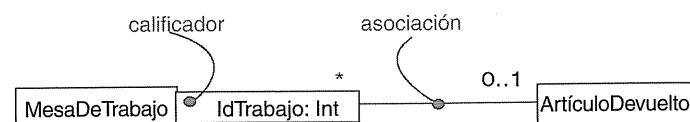


Figura 10.5: Calificación.

La agregación simple se discute en el Capítulo 5.

Composición. La agregación resulta ser un concepto simple con una semántica bastante profunda. La agregación simple es puramente conceptual y no hace más que distinguir un “todo” de una “parte”. La agregación simple no cambia el significado de la navegación a través de la asociación entre el todo y sus partes, ni liga las vidas del todo y las partes.

Un atributo es básicamente una forma abreviada para la composición; los atributos se discuten en los Capítulos 4 y 9.

Sin embargo, existe una variación de la agregación simple (la composición) que añade una semántica importante. La composición es una forma de agregación, con una fuerte relación de pertenencia y vidas coincidentes de la parte con el todo. Las partes con una multiplicidad no fijaada pueden crearse después de la parte compuesta a la que pertenecen, pero una vez creadas viven y mueren con ella. Tales partes también se pueden eliminar explícitamente antes de la eliminación de la parte compuesta.

Esto significa que, en una agregación compuesta, un objeto puede formar parte de sólo una parte compuesta a la vez. Por ejemplo, en un sistema de ventanas, un Marco pertenece exactamente a una Ventana. Esto contrasta con la agregación simple, en la que una parte se puede compartir por varios agregados. Por ejemplo, en el modelo de una casa, una Pared puede ser parte de uno o más objetos Habitacion.

Además, en una agregación compuesta, la parte compuesta es responsable de disponer de las partes, lo que significa que debe gestionar la creación y destrucción de éstas. Por ejemplo, al crear un Marco en un sistema de ventanas, debe asignarse a una Ventana contenedora. Análogamente, al destruir la Ventana, Ventana debe a su vez destruir sus partes Marco.

Como se muestra en la Figura 10.6, la composición es realmente un tipo especial de asociación, y se especifica adornando una asociación simple con un rombo relleno en el extremo del todo.

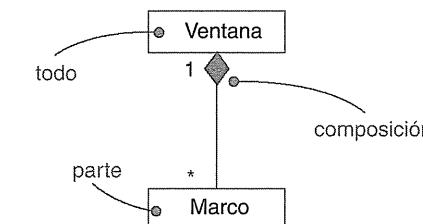


Figura 10.6: Composición.

La estructura interna se discute en el Capítulo 15.

Los atributos se discuten en los Capítulos 4 y 9.

Nota: De forma alternativa, se puede representar la composición anidando los símbolos de las partes dentro del símbolo del compuesto. Esta forma es más útil cuando se quieren destacar las relaciones entre las partes que se aplican sólo en el contexto del todo.

Clases asociación. En una asociación entre dos clases, la propia asociación puede tener propiedades. Por ejemplo, en una relación empleado/patrón entre una Compañía y una Persona, hay un Trabajo que representa las propiedades de esa relación y que se aplican exactamente a un par de Persona y Compañía. No sería apropiado modelar esta situación con una asociación de Compañía a Trabajo junto con una asociación de Trabajo a Persona. Eso no ligaría una instancia específica de Trabajo al par específico de Compañía y Persona.

En UML, esto se modelaría con una clase asociación, la cual es un elemento de modelado con propiedades tanto de asociación como de clase. Una clase asociación puede verse como una asociación que también tiene propiedades de clase, o una clase que también tiene propiedades de asociación. Una clase asociación se dibuja con un símbolo de clase unido por una línea discontinua a una asociación, como se muestra en la Figura 10.8.

Nota: Algunas veces se desea tener las mismas propiedades en varias clases asociación distintas. Sin embargo, una clase asociación no se puede conectar a más de una asociación, ya que una clase asociación es la propia asociación. Para conseguir este propósito, se puede definir una clase (C) y luego hacer que cada clase asociación que necesite esas características herede de C o utilice a C como el tipo de un atributo.

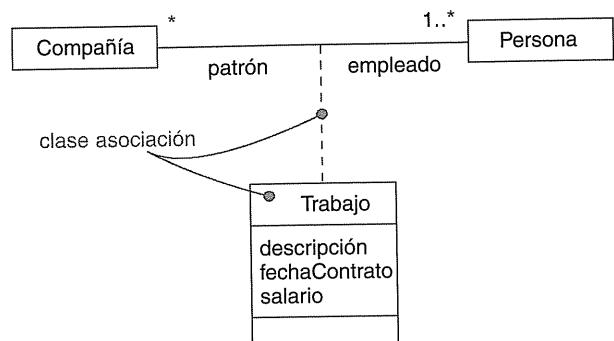


Figura 10.7: Clases asociación.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Restricciones. Las propiedades simples y avanzadas de las asociaciones que han sido presentadas hasta ahora son suficientes para la mayoría de las relaciones estructurales que aparecen al modelar. Sin embargo, si se quiere especificar ciertos matices, UML define cinco restricciones que pueden aplicarse a las asociaciones.

En primer lugar, se puede especificar si los objetos de un extremo de la asociación (con una multiplicidad mayor que uno) están ordenados o no.

1. ordered Especifica que el conjunto de objetos en un extremo de una asociación sigue un orden explícito.

Por ejemplo, en una asociación Usuario/Clave, las Claves asociadas con el Usuario podrían mantenerse en orden de menos a más recientemente usada, y se marcarían como ordered. Si no existe la palabra clave, los objetos no están ordenados.

Estas propiedades de mutabilidad también se aplican a los atributos, como se discute en el Capítulo 9; los enlaces se discuten en el Capítulo 16.

En segundo lugar, se puede especificar que los objetos en un extremo de la asociación son únicos (es decir, forman un conjunto) o no lo son (es decir, forman una bolsa).

2. set Objetos únicos, sin duplicados.
 3. bag Objetos no únicos; puede haber duplicados.
 4. ordered set Objetos únicos, pero ordenados.
 5. list o sequence Objetos ordenados; puede haber duplicados.

Por último, hay una restricción que restringe los cambios posibles de las instancias de una asociación.

6. readonly

Un enlace, una vez añadido desde un objeto del otro extremo de la asociación, no se puede modificar ni eliminar. El valor por defecto, en ausencia de esta restricción, es permitir los cambios.

Nota: Para ser precisos, ordered y readonly son propiedades de un extremo de una asociación. No obstante, se representan con la notación de las restricciones.

Realización

Una *realización* es una relación semántica entre clasificadores, en la cual un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Gráficamente, una realización se representa como una línea dirigida discontinua, con una gran flecha hueca que apunta al clasificador que especifica el contrato.

La realización es lo suficientemente diferente de la dependencia, la generalización y la asociación para ser tratada como un tipo diferente de relación. Semánticamente, la realización es algo así como una mezcla entre dependencia y generalización, y su notación es una combinación de la notación para la dependencia y la generalización. La realización se utiliza en dos circunstancias: en el contexto de las interfaces y en el contexto de las colaboraciones.

Las interfaces se discuten en el Capítulo 11; las clases se discuten en los Capítulos 4 y 9; los componentes se discuten en el Capítulo 15; las cinco vistas de una arquitectura se discuten en el Capítulo 2.

La mayoría de las veces la realización se empleará para especificar la relación entre una interfaz y la clase o el componente que proporciona una operación o servicio para ella. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o componente. Por consiguiente, una interfaz especifica un contrato que debe llevar a cabo una clase o un componente. Una interfaz puede ser realizada por muchas clases o componentes, y una clase o un componente pueden realizar muchas interfaces. Quizás lo más interesante de las interfaces sea que permiten separar la especificación de un contrato (la propia interfaz) de su implementación (por una clase o un componente). Además, las interfaces incluyen las partes lógicas y físicas de la arquitectura de un sistema. Por ejemplo, como se muestra en la Figura 10.8, una clase (como ReglasNegocioCuenta en un sistema de entrada de pedidos) en una vista de diseño de un sistema podría realizar una interfaz dada (como IAgenteDeReglas). Esta misma interfaz (IAgenteDeReglas) también podría ser reali-

zada por un componente (como `reglascuenta.dll`) en la vista de implementación del sistema. Como muestra la figura, una realización se puede representar de dos maneras: de la forma canónica (con el estereotipo `interface` y la línea dirigida discontinua con la flecha hueca) y de la forma abreviada (con la notación en forma de piruleta).

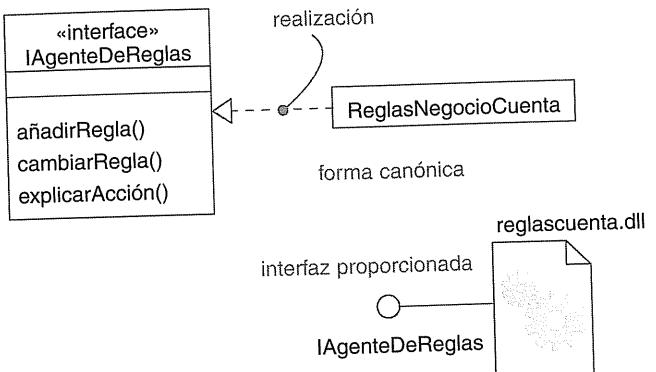


Figura 10.8: Realización de una interfaz.

Los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Como se muestra en la Figura 10.9, también se utiliza la realización para especificar la relación entre un caso de uso y la colaboración que realiza ese caso de uso. En esta circunstancia, casi siempre se utiliza la forma canónica de la realización.

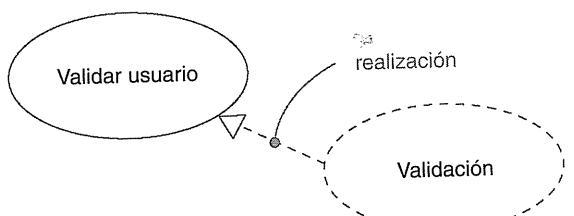


Figura 10.9: Realización de un caso de uso.

Nota: El que una clase o un componente realice una interfaz significa que los clientes pueden confiar en que la clase o el componente llevan a cabo fielmente el comportamiento especificado en la interfaz. Esto quiere decir que la clase o el componente implementan todas las operaciones de la interfaz, responden a todas sus señales, y siempre siguen el protocolo establecido por la interfaz para los clientes que utilizan esas operaciones o envían esas señales.

Técnicas comunes de modelado

Modelado de redes de relaciones

El modelado del vocabulario de un sistema y el modelado de la distribución de responsabilidades en un sistema se discuten en el Capítulo 4.

Los casos de uso se discuten en el Capítulo 17.

Al modelar el vocabulario de un sistema aparecen docenas, si no cientos o miles, de clases, interfaces, componentes, nodos y casos de uso. Es difícil establecer una frontera bien definida alrededor de cada una de estas abstracciones. Establecer la miríada de relaciones entre estas abstracciones es aún más difícil: requiere hacer una distribución equilibrada de responsabilidades en el sistema global, con abstracciones individuales muy cohesivas y relaciones expresivas, pero todo débilmente acoplado.

Cuando se modelen estas redes de relaciones:

- No hay que comenzar de forma aislada. Deben aplicarse los casos de uso y los escenarios para guiar el descubrimiento de las relaciones entre un conjunto de abstracciones.
- En general, hay que comenzar modelando las relaciones estructurales que estén presentes. Éstas reflejan la vista estática del sistema y por ello son bastante tangibles.
- A continuación hay que identificar las posibles relaciones de generalización/especialización; debe usarse la herencia múltiple de forma moderada.
- Sólo después de completar los pasos precedentes se deberán buscar dependencias; normalmente representan formas más sutiles de conexión semántica.
- Para cada tipo de relación hay que comenzar con su forma básica y aplicar las características avanzadas sólo cuando sean absolutamente necesarias para expresar la intención.
- Hay que recordar que no es deseable ni necesario modelar en un único diagrama o vista todas las relaciones entre un conjunto de abstracciones. En vez de ello, debe extenderse gradualmente el conjunto de relaciones del sistema, considerando diferentes vistas de éste. Hay que resaltar los conjuntos interesantes de relaciones en diagramas individuales.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2; el Proceso Unificado de Rational se resume en el Apéndice B.

La clave para tener éxito al modelar redes complejas de relaciones es hacerlo de forma incremental. Hay que incrementar gradualmente las relaciones conforme se añaden a la estructura de la arquitectura de un sistema. Hay que simplificar

esas relaciones conforme se descubren ocasiones de aplicar mecanismos comunes. En cada versión del proceso de desarrollo, se deben revisar las relaciones entre las abstracciones clave del sistema.

Nota: En la práctica (y especialmente si se sigue un proceso de desarrollo incremental e iterativo), las relaciones de los modelos derivarán de decisiones explícitas del modelador, así como de la ingeniería inversa de la implementación.

Sugerencias y consejos

Al modelar relaciones avanzadas en UML, conviene recordar que hay disponible un amplio rango de bloques de construcción, desde asociaciones simples hasta propiedades más detalladas de navegación, calificación, agregación, etc. Se debe elegir la relación y los detalles de la relación que mejor encajen con una abstracción dada. Una relación bien estructurada:

- Sólo muestra las características necesarias para que los clientes usen la relación y oculta las demás.
- No es ambigua en su objetivo ni en su semántica.
- No está tan sobreespecificada que elimine cualquier grado de libertad de los implementadores.
- No está tan poco especificada que quede ambiguo el significado de la relación.

Cuando se dibuje una relación en UML:

- Hay que mostrar sólo aquellas propiedades de la relación que sean importantes para comprender la abstracción en su contexto.
- Hay que elegir una versión estereotipada que proporcione la mejor señal visual del propósito de la relación.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 11

INTERFACES, TIPOS Y ROLES

En este capítulo

- Interfaces, tipos, roles y realización.
- Modelado de las líneas de separación en un sistema.
- Modelado de tipos estáticos y dinámicos.
- Obtener interfaces comprensibles y accesibles.

Las interfaces definen una línea entre la especificación de lo que una abstracción hace y la implementación de cómo lo hace. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o de un componente.

Las interfaces se utilizan para visualizar, especificar, construir y documentar las líneas de separación dentro de un sistema. Los tipos y los roles proporcionan mecanismos para modelar la conformidad estática y dinámica de una abstracción con una interfaz en un contexto específico.

Una interfaz bien estructurada proporciona una clara separación entre las vistas externa e interna de una abstracción, haciendo posible comprender y abordar una abstracción sin tener que sumergirse en los detalles de su implementación.

Introducción

El diseño de casas se discute en el Capítulo 1.

No tendría sentido construir una casa donde hubiese que romper los cimientos cada vez que hubiese que pintar las paredes. Asimismo, nadie desearía vivir en un sitio donde hubiese que reinstalar los cables cada vez que se cambiara una lámpara. Al propietario de un gran edificio no le haría mucha gracia tener que

mover puertas o reemplazar las cajas eléctricas o del teléfono cada vez que se instalase un nuevo inquilino.

Siglos de experiencia en la construcción han proporcionado una gran cantidad de información práctica para ayudar a los constructores a evitar estos problemas obvios (y a veces no tan obvios) que aparecen cuando un edificio crece o cambia con el tiempo. En terminología del software, llamamos a esto diseñar con una clara separación de intereses. Por ejemplo, en un edificio bien estructurado, la fachada de la estructura puede modificarse o cambiarse sin afectar al resto del edificio. De la misma manera, los muebles de un edificio pueden cambiarse de sitio sin cambiar la infraestructura. Los servicios instalados por dentro de las paredes, como electricidad, calefacción, fontanería y eliminación de residuos pueden cambiarse con un poco de escombro y trabajo, pero aun y así no hay que destruir la estructura básica del edificio para hacer eso.

Las prácticas estándar de construcción no sólo ayudan a construir edificios que pueden evolucionar a lo largo del tiempo, sino que hay muchas interfaces estándar con las que se puede construir, y que permiten usar componentes conocidos disponibles en el mercado, cuyo uso ayuda a reducir en última instancia el coste de la construcción y el mantenimiento. Por ejemplo, hay tamaños estándar para bloques de madera, que hacen fácil construir paredes que sean múltiples de un tamaño común. Hay tamaños estándar de puertas y ventanas, lo que significa que no hay que construir artesanalmente cada abertura en el edificio. Incluso hay estándares para los enchufes eléctricos y de teléfono (aunque varían de un país a otro) que hacen más fácil combinar y acoplar equipos electrónicos.

Los frameworks se discuten en el Capítulo 29.

En el software es importante construir sistemas con una clara separación de intereses, de forma que, al evolucionar el sistema, los cambios en una parte del sistema no se propaguen, afectando a otras partes del sistema. Una forma importante de lograr este grado de separación es especificar unas líneas de separación claras en el sistema, estableciendo una frontera entre aquellas partes que pueden cambiar independientemente. Además, al elegir las interfaces apropiadas, se pueden tomar componentes estándar, bibliotecas y *frameworks* para implementar esas interfaces, sin tener que construirlas uno mismo. Al ir descubriendo mejores implementaciones, se pueden reemplazar las viejas sin afectar a sus usuarios.

Las clases se discuten en los Capítulos 4 y 9; los componentes se discuten en el Capítulo 15.

En UML, las interfaces se emplean para modelar las líneas de separación de un sistema. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o un componente. Al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de una implementación de ella. Los clientes pueden trabajar con esa interfaz, y

se puede construir o comprar cualquier implementación de ella, siempre que esta implementación satisfaga las responsabilidades y el contrato indicado en la interfaz.

Los paquetes se discuten en el Capítulo 12; los subsistemas se discuten en el Capítulo 32.

Los componentes se discuten en el Capítulo 15.

Muchos lenguajes de programación soportan el concepto de interfaces, incluyendo Java y el IDL de CORBA. Las interfaces no sólo son importantes para separar la especificación y la implementación de una clase o un componente, sino que, al pasar a sistemas más grandes, se pueden usar las interfaces para especificar la vista externa de un paquete o subsistema.

UML proporciona una representación gráfica de las interfaces, como se muestra en la Figura 11.1. Esta notación permite ver la especificación de una abstracción separada de cualquier implementación.

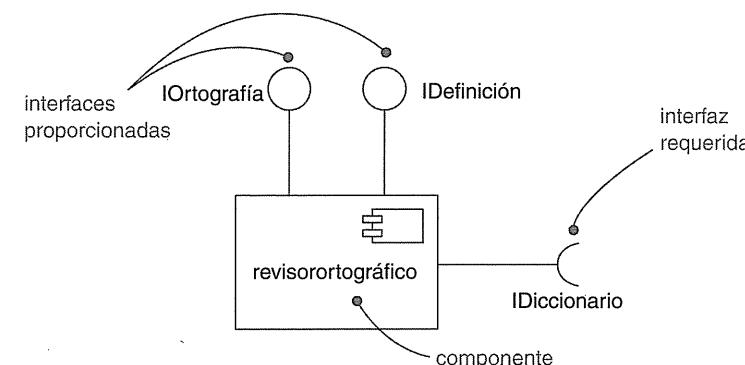


Figura 11.1: Interfaces.

Términos y conceptos

Una *interfaz* es una colección de operaciones que se usa para especificar un servicio de una clase o de un componente. Un *tipo* es un estereotipo de una clase utilizado para especificar un dominio de objetos, junto con las operaciones (pero no los métodos) aplicables al objeto. Un *rol* es el comportamiento de una entidad participante en un contexto particular.

Gráficamente, una interfaz se representa como una clase estereotipada para mostrar sus operaciones y otras propiedades. Para mostrar la relación entre una clase y sus interfaces existe una notación especial. Una interfaz proporcionada (la que representa servicios prestados por la clase) se dibuja como un pequeño círculo

unido a la clase. Una interfaz requerida (la que representa servicios que una clase espera de otra) se dibuja como un pequeño semicírculo unido a la clase.

Nota: Las interfaces también se pueden usar para especificar un contrato para un caso de uso o un subsistema.

Nombres

El nombre de una interfaz debe ser único dentro del paquete que la contiene, como se discute en el Capítulo 12.

Cada interfaz ha de tener un nombre que la distinga de otras interfaces. Un *nombre* es una cadena de texto. Ese nombre solo se denomina *nombre simple*; un *nombre calificado* consta del nombre de la interfaz precedido por el nombre del paquete en el que se encuentra. Una interfaz puede dibujarse mostrando sólo su nombre, como se ilustra en la Figura 11.2.

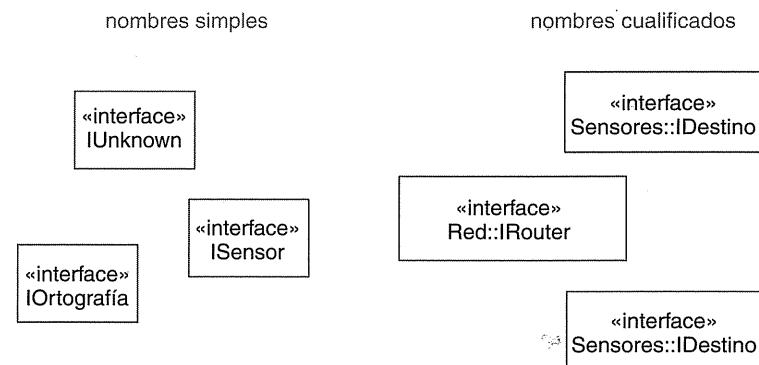


Figura 11.2: Nombres simples y nombres calificados.

Nota: El nombre de una interfaz puede ser texto con cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se emplean para separar el nombre de la interfaz y el del paquete que la contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de las interfaces son sustantivos cortos o expresiones nominales extraídas del vocabulario del sistema que se está modelando.

Operaciones

Una interfaz es una colección de operaciones con un nombre, que se usa para especificar un servicio de una clase o de un componente. Al contrario que las

Las operaciones se discuten en los Capítulos 4 y 9; los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

clases y los tipos, las interfaces no especifican ninguna implementación (así que no pueden incluir métodos, que proporcionan la implementación de una operación). Como una clase, una interfaz puede incluir cualquier número de operaciones. Éstas pueden adornarse con propiedades de visibilidad y concurrencia, estereotipos, valores etiquetados y restricciones.

Cuando se declara una interfaz, se representa como una clase estereotipada, listando sus operaciones en el compartimento apropiado. Las operaciones se pueden representar sólo con su nombre o pueden extenderse para mostrar la firma completa y otras propiedades, como se muestra en la Figura 11.3.

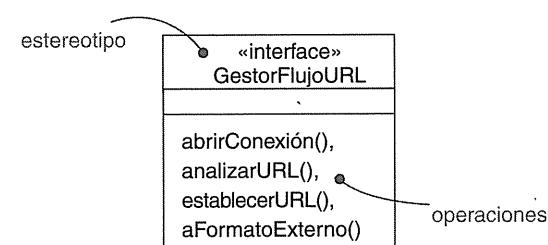


Figura 11.3: Operaciones.

Los eventos se discuten en el Capítulo 21.

Las relaciones se discuten en los Capítulos 5 y 10.

Nota: También se pueden asociar señales a una interfaz.

Relaciones

Al igual que una clase, una interfaz puede participar en relaciones de generalización, asociación y dependencia. Además, una interfaz puede participar en relaciones de realización. La realización es una relación semántica entre dos clasificadores en la que un clasificador especifica un contrato que el otro clasificador garantiza llevar a cabo.

Una interfaz especifica un contrato para una clase o un componente sin dictar su implementación. Una clase o un componente puede realizar muchas interfaces. Al hacer esto, se compromete a cumplir todos esos contratos fielmente, lo que significa que proporciona un conjunto de métodos que implementan apropiadamente las operaciones definidas en la interfaz. El conjunto de servicios que se compromete a cumplir se denomina *interfaz proporcionada*. Análogamente, una clase o un componente pueden depender de varias interfaces. Al hacer esto, esperan que estos contratos sean cumplidos por algún conjunto de componentes que los realizan. El conjunto de servicios que una clase espera de otra se denomina *interfaz requerida*. Por eso decimos que una interfaz representa una

línea de separación en un sistema. Una interfaz especifica un contrato, y el cliente y el proveedor de cada lado pueden cambiar independientemente, siempre que cada uno cumpla sus obligaciones en el contrato.

Como se muestra en la Figura 11.4, es posible indicar que un elemento realiza una interfaz de dos formas. La primera es la forma sencilla en la que la interfaz y su realización se representan como una piruleta conectada a un lado de una clase o de un componente (en el caso de una interfaz proporcionada). Esta forma es útil cuando simplemente se desea mostrar las líneas de separación del sistema; normalmente es la forma preferida. Sin embargo, la limitación de este estilo es que no se pueden visualizar directamente las operaciones o las señales proporcionadas por la interfaz. La otra forma es la forma expandida, en la cual se representa una interfaz como una clase estereotipada que permite ver las operaciones y otras propiedades, y dibujar una relación de realización (en el caso de una interfaz proporcionada) o una dependencia (en el caso de una interfaz requerida) del clasificador o del componente a la interfaz. En UML, una realización se representa como una línea discontinua con una flecha vacía apuntando a la interfaz. Esta notación es una mezcla entre la generalización y la dependencia.

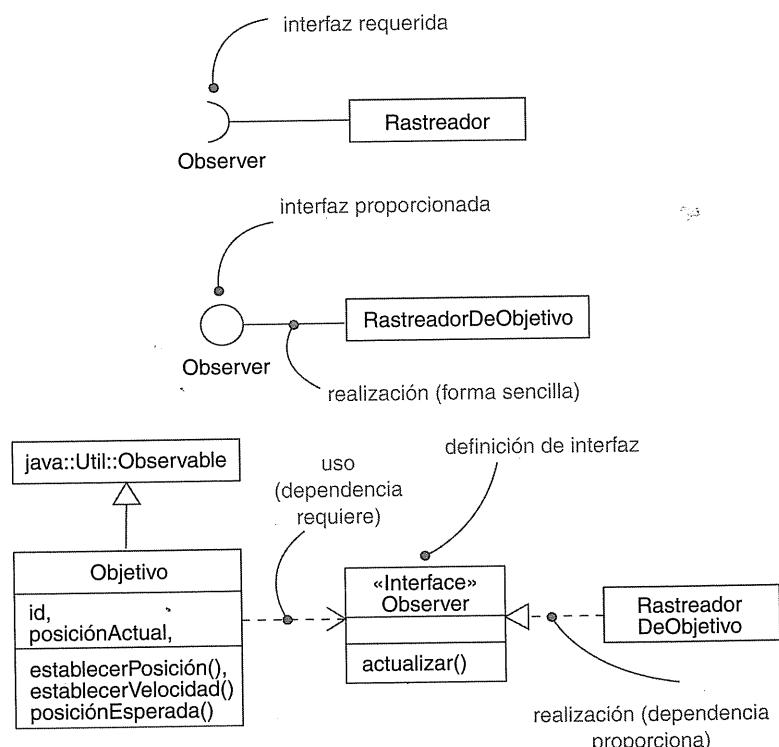


Figura 11.4: Realizaciones.

Las clases abstractas se discuten en el Capítulo 4; los componentes se discuten en el Capítulo 15.

Nota: Las interfaces son similares a las clases abstractas. Por ejemplo, ninguna de las dos puede tener instancias directas. Sin embargo, una clase abstracta puede implementar sus operaciones concretas. Una interfaz es más parecida a una clase abstracta en la que todas las operaciones también son abstractas.

Comprender una interfaz

Las operaciones y sus propiedades se discuten en el Capítulo 9; la semántica de concurrencia se discute en el Capítulo 24.

Cuando se comienza a manejar una interfaz, lo primero que se ve es un conjunto de operaciones que especifican un servicio de una clase o de un componente. Si se profundiza un poco más, se ve la firma completa de las operaciones, junto a otras propiedades especiales, como la visibilidad, el alcance y la semántica de concurrencia.

Estas propiedades son importantes, pero para las interfaces complejas no son suficientes para ayudar a comprender la semántica del servicio que representan; mucho menos para saber cómo usar esas operaciones correctamente. En ausencia de otra información, hay que sumergirse en alguna abstracción que realice a la interfaz para comprender qué hace cada operación y cómo se ha previsto que estas operaciones trabajen juntas. Sin embargo, esto contrasta con el propósito de una interfaz, que es proporcionar una clara separación de intereses en un sistema.

Las pre y postcondiciones y los invariantes se discuten en el Capítulo 9; las máquinas de estados se discuten en el Capítulo 22; las colaboraciones se discuten en el Capítulo 28; OCL se discute en el Capítulo 6.

En UML se puede proporcionar mucha más información a una interfaz, para hacerla comprensible y manejable. En primer lugar, se pueden asociar pre y postcondiciones a cada operación e invariantes a la clase o componente. Al hacer esto, un cliente que necesite usar una interfaz será capaz de entender qué hace y cómo utilizarla, sin tener que indagar en una implementación. Si hace falta ser riguroso, se puede emplear el OCL de UML para especificar formalmente la semántica. En segundo lugar, es posible asociar una máquina de estados a una interfaz. Esta máquina de estados se puede usar para especificar el orden parcial legal de las operaciones de la interfaz. En tercer lugar, es posible asociar colaboraciones a la interfaz. Las colaboraciones se pueden emplear para especificar el comportamiento esperado de la interfaz a través de una serie de diagramas de interacción.

Técnicas comunes de modelado

Modelado del vocabulario de un sistema

La mayoría de las veces, las interfaces se emplean para modelar las líneas de separación de un sistema compuesto de componentes software, tales

Los componentes se discuten en el Capítulo 15; los sistemas se discuten en el Capítulo 32.

como Eclipse, .NET o Java Beans. Algunos componentes de otros sistemas pueden ser reutilizados o comprados a terceros; otros se crearán a partir de cero. En cualquier caso, se necesitará escribir algún código destinado a combinar los componentes. Esto exige comprender las interfaces proporcionadas y aquellas en las que se basa cada componente.

Identificar las líneas de separación en un sistema implica identificar líneas claras de demarcación en la arquitectura. A ambos lados de esas líneas habrá componentes que pueden cambiar independientemente, sin afectar a los componentes del otro lado, mientras los componentes de ambos lados conformen con el contrato especificado por la interfaz.

Los patrones y los frameworks se discuten en el Capítulo 29.

Cuando se reutiliza un componente de otro sistema o cuando se adquiere a terceros, probablemente lo único que se tiene es un conjunto de operaciones con alguna documentación mínima sobre el significado de cada una. Esto es útil, pero no suficiente. Es más importante comprender el orden en el que se debe llamar a cada operación, y qué mecanismos subyacentes contiene la interfaz. Desafortunadamente, dado un componente con una documentación muy pobre, lo mejor que uno puede hacer es construir, mediante ensayo y error, un modelo conceptual de cómo trabaja esa interfaz. Entonces se puede documentar este conocimiento modelando con interfaces de UML esa línea de separación del sistema, de forma que, más tarde, unos y otros puedan manejar ese componente más fácilmente. Análogamente, cuando uno crea un componente propio, se necesita comprender su contexto, es decir, especificar las interfaces en las que se basa, así como las interfaces que presenta al mundo, y en las que otros se pueden basar.

Nota: La mayoría de los sistemas de componentes, como Eclipse y Enterprise Java Beans, proporcionan introspección de componentes, lo que quiere decir que un programa puede consultar a una interfaz para determinar sus operaciones. Éste es el primer paso para entender la naturaleza de cualquier componente poco documentado.

Para modelar el vocabulario de un sistema:

- Dentro de la colección de clases y componentes del sistema, hay que dibujar una línea alrededor de aquellos que tienden a acoplarse estrechamente con otros conjuntos de clases y componentes.
- Hay que refinar la agrupación realizada considerando el impacto del cambio. Las clases o componentes que tienden a cambiar juntos deberían agruparse juntos como colaboraciones.

Las colaboraciones se discuten en el Capítulo 28.

- Hay que considerar las operaciones y las señales que cruzan estos límites, desde instancias de un conjunto de clases o componentes hacia instancias de otros conjuntos de clases o componentes.
- Hay que empaquetar los conjuntos relacionados lógicamente de estas operaciones y señales como interfaces.
- Para cada colaboración del sistema, hay que identificar las interfaces que requiere (importa) y las que suministra a otros (exporta). La importación de interfaces se modela con relaciones de dependencia y la exportación de interfaces con relaciones de realización.
- Para cada interfaz del sistema, hay que documentar su dinámica mediante pre y postcondiciones para cada operación, y casos de uso y máquinas de estados para la interfaz como un todo.

Por ejemplo, la Figura 11.5 representa las líneas de separación alrededor de un componente LibroMayor, extraído de un sistema financiero. Este componente proporciona (realiza) tres interfaces: IUnknown, ILibroMayor e ITransacción. En este diagrama, IUnknown se muestra expandida; las otras dos se muestran en su forma sencilla, como piruletas. Estas tres interfaces son realizadas por LibroMayor y se exportan a otros componentes para que puedan basarse en ellas.

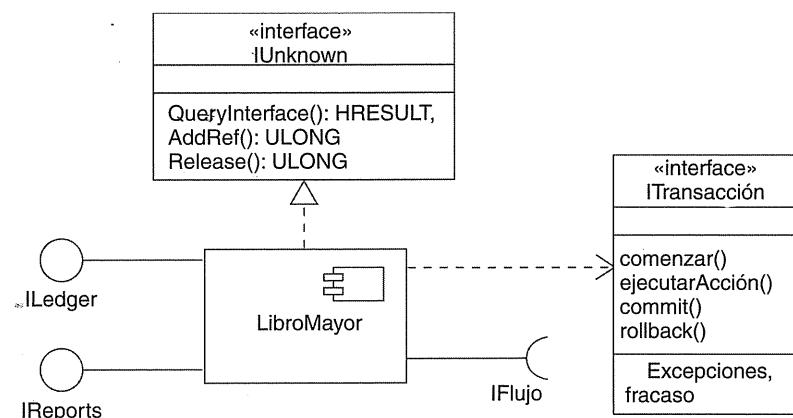


Figura 11.5: Modelado de las líneas de separación en un sistema.

Como también se ve en este diagrama, LibroMayor requiere (usa) dos interfaces, IFlujo e ITransaccion, la última de las cuales se presenta expandida. El componente LibroMayor requiere estas dos interfaces para un funcionamiento correcto. Por tanto, en un sistema en ejecución, hay que proveer compo-

nentes que realicen ambas interfaces. Al identificar interfaces como `ITransaccion`, se han desacoplado efectivamente los componentes a ambos lados de la interfaz, lo que permite emplear cualquier componente conforme con ella.

Los casos de uso se discuten en el Capítulo 17.

Las interfaces como `ITransaccion` son algo más que una simple colección de operaciones. Esta interfaz particular hace algunas suposiciones sobre el orden en que se deberían llamar estas operaciones. Aunque no se muestra aquí, se podrían asociar casos de uso a esta interfaz y enumerar las formas más frecuentes de usarla.

Modelado de tipos estáticos y dinámicos

Las instancias se discuten en el Capítulo 13.

La mayoría de los lenguajes de programación orientados a objetos están tipados estáticamente, lo que significa que el tipo de un objeto se establece en el momento en que se crea el objeto. Incluso así, ese objeto probablemente jugará diferentes roles a lo largo del tiempo. Esto significa que los clientes que utilicen a ese objeto interactuarán con él a través de diferentes conjuntos de interfaces, que representan conjuntos de operaciones interesantes, y posiblemente solapados.

Los diagramas de clases se discuten en el Capítulo 8.

El modelado de la naturaleza estática de un objeto puede visualizarse en un diagrama de clases. Sin embargo, cuando se modelan cosas como objetos del negocio, que naturalmente cambian sus roles a través de un flujo de trabajo, a veces es útil modelar explícitamente la naturaleza dinámica del tipo de ese objeto. En esas circunstancias, un objeto puede ganar y perder tipos a lo largo de su vida. También se puede modelar el ciclo de vida del objeto utilizando una máquina de estados.

Para modelar un tipo dinámico:

Las asociaciones y las generalizaciones se discuten en los Capítulos 5 y 10; los diagramas de interacción se discuten en el Capítulo 19; las dependencias se discuten en los Capítulos 5 y 10.

- Hay que especificar los diferentes tipos posibles del objeto, representando cada tipo como una clase (si la abstracción requiere estructura y comportamiento) o como una interfaz (si la abstracción sólo requiere comportamiento).
- Hay que modelar todos los roles que puede asumir la clase del objeto en cualquier momento. Éstos se pueden estereotipar como `<<dynamic>>`. (Este estereotipo no está predefinido en UML, pero se puede añadir).
- En un diagrama de interacción, hay que representar apropiadamente cada instancia de la clase tipada dinámicamente. Debe mostrarse el

tipo de la instancia entre corchetes debajo del nombre del objeto, igual que si fuera un estado. (Estamos utilizando la sintaxis de UML de una forma novedosa, pero que creemos consistente con la intención de los estados).

Por ejemplo, la Figura 11.6 muestra los roles que pueden jugar las instancias de la clase `Persona` en el contexto de un sistema de recursos humanos.

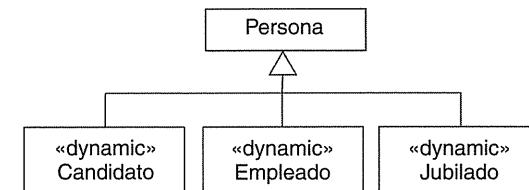


Figura 11.6: Modelado de tipos estáticos.

Este diagrama especifica que las instancias de la clase `Persona` pueden ser de cualquiera de los tres tipos (a saber, `Candidato`, `Empleado` o `Jubilado`).

Sugerencias y consejos

Al modelar una interfaz en UML, hay que recordar que cada interfaz debe representar una línea de separación en el sistema, separando especificación de implementación. Una interfaz bien estructurada:

- Es sencilla, aunque completa, y proporciona todas las operaciones necesarias y suficientes para especificar un único servicio.
- Es comprensible, y proporciona suficiente información tanto para permitir su uso como para su realización sin tener que examinar algún uso ya existente o alguna implementación.
- Es manejable, y proporciona información para guiar al usuario hacia sus propiedades claves sin estar sobrecargado por los detalles de un gran número de operaciones.

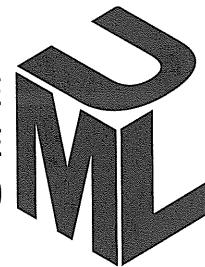
Cuando se dibuje una interfaz en UML:

- Hay que emplear la notación con forma de piruleta o enchufe cuando sólo sea necesario especificar la presencia de una línea de separación en

el sistema. La mayoría de las veces esto es lo que se necesitará para los componentes, no para las clases.

- Hay que emplear la forma expandida cuando sea necesario visualizar los detalles del propio servicio. La mayoría de las veces esto es lo que se necesita para especificar las líneas de separación en un sistema asociado a un paquete o a un subsistema.

LENGUAJE
UNIFICADO DE
MODELADO



Capítulo 12 PAQUETES

En este capítulo

- Paquetes, visibilidad, importación y exportación.
- Modelado de grupos de elementos.
- Modelado de vistas arquitectónicas.
- Transición a grandes sistemas.

Visualizar, especificar, construir y documentar grandes sistemas conlleva manejar una cantidad de clases, interfaces, componentes, nodos, diagramas y otros elementos que puede ser muy elevada. Conforme va creciendo el sistema hasta alcanzar un gran tamaño, se hace necesario organizar estos elementos en bloques mayores. En UML el paquete es un mecanismo de propósito general para organizar elementos de modelado en grupos.

Los paquetes se utilizan para organizar los elementos de modelado en partes mayores que se pueden manipular como un grupo. La visibilidad de estos elementos puede controlarse para que algunos sean visibles fuera del paquete mientras que otros permanecen ocultos. Los paquetes también se pueden emplear para presentar diferentes vistas de la arquitectura del sistema.

Los paquetes bien diseñados agrupan elementos cercanos semánticamente y que suelen cambiar juntos. Por tanto, los paquetes bien estructurados son cohesivos y poco acoplados, y el acceso a su contenido está muy controlado.

Introducción

La diferencia entre construir una caseta para el perro y construir un rascacielos se discuten en el Capítulo 1.

Las casetas de perro no son complejas: hay cuatro paredes, una de ellas con un agujero del tamaño de un perro, y un techo. Al construir una caseta de perro sólo se necesitan unas cuantas tablas. No hay mucha más estructura.

Las casas son más complejas. Paredes, techos y suelos se combinan en abstracciones mayores que llamamos habitaciones. Incluso esas habitaciones se organizan en abstracciones mayores: la zona pública, la zona de dormir, la zona de trabajo, etc. Estos grupos mayores no tienen por qué manifestarse como algo que construir en la propia casa, sino que pueden ser simplemente los nombres que damos a habitaciones relacionadas lógicamente, y que se aplican cuando hablamos sobre el uso que se hará de la casa.

Los grandes edificios son muy complejos. No sólo existen estructuras elementales, como paredes, techos y suelos, sino que hay estructuras más complejas, tales como las zonas públicas, el área comercial y la zona de oficinas. Estas estructuras probablemente se agruparán en otras aún más complejas, como las zonas de alquileres y las zonas de servicios del edificio. Puede que estas estructuras más complejas no tengan nada que ver con el edificio final, sino que sean simplemente artefactos que se utilizan para organizar los planos del edificio.

Todos los sistemas grandes se jerarquizan en niveles de esta forma. De hecho, quizás la única forma de comprender un sistema complejo sea agrupando las abstracciones en grupos cada vez mayores. La mayoría de las agrupaciones básicas (como las habitaciones) son, por derecho propio, abstracciones de la misma naturaleza que las clases, para las que puede haber muchas instancias. La mayoría de las abstracciones mayores son puramente conceptuales (como el área comercial), para las que no existen instancias reales. No existen objetos distinguidos en el sistema, sino que más bien representan vistas del propio sistema. Este último tipo de agrupaciones no tiene una identidad individual en el sistema desplegado; representa agrupamientos de partes seleccionadas a través de todo el sistema.

En UML las abstracciones que organizan un modelo se denominan paquetes. Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los paquetes ayudan a organizar los elementos en los modelos con el fin de comprenderlos más fácilmente. Los paquetes también permiten controlar el acceso a sus contenidos para controlar las líneas de separación en la arquitectura del sistema.

UML proporciona una representación gráfica de los paquetes, como se muestra en la Figura 12.1. Esta notación permite visualizar grupos de elementos que se pueden manipular como un todo y de una forma que permite controlar la visibilidad y el acceso a elementos individuales.

La arquitectura del software se discute en el Capítulo 2; el modelado de la arquitectura de un sistema se discute en el Capítulo 32.

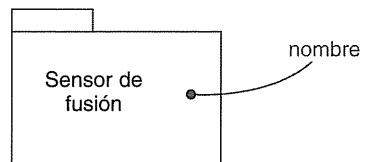


Figura 12.1: Paquetes.

Términos y conceptos

Un *paquete* es un mecanismo de propósito general para organizar el propio modelo de manera jerárquica. Gráficamente, un paquete se representa como una carpeta con una pestaña. El nombre del paquete va en la carpeta (si no se muestra su contenido) o en la pestaña (si se muestra el contenido).

Nombres

El nombre de un paquete debe ser único dentro del paquete que lo contiene.

Cada paquete ha de tener un nombre que lo distinga de otros paquetes. Un nombre es una cadena de texto. El *nombre* solo se denomina *nombre simple*; un *nombre calificado* consta del nombre del paquete precedido por el nombre del paquete en el que se encuentra, si es el caso. Para separar los nombres de los paquetes se emplea un signo de dos puntos duplicado (::). Un paquete se dibuja normalmente mostrando sólo su nombre, como se ve en la Figura 12.2. Al igual que con las clases, los paquetes se pueden dibujar adornados con valores etiquetados o con apartados adicionales para mostrar sus detalles.

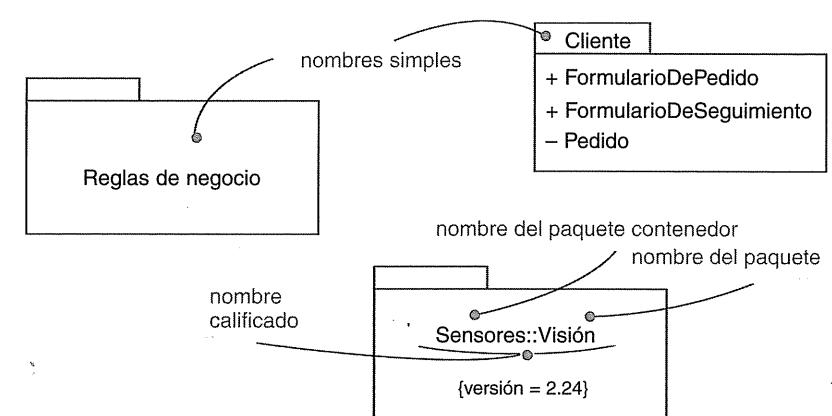


Figura 12.2: Nombres de paquetes simples y nombres calificados.

Nota: El nombre de un paquete puede ser texto con cualquier número de letras, dígitos y ciertos signos de puntuación (excepto signos como los dos puntos, utilizados para separar el nombre de un paquete del nombre de su paquete contenedor) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de los paquetes son nombres cortos o expresiones nominales extraídos del vocabulario del modelo.

La composición se discute en el Capítulo 10.

Elementos contenidos

Un paquete puede contener otros elementos, tales como clases, interfaces, componentes, nodos, colaboraciones, casos de uso, diagramas e incluso otros paquetes. La posesión es una relación compuesta, lo que significa que el elemento se declara en el paquete. Si el paquete se destruye, el elemento es destruido. Cada elemento pertenece exclusivamente a un único paquete.

Nota: El paquete contiene a los elementos de modelado declarados en él. Éstos pueden incluir elementos como clases, asociaciones, generalizaciones, dependencias y notas. El paquete no contiene a los elementos que son simplemente referenciados desde él.

Un paquete forma un espacio de nombres, lo que quiere decir que los elementos de la misma categoría deben tener nombres únicos en el contexto de su paquete contenedor. Por ejemplo, no se pueden tener dos clases llamadas Cola dentro del mismo paquete, pero se puede tener una clase Cola en el paquete P1 y otra clase (diferente) llamada Cola en el paquete P2. Las clases P1::Cola y P2::Cola son, de hecho, clases diferentes y se pueden distinguir por sus nombres calificados. Elementos de diferentes tipos pueden tener el mismo nombre.

Nota: Si es posible, es mejor evitar la duplicación de nombres en diferentes paquetes, para prevenir el peligro de confusión.

Elementos de diferentes tipos pueden tener el mismo nombre dentro de un paquete. Así, se puede disponer de una clase llamada Temporizador, así como de un componente llamado Temporizador dentro del mismo paquete. En la práctica, sin embargo, para evitar la confusión, es mejor asociar a cada elemento un nombre único para todas las categorías dentro de un paquete.

La importación se discute más adelante en este capítulo.

Los paquetes pueden contener a otros paquetes. Esto significa que es posible descomponer los modelos jerárquicamente. Por ejemplo, se puede tener una clase Camara en el paquete Vision que a su vez esté contenido en el paquete Sensores. El nombre completo de la clase es Sensores::Vision::Camara. En la práctica es mejor evitar paquetes muy anidados. Aproximadamente, dos o tres niveles de anidamiento es el límite manejable. En vez del anidamiento, se usará la importación para organizar los paquetes.

Esta semántica de posesión convierte a los paquetes en un mecanismo importante para enfrentarse al crecimiento. Sin los paquetes, se acabaría con grandes modelos planos en los que todos los elementos deberían tener nombres únicos (una situación inmanejable, especialmente cuando se compran clases y otros elementos desarrollados por varios equipos). Los paquetes ayudan a controlar los elementos que componen un sistema mientras evolucionan a diferentes velocidades a lo largo del tiempo.

Como se aprecia en la Figura 12.3, se puede mostrar explícitamente el contenido de un paquete, bien textualmente, bien gráficamente. Cuando se muestran los elementos que posee, el nombre del paquete se coloca en la pestaña de la carpeta. En la práctica, normalmente no se muestra el contenido del paquete de esta forma. En vez de ello, se emplean herramientas software para examinar en detalle los contenidos de un paquete.

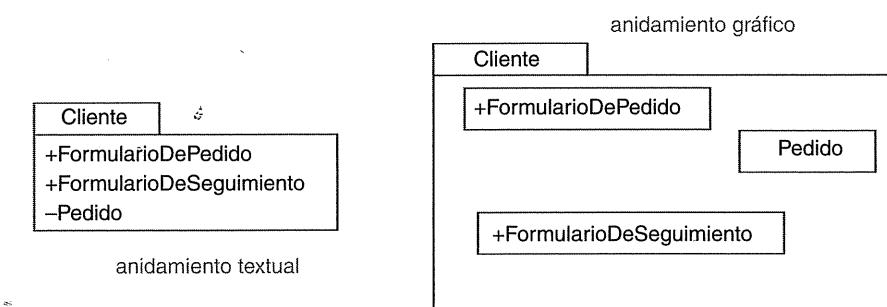


Figura 12.3: Elementos contenidos.

Nota: UML asume que existe un paquete raíz anónimo en un modelo, y como consecuencia de ello los elementos de cada tipo en nivel más alto de un modelo deben tener nombres únicos.

Visibilidad

La visibilidad se discute en el Capítulo 9.

Se puede controlar la visibilidad de los elementos contenidos en un paquete, del mismo modo que se puede controlar la visibilidad de los atributos y operaciones de una clase. Normalmente, un elemento contenido en un paquete es público, es decir, es visible a los contenidos de cualquier paquete que importe al paquete contenedor del elemento. Por el contrario, los elementos protegidos sólo pueden ser vistos por los hijos, y los elementos privados no son visibles fuera del paquete en el que se declaran. En la Figura 12.3, FormularioDePedido es una parte pública del paquete Cliente, y Pedido es una parte privada. Un paquete que importe a Cliente verá a FormularioDePedido, pero no verá a Pedido. Visto desde fuera, el nombre totalmente calificado de FormularioDePedido sería Cliente::FormularioDePedido.

Para especificar la visibilidad de un elemento contenido en un paquete se antepone al nombre del elemento el símbolo de visibilidad apropiado. Los elementos públicos se muestran con su nombre precedido del símbolo +, como pasa con FormularioDePedido en la Figura 12.3. El conjunto de las partes públicas de un paquete constituye la interfaz del paquete.

Al igual que con las clases, se puede designar a un elemento como protegido o privado, con el nombre del elemento precedido del símbolo # o del símbolo -, respectivamente. Los elementos protegidos sólo son visibles para los paquetes que heredan de otro paquete; los elementos privados no son visibles para nadie fuera del paquete.

La visibilidad de paquete indica que una clase es visible a otras clases declaradas en el mismo paquete, pero invisible a clases declaradas en otros paquetes. La visibilidad de paquete se muestra poniendo el símbolo ~ delante del nombre de la clase.

Importación y exportación

Supongamos que tenemos dos clases llamadas A y B, que se conocen mutuamente. Al ser compañeras, A puede ver a B y B puede ver a A, así que cada una depende de la otra. Dos clases tan sólo constituyen un sistema trivial, así que realmente no se necesita ningún tipo de empaquetamiento.

Ahora imaginemos que tenemos unos cientos de clases, que se conocen entre ellas. No hay límites para la intrincada red de relaciones que se puede establecer. Además, no hay forma de comprender un grupo de clases tan grande y desorga-

nizado. Éste es un problema real para grandes sistemas (el acceso simple y no restringido no permite el crecimiento). Para estas situaciones se necesita algún tipo de empaquetamiento controlado para organizar las abstracciones.

Las relaciones de dependencia se discuten en el Capítulo 5; los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Así que supongamos que en lugar de esto se coloca A en un paquete y B en otro, teniendo cada paquete conocimiento del otro. Supongamos también que A y B se declaran ambas como públicas en sus respectivos paquetes. Ésta es una situación muy diferente. Aunque A y B son ambas públicas, acceder a una de las clases desde el otro paquete requiere un nombre calificado. Sin embargo, si el paquete de A importa al paquete de B, A puede ahora ver a B, aunque B no puede ver a A sin un nombre calificado. La importación añade los elementos públicos del paquete destino al espacio de nombres público del paquete importador. En UML una relación de importación se modela como una dependencia con el estereotipo **import**. Al empaquetar las abstracciones en bloques significativos y luego controlar los accesos mediante la importación, se puede controlar la complejidad de tener un gran número de abstracciones.

Nota: Realmente, aquí se aplican dos estereotipos (**import** y **access**) y ambos especifican que el paquete origen tiene acceso al contenido del destino. **Import** añade el contenido del destino al espacio de nombres público del origen, de forma que no hay que calificar los nombres. Esto implica la posibilidad de colisión de nombres que hay que evitar para tener un modelo bien formado. **Access** añade el contenido del paquete destino al espacio de nombres privado del origen. La única diferencia es que no se pueden reexportar los elementos importados si un tercer paquete importa el paquete origen inicial. La mayoría de las veces se usará **import**.

Las partes públicas de un paquete son sus exportaciones. Por ejemplo, en la Figura 12.4, el paquete GUI exporta dos clases, Ventana y Formulario. GestorEventos no es exportada por GUI; GestorEventos es una parte protegida del paquete.

Las partes que exporta un paquete son sólo visibles al contenido de aquellos paquetes que lo importan explícitamente. En este ejemplo, Politicas importa explícitamente al paquete GUI. Las clases GUI::Ventana y GUI::Formulario son, por tanto, visibles para el contenido del paquete Politicas usando simplemente sus nombres simples Ventana y Formulario. Sin embargo, GUI::GestorEventos no es visible porque es protegido. Como el paquete Servidor no importa a GUI, el contenido de Servidor puede acceder a contenido público de GUI, pero debe

Las interfaces se discuten en el Capítulo 11.

usar los nombres calificados para hacer esto; por ejemplo, GUI : : Window. Análogamente, el contenido de GUI no tiene permiso para acceder al contenido de Servidor ya que éste es privado; es inaccesible utilizando incluso nombres calificados.

Las dependencias de importación y acceso son transitivas. En este ejemplo, Cliente importa Políticas y Políticas importa GUI, así que Cliente importa GUI de manera transitiva. Si Políticas accede a GUI, en vez de importarlo, Cliente no añade los elementos de GUI a su espacio de nombres, pero aún podría referenciarlos utilizando nombres calificados (como GUI : : Window).

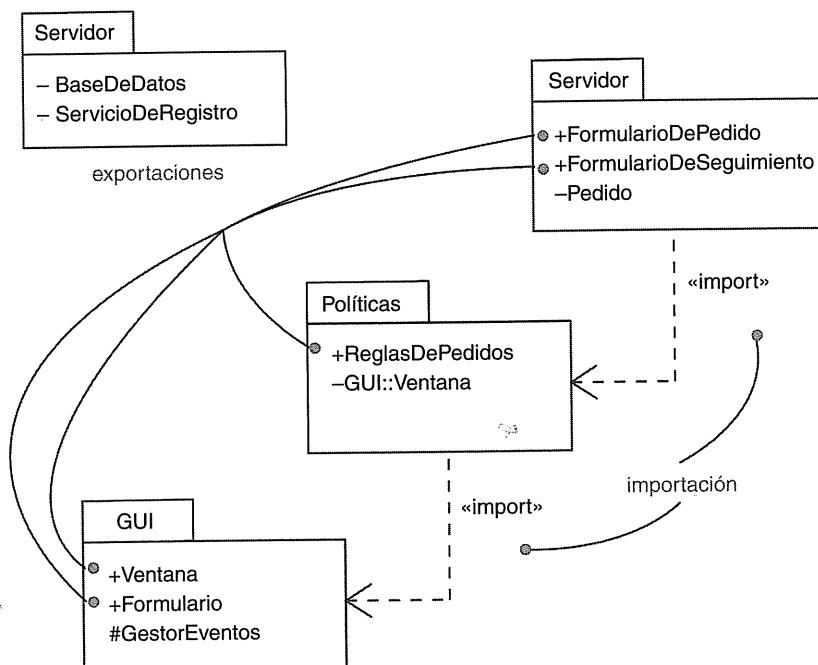


Figura 12.4: Importación y exportación.

Nota: Si un elemento es visible en un paquete, es visible en todos los paquetes incluidos en ese paquete. Los paquetes anidados pueden ver todo lo que los paquetes que los contienen pueden ver. Un nombre en un paquete anidado puede ocultar un nombre en un paquete contenedor, en cuyo caso se necesita un nombre calificado para referenciarlo.

Técnicas comunes de modelado

Modelado de grupos de elementos

El objetivo más frecuente para el que se utilizan los paquetes es organizar elementos de modelado en grupos a los que se puede dar un nombre y manejar como un conjunto. Si se está desarrollando una aplicación trivial, no harán falta paquetes. Todas las abstracciones encajarán perfectamente en un paquete. Sin embargo, para cualquier otro sistema, se detectará que muchas de las clases, interfaces, componentes y nodos tienden a agruparse de forma natural. Estos grupos se modelan como paquetes.

Hay una distinción importante entre clases y paquetes: las clases son abstracciones de cosas encontradas en el problema o en la solución; los paquetes son los mecanismos que se emplean para organizar los elementos del modelo. Los paquetes no aparecen en el sistema en ejecución; son estrictamente mecanismos para organizar el diseño.

La mayoría de las veces, los paquetes se utilizarán para agrupar el mismo tipo de elementos. Por ejemplo, todas las clases de la vista de diseño del sistema y sus correspondientes relaciones se podrían separar en una serie de paquetes, utilizando las dependencias de importación de UML para controlar el acceso entre esos paquetes. De la misma forma se podrían organizar todos los componentes de la vista de implementación del sistema.

Los paquetes también se pueden emplear para agrupar diferentes tipos de elementos. Por ejemplo, en un sistema en desarrollo por un equipo distribuido geográficamente, se podrían utilizar los paquetes como las unidades básicas para la gestión de configuraciones, colocando en ellos todas las clases y diagramas que cada equipo puede registrar y verificar independientemente. De hecho, es frecuente emplear los paquetes para agrupar elementos de modelado y sus diagramas asociados.

Para modelar grupos de elementos:

- Hay que examinar los elementos de modelado de una determinada vista arquitectónica en busca de grupos definidos por elementos cercanos entre sí desde un punto de vista conceptual o semántico.
- Hay que englobar cada uno de esos grupos en un paquete.
- Para cada paquete, hay que distinguir los elementos que podrán ser accedidos desde fuera. Deben marcarse estos elementos como públicos,

y los demás como protegidos o privados. En caso de duda, debe ocultarse el elemento.

- Hay que conectar explícitamente los paquetes que dependen de otros a través de dependencias de importación.
- En el caso de familias de paquetes, hay que conectar los paquetes especializados con sus partes más generales por medio de generalizaciones.

Por ejemplo, la Figura 12.5 representa un conjunto de paquetes que organiza las clases de la vista de diseño de un sistema de información en una arquitectura clásica de tres capas. Los elementos del paquete *Servicios de Usuario* proporcionan la interfaz visual para presentar información y capturar datos. Los elementos del paquete *Servicios de Datos* mantienen, acceden y actualizan los datos. Los elementos del paquete *Servicios de Negocio* enlazan los elementos de los otros dos paquetes e incluyen todas las clases y demás elementos que gestionan las solicitudes del usuario para ejecutar una tarea del negocio, incluyendo las reglas que dictan las políticas de gestión de los datos.

El valor etiquetado documentation se discute en el Capítulo 6.

En un sistema trivial se podrían agrupar todas las abstracciones en un paquete. Pero al organizar las clases y demás elementos de la vista de diseño en tres paquetes, no sólo se hace más comprensible el sistema, sino que se puede controlar el acceso a los elementos del modelo, ocultando unos y exportando otros.

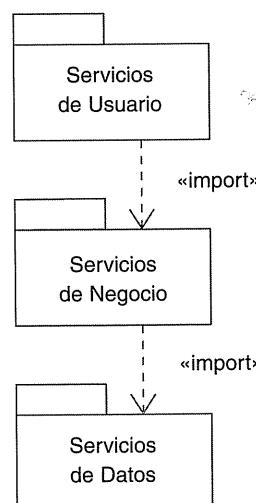


Figura 12.5: Modelado de grupos de elementos.

Nota: Al representar modelos como éstos, normalmente se mostrarán los elementos principales de cada paquete. Para clarificar el propósito de cada paquete también se puede mostrar un valor etiquetado documentation.

Modelado de vistas arquitectónicas

El valor etiquetado documentation se discute en el Capítulo 6.

Las vistas están relacionadas con los modelos, como se discute en el Capítulo 32.

El uso de paquetes para agrupar elementos relacionados es importante; no se pueden desarrollar modelos complejos sin utilizarlos. Este enfoque funciona bien para organizar elementos relacionados como clases, interfaces, componentes, nodos y diagramas. Cuando se consideran las diferentes vistas de la arquitectura de un sistema software, incluso se necesitan bloques mayores. Los paquetes se pueden emplear para modelar las vistas de una arquitectura.

Recuérdese que una vista es una proyección de la organización y estructura de un sistema, centrada en un aspecto particular del sistema. Esta definición tiene dos implicaciones. Primera, se puede descomponer un sistema en paquetes casi ortogonales, cada uno de los cuales cubre un conjunto de decisiones significativas arquitectónicamente. Por ejemplo, podría tenerse una vista de diseño, una vista de interacción, una vista de implementación, una vista de despliegue y una vista de casos de uso. Segunda, esos paquetes contienen todas las abstracciones pertinentes para esa vista. Por ejemplo, todos los componentes del modelo pertenecerían al paquete que representara la vista de implementación. No obstante, los paquetes pueden referenciar a elementos de otros paquetes.

Para modelar vistas arquitectónicas:

- Hay que identificar el conjunto de vistas arquitectónicas que son significativas en el contexto del problema. En la práctica, normalmente se incluyen una vista de diseño, una vista de interacción, una vista de implementación, una vista de despliegue y una vista de casos de uso.
- Hay que colocar en el paquete adecuado los elementos (y diagramas) necesarios y suficientes para visualizar, especificar, construir y documentar la semántica de cada vista.
- Si es necesario, hay que agrupar aún más estos elementos en sus propios paquetes.

Normalmente existirán dependencias entre los elementos de diferentes vistas. Así que, en general, hay que permitir a cada vista en la cima del sistema estar abierta al resto de las vistas en el mismo nivel.

El modelado de sistemas se discute en el Capítulo 32.

Por ejemplo, la Figura 12.6 ilustra una descomposición canónica de nivel superior apropiada incluso para los sistemas más complejos que puedan aparecer.

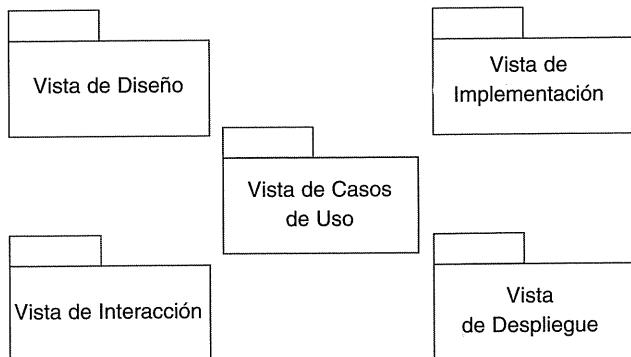


Figura 12.6: Modelado de las vistas arquitectónicas.

Sugerencias y consejos

Cuando se modelan paquetes en UML hay que recordar que existen sólo para ayudar a organizar los elementos del modelo. Si se tienen abstracciones que se manifiestan como objetos en el sistema real, no se deben utilizar paquetes. En vez de ello, se utilizarán elementos de modelado, tales como clases o componentes. Un paquete bien estructurado:

- Es cohesivo, y proporciona un límite bien definido alrededor de un conjunto de elementos relacionados.
- Está poco acoplado: exporta sólo aquellos elementos que otros paquetes necesitan ver realmente, e importa sólo aquellos elementos necesarios y suficientes para que los elementos del paquete hagan su trabajo.
- No está profundamente anidado, porque las capacidades humanas para comprender estructuras profundamente anidadas son limitadas.
- Posee un conjunto equilibrado de elementos; los paquetes de un sistema no deben ser demasiado grandes en relación a los otros (si es necesario, deben dividirse) ni demasiado pequeños (deben combinarse los elementos que se manipulen como un grupo).

Cuando se dibuje un paquete en UML:

- Hay que emplear la forma simple del icono de un paquete a menos que sea necesario revelar explícitamente el contenido.
- Cuando se revele el contenido de un paquete, hay que mostrar sólo los elementos necesarios para comprender el significado del paquete en el contexto.
- Especialmente si se están usando los paquetes para modelar elementos sujetos a una gestión de configuraciones, hay que revelar los valores de las etiquetas asociadas a las versiones.



Capítulo 13 INSTANCIAS

En este capítulo

- Instancias y objetos.
- Modelado de instancias concretas.
- Modelado de instancias prototípicas.
- El mundo real y el mundo conceptual de las instancias.

Véase el Capítulo 15 para una discusión de la estructura interna, la cual es preferible cuando se trabaja con objetos prototípicos y roles.

Los términos “instancia” y “objeto” son en gran parte sinónimos y, por ello, la mayoría de las veces pueden intercambiarse. Una instancia es una manifestación concreta de una abstracción a la que se puede aplicar un conjunto de operaciones y que puede tener un estado que almacena los efectos de la operación.

Las instancias se utilizan para modelar cosas concretas del mundo real. Casi todos los bloques de construcción de UML participan de esta dicotomía clase/objeto. Por ejemplo, puede haber casos de uso e instancias de casos de uso, nodos e instancias de nodos, asociaciones e instancias de asociaciones, etc.

Introducción

Supongamos que alguien ha decidido construir una casa para su familia. Al decir “casa” en vez de “coche”, ya se ha empezado a estrechar el vocabulario del espacio de la solución. Casa es una abstracción de “una morada permanente o semipermanente cuyo propósito es proporcionar cobijo”. Coche es “un vehículo móvil impulsado con algún tipo de energía cuyo propósito es transportar gente de un lugar a otro”. Mientras intentan compaginarse los requisitos contradictorios que conforman el problema, se deseará refinar la abstracción de esta casa. Por ejemplo, podría decidirse una “casa con tres habitaciones y un sótano”, un tipo de casa, si bien más especializado.

Cuando el constructor finalmente entrega las llaves de la casa y la familia entra por la puerta principal, la familia está frente a algo concreto y específico. Para esta familia ya no sería una casa con tres habitaciones y un sótano, sino que es “mi casa con tres habitaciones y un sótano, situada en la calle Primero de mayo, número 31”. Si la familia es especialmente sentimental, incluso podría darle un nombre como Santuario o Mi Ruina.

Hay una diferencia fundamental entre una casa de tres habitaciones y un sótano y mi casa de tres habitaciones llamada Santuario. La primera es una abstracción que representa un cierto tipo de casa con varias propiedades; la segunda es una instancia concreta de esa abstracción, y representa algo que se manifiesta en el mundo real, con valores reales para cada una de esas propiedades.

Una abstracción denota la esencia ideal de una cosa; una instancia denota una manifestación concreta. Esta separación de abstracción e instancia aparecerá en todo lo que se modele. Para una abstracción dada puede haber innumerables instancias. Para una instancia dada, habrá una abstracción que especifique las características comunes a todas las instancias.

Las clases se discuten en los Capítulos 4 y 9; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los casos de uso se discuten en el Capítulo 17. En realidad, UML utiliza el término especificación de instancia, pero esto es una sutileza del metamodelo.

En UML se pueden representar abstracciones y sus instancias. Casi todos los bloques de construcción de UML (principalmente las clases, componentes, nodos y casos de uso) pueden modelarse en función de su esencia o en función de sus instancias. La mayoría de las veces se trabajará con ellos como abstracciones. Cuando se desee modelar manifestaciones concretas será necesario trabajar con sus instancias.

UML proporciona una notación gráfica para las instancias, como se muestra en la Figura 13.1. Esta notación permite visualizar instancias con nombre, así como otras anónimas.

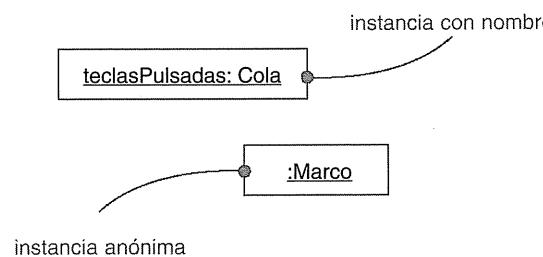


Figura 13.1: Instancias.

Términos y conceptos

La dicotomía clase/objeto de UML se discute en el Capítulo 2.

Las asociaciones se discuten en los Capítulos 5 y 10; los enlaces se discuten en los Capítulos 14 y 16.

Los clasificadores se discuten en el Capítulo 9.

Una *instancia* es una manifestación concreta de una abstracción a la que se puede aplicar un conjunto de operaciones y que posee un estado que almacena el efecto de las operaciones. *Instancia* y *objeto* son en gran parte sinónimos. Gráficamente, una instancia se representa subrayando su nombre.

Nota: Normalmente, la manifestación concreta de una clase se llama *objeto*. Los objetos son instancias de clases, por lo que es perfectamente apropiado decir que todos los objetos son instancias, aunque algunas instancias no son objetos (por ejemplo, una instancia de una asociación no es un objeto, es sólo una instancia, también llamada *enlace*). Sólo los modeladores avanzados se preocuparán realmente de esta sutil distinción.

Abstracciones e instancias

Las instancias no aparecen aisladas; casi siempre están ligadas a una abstracción. La mayoría de las instancias que se modelen en UML serán instancias de clases (y éstas se llaman *objetos*), aunque se pueden tener instancias de otros elementos, como componentes, nodos, casos de uso y asociaciones. En UML una instancia es fácilmente distinguible de una abstracción. Para identificar una instancia, se subraya su nombre.

En sentido general, un *objeto* es algo que ocupa espacio en el mundo real o conceptual, y al que se le pueden hacer cosas. Por ejemplo, una instancia de un nodo es normalmente un computador que se encuentra físicamente en una habitación; una instancia de un componente ocupa algo de espacio en el sistema de archivos; una instancia de un registro de un cliente consume algo de memoria física. Análogamente, una instancia de un billete de avión es algo sobre lo que se pueden hacer cálculos.

Se puede utilizar UML para modelar estas instancias físicas, pero también se pueden modelar cosas que no son tan concretas. Por ejemplo, una clase abstracta, por definición, no puede tener instancias directas. Sin embargo, se pueden modelar instancias indirectas de clases abstractas para mostrar el uso de una instancia prototípica de esa clase abstracta. Literalmente, un objeto así no puede existir. Pero, en la práctica, esa instancia permite dar nombre a una de las instancias potenciales de los hijos concretos de esa clase abstracta. Lo mismo se aplica a las interfaces. Por su propia definición, las interfaces no pueden tener instancias directas, pero se puede modelar una instancia prototípica de una

Los diagramas de objetos se discuten en el Capítulo 14.

interfaz, que representará a una de las instancias potenciales de las clases que realicen esa interfaz.

Cuando se modelan instancias, éstas se colocan en los diagramas de objetos (si se desea visualizar sus detalles estructurales) o en diagramas de interacción y de actividades (si se desea visualizar su participación en situaciones dinámicas). Aunque normalmente no es necesario, se pueden colocar objetos en diagramas de clases para representar explícitamente la relación de un objeto con su abstracción.

Tipos

Los diagramas de interacción se discuten en el Capítulo 19; los diagramas de actividades se discuten en el Capítulo 20; los tipos dinámicos se discuten en el Capítulo 11; los clasificadores se discuten en el Capítulo 9.

Una instancia tiene un tipo. El tipo de una instancia real debe ser un clasificador concreto, pero la especificación de una instancia (que no representa a una instancia particular) puede tener un tipo abstracto. En la notación, el nombre de la instancia va seguido de dos puntos y del tipo, por ejemplo, `t : Transaccion`.

El clasificador de una instancia es normalmente estático. Por ejemplo, una vez creada una instancia de una clase, su clase no cambiará durante la vida del objeto. Sin embargo, en algunas situaciones de modelado y en algunos lenguajes de programación, es posible cambiar la abstracción de una instancia. Por ejemplo, un objeto Oruga podría convertirse en un objeto Mariposa. Es el mismo objeto, pero de una diferente abstracción.

Nota: Durante el desarrollo también es posible tener instancias sin ningún clasificador asociado, que se pueden representar como un objeto sin nombre de abstracción, tal como se muestra en la Figura 13.2. Estos objetos huérfanos se pueden introducir cuando se necesita modelar un comportamiento muy abstracto, aunque finalmente se deben ligar tales instancias a una abstracción si se quiere imponer algún grado de semántica sobre el objeto.

Nombres

Las operaciones se discuten en los Capítulos 4 y 9; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27.

Cada instancia debe tener un nombre que la distinga de las otras instancias dentro de su contexto. Normalmente, un objeto existe en el contexto de una operación, un componente o un nodo. Un *nombre* es una cadena de texto, como `t` y `miCliente`, en la Figura 13.2. Ese nombre solo se llama *nombre simple*. La abstracción de la instancia puede tener un nombre simple (como `Transaccion`) o puede tener un *nombre calificado* (como `Multimedia : AudioStream`), el cual consta del nombre de la abstracción precedido por el nombre del paquete en el que ésta se encuentra.

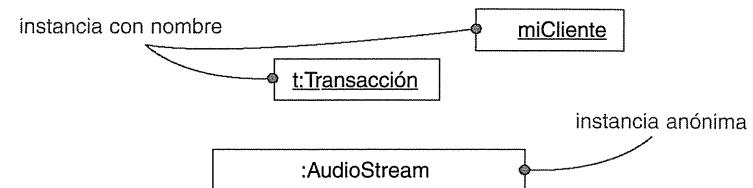


Figura 13.2: Instancias con nombre y anónimas.

Cuando se da nombre a un objeto de forma explícita, realmente se le está dando un nombre (como `miCliente`) utilizable por una persona. También se puede dar nombre simplemente a un objeto (como `miCliente`) y omitir el nombre de su abstracción si es obvia en el contexto dado. En muchos casos, sin embargo, el nombre real de un objeto sólo es conocido por el computador en el que existe. En tales casos, se puede representar un objeto anónimo (como `:AudioStream`). Cada aparición de un objeto anónimo se considera distinta de las otras apariciones. Incluso si no se conoce la abstracción asociada al objeto, al menos se le puede dar un nombre explícito (tal como `agente :`).

El nombre y el tipo de un objeto forman una cadena en la notación, por ejemplo, `t : Transaccion`. Para un objeto (en oposición a un rol dentro de una clase estructurada), se debe subrayar la cadena completa.

Nota: El nombre de una instancia puede ser texto formado por cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se utilizan para separar el nombre de una instancia del nombre de su abstracción) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de las instancias son nombres cortos o expresiones nominales extraídos del vocabulario del sistema que se está modelando. Normalmente, en el nombre de una instancia se pone en mayúsculas la primera letra de cada palabra excepto la primera, como en `t` o `miCliente`.

Operaciones

Las operaciones se discuten en los Capítulos 4 y 9; el polimorfismo se discute en el Capítulo 9.

Un objeto no sólo es algo que normalmente ocupa espacio en el mundo real; también es algo a lo que se le pueden hacer cosas. Las operaciones que se pueden ejecutar sobre un objeto se declaran en la abstracción del objeto. Por ejemplo, si la clase `Transaccion` define la operación `commit`, entonces, dada la instancia `t : Transaccion`, se pueden escribir expresiones como

`t.commit()`. La ejecución de esta expresión significa que sobre `t` (el objeto) opera `commit` (la operación). Dependiendo de la jerarquía de herencia asociada con `Transaccion`, esta operación podría ser invocada polimórficamente o no.

Estado

Un objeto también tiene estado, que en este sentido incluye todas las propiedades del objeto más los valores actuales de esas propiedades (incluyendo también enlaces y objetos relacionados, según el punto de vista del modelador). Estas propiedades incluyen los atributos y asociaciones del objeto, así como sus partes agregadas. El estado de un objeto es, pues, dinámico. De forma que, al visualizar su estado, realmente se está especificando el valor de su estado en un momento dado del tiempo y del espacio. Es posible mostrar el cambio de estado de un objeto representándolo muchas veces en el mismo diagrama de interacción, pero con cada ocurrencia representando un estado diferente.

Los atributos se discuten en el Capítulo 4; los diagramas de interacción se discuten en el Capítulo 19. Otra forma de mostrar el cambio de estado de un objeto individual a lo largo del tiempo es utilizar máquinas de estados, que se discuten en el Capítulo 22.

Cuando se opera sobre un objeto, normalmente se cambia su estado; cuando se consulta a un objeto, su estado no se modifica. Por ejemplo, al hacer una reserva aérea (representada por el objeto `r : Reserva`), se podría establecer el valor de uno de sus atributos (por ejemplo, `precio = 395.75`). Si se modifica la reserva, quizás añadiendo una nueva ruta al itinerario, entonces el estado podría cambiar (por ejemplo, `precio = 1024.86`).

Como se muestra en la Figura 13.3, se puede emplear UML para mostrar el valor de los atributos de un objeto. Por ejemplo, `miCliente` se muestra con el atributo `id` contenido el valor “432-89-1783”. En este caso, el tipo de `id` (`SSN`) se muestra explícitamente, aunque puede omitirse (como en `activo = True`), porque su tipo se puede encontrar en la declaración de `id` en la clase asociada a `miCliente`.

Se puede asociar una máquina de estados con una clase, lo cual es especialmente útil al modelar sistemas dirigidos por eventos o al modelar el tiempo de vida de una clase. En estos casos, también se puede mostrar el estado de esta máquina para un objeto dado en un momento dado. El estado se muestra entre corchetes después del tipo. Por ejemplo, como se muestra en la Figura 13.3, el objeto `c` (una instancia de la clase `Teléfono`) se encuentra en el estado `EsperandoRespuesta`, un estado con nombre definido en la máquina de estados de `Teléfono`.

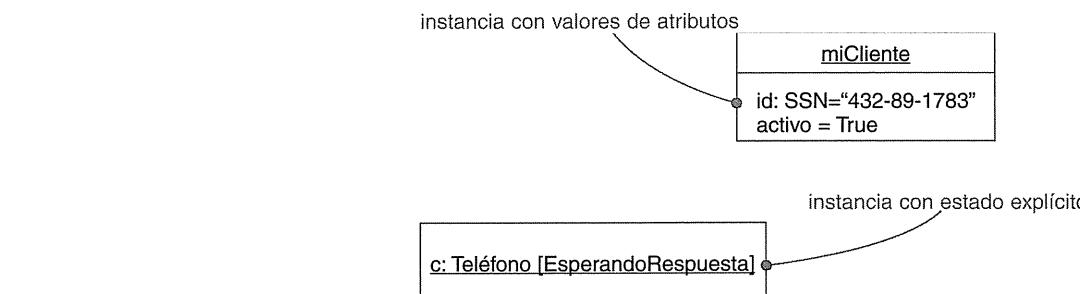


Figura 13.3: Estado de un objeto.

Nota: Ya que un objeto puede estar en varios estados simultáneamente, también se puede mostrar una lista de estados actuales.

Otras características

Los procesos e hilos se discuten en el Capítulo 23.

Los procesos e hilos son un elemento importante de la vista de procesos de un sistema, así que UML proporciona una señal visual para distinguir los elementos activos (aquellos que forman parte de un proceso o hilo y representan una raíz de un flujo de control) de los pasivos. Se pueden declarar clases activas que materializan un proceso o hilo, y a su vez se puede distinguir una instancia de una clase activa, como se muestra en la Figura 13.4.

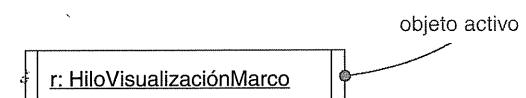


Figura 13.4: Objetos activos.

Los diagramas de interacción se discuten en el Capítulo 19.

Los enlaces se discuten en los Capítulos 14 y 16; los atributos y las operaciones estáticas se discuten en el Capítulo 9.

Nota: La mayoría de las veces se utilizarán los objetos activos en el contexto de diagramas de interacción que modelan múltiples flujos de control. Cada objeto activo representa la raíz de un flujo de control y puede utilizarse para nombrar a distintos flujos.

Hay otros dos elementos de UML que pueden tener instancias. El primero es una asociación. Una instancia de una asociación es un enlace. Un enlace es una conexión semántica entre objetos. Un enlace se representa con una línea, al igual que una asociación, pero puede distinguirse de ella porque los enlaces sólo conectan objetos.

El segundo tipo de instancia es un atributo estático (o atributo de clase). Un atributo estático es, en efecto, un objeto que pertenece a la clase, y que es accesible

ble por todas las instancias de ésta. Por tanto, en una declaración de la clase, se muestra como un atributo subrayado.

Elementos estándar

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Todos los mecanismos de extensibilidad de UML se aplican a los objetos. Sin embargo, normalmente, no se aplica un estereotipo directamente a una instancia, ni se le da un valor etiquetado propio. En vez de eso, el estereotipo y los valores etiquetados de una instancia derivan del estereotipo y valores etiquetados de su abstracción asociada. Por ejemplo, como se muestra en la Figura 13.5, se puede indicar explícitamente un estereotipo de un objeto, así como su abstracción.

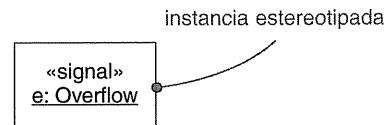


Figura 13.5: Objetos estereotipados.

UML define dos estereotipos estándar que se aplican a las relaciones de dependencia entre objetos y entre clases:

1. **instanceOf** Especifica que el objeto cliente es una instancia del clasificador proveedor de la dependencia. Esto raras veces se representa gráficamente; normalmente se muestra mediante notación textual, después de un símbolo de dos puntos.
2. **instantiate** Especifica que la clase cliente crea instancias de la clase proveedora.

Técnicas comunes de modelado

Modelado de instancias concretas

Cuando se modelan instancias concretas, en realidad se están visualizando cosas que existen en el mundo real. Por ejemplo, no se puede ver exactamente una instancia de una clase `Cliente`, a menos que ese cliente esté delante de uno; sin embargo, en un depurador debe ser posible ver una representación de ese objeto.

Los diagramas de componentes se discuten en el Capítulo 15; los diagramas de despliegue se discuten en el Capítulo 31; los diagramas de objetos se discuten en el Capítulo 14.

Una de las cosas para las que se emplean los objetos es para modelar instancias concretas del mundo real. Por ejemplo, si se quiere modelar la topología de una red de computadores de una empresa, se utilizarán diagramas de despliegue que contendrán instancias de nodos. Análogamente, si se desea modelar los componentes que se encuentran en los nodos físicos de esta red, se utilizarán diagramas de componentes que contendrán instancias de los componentes. Por último, si se dispone de un depurador conectado al sistema en ejecución, éste podría mostrar las relaciones estructurales entre instancias dibujando un diagrama de objetos.

Para modelar instancias concretas:

- Hay que identificar aquellas instancias que son necesarias y suficientes para visualizar, especificar, construir o documentar el problema que se está modelando.
- Hay que representar esos objetos en UML como instancias. Cuando sea posible, hay que dar un nombre a cada objeto. Si no hay un nombre significativo para el objeto, puede representarse como un objeto anónimo.
- Hay que mostrar el estereotipo, los valores etiquetados y los atributos (con sus valores) de cada instancia necesarios y suficientes para modelar el problema.
- Hay que representar estas instancias y sus relaciones en un diagrama de objetos u otro diagrama apropiado para el tipo de instancia.

Por ejemplo, la Figura 13.6 muestra un diagrama de objetos extraído de la ejecución de un sistema de validación de tarjetas de crédito, quizás tal como sería visto por un depurador que estuviera probando el sistema en ejecución.

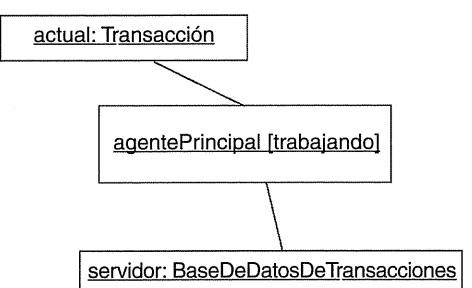


Figura 13.6: Modelado de instancias concretas.

Sugerencias y consejos

Cuando se modelan instancias en UML, hay que recordar que toda instancia debe denotar una manifestación concreta de alguna abstracción, normalmente una clase, componente, nodo, caso de uso o asociación. Una instancia bien estructurada:

- Está asociada explícitamente con una abstracción específica.
- Tiene un nombre único extraído del vocabulario del dominio del problema o del dominio de la solución.

Cuando se dibuje una instancia en UML:

- Hay que representar el nombre de la abstracción de la cual es instancia, a menos que sea obvio por el contexto.
- Hay que mostrar el estereotipo de la instancia y el estado sólo hasta donde sea necesario para comprender el objeto en su contexto.
- Si son visibles, las listas largas de atributos y sus valores deben organizarse agrupándolos según su categoría.



Capítulo 14 DIAGRAMAS DE OBJETOS

En este capítulo

- Modelado de estructuras de objetos.
- Ingeniería directa e inversa.

Los diagramas de objetos modelan las instancias de los elementos existentes en los diagramas de clases. Un diagrama de objetos muestra un conjunto de objetos y sus relaciones en un momento concreto.

Los diagramas de objetos se utilizan para modelar la vista de diseño estática o la vista de procesos estática de un sistema. Esto conlleva el modelado de una instantánea del sistema en un momento concreto y la representación de un conjunto de objetos, sus estados y sus relaciones.

Los diagramas de objetos no sólo son importantes para visualizar, especificar y documentar modelos estructurales, sino también para construir los aspectos estáticos de sistemas a través de ingeniería directa e inversa.

Introducción

Si uno no conoce bien el juego, el fútbol parece un deporte muy sencillo (un grupo incontrolado de personas que corren locamente sobre un terreno de juego persiguiendo una pelota). Al mirar la difusa imagen de cuerpos en movimiento, difícilmente parece que haya alguna sutileza o estilo en ello.

Si se congela el movimiento un momento y se clasifican los jugadores individuales, aparece una imagen muy diferente del juego. Ya no es una masa de gente, sino que se pueden distinguir delanteros, centrocampistas y defensas.

Profundizando un poco se entenderá cómo colaboran los jugadores, siguiendo estrategias para conseguir el gol, mover el balón, robar el balón y atacar. En un equipo ganador no se encontrarán jugadores colocados al azar en el campo.

En realidad, en cualquier momento del juego, su colocación en el campo y sus relaciones con otros jugadores están bien calculadas.

Algo parecido ocurre al intentar visualizar, especificar, construir o documentar un sistema con gran cantidad de software. Si se fuera a trazar el flujo de control de un sistema en ejecución, rápidamente se perdería la visión de conjunto de cómo se organizan las partes del sistema, especialmente si se tienen varios hilos de control. Análogamente, si se tiene una estructura de datos compleja, no ayuda mucho mirar simplemente el estado de un objeto en un momento dado. En vez de ello, es necesario estudiar una instantánea del objeto, sus vecinos y las relaciones con estos vecinos. En todos los sistemas orientados a objetos, excepto en los más simples, existirá una multitud de objetos, cada uno de los cuales mantiene una relación precisa con los demás. De hecho, cuando un sistema orientado a objetos falla, no suele ser por un fallo en la lógica, sino porque se rompen conexiones entre objetos o por un estado no válido en objetos individuales.

Los diagramas de clases se discuten en el Capítulo 8; las interacciones se discuten en el Capítulo 16; los diagramas de interacción se discuten en el Capítulo 19.

Con UML, los diagramas de clases se utilizan para visualizar los aspectos estáticos de los bloques de construcción del sistema. Los diagramas de interacción se utilizan para ver los aspectos dinámicos del sistema, y constan de instancias de estos bloques de construcción y los mensajes enviados entre ellos. Un diagrama de objetos contiene un conjunto de instancias de los elementos existentes en un diagrama de clases. Por tanto, un diagrama de objetos expresa la parte estática de una interacción, y consta de los objetos que colaboran, pero sin ninguno de los mensajes que se envían entre ellos. En ambos casos, un diagrama de objetos congela un instante en el tiempo, como se muestra en la Figura 14.1.

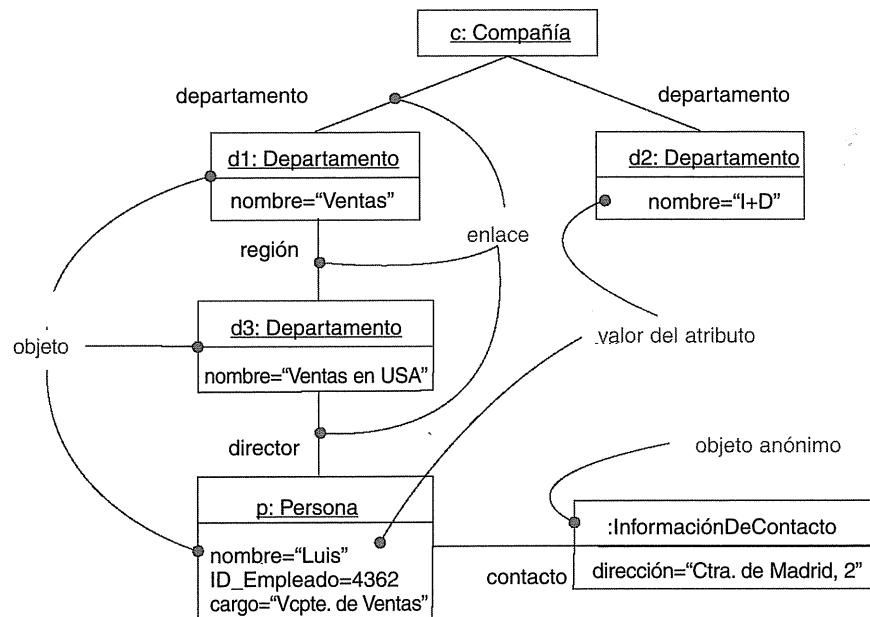


Figura 14.1: Un diagrama de objetos.

Términos y conceptos

Un *diagrama de objetos* es un diagrama que representa un conjunto de objetos y sus relaciones en un momento concreto. Gráficamente, un diagrama de objetos es una colección de nodos y arcos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de objetos es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de objetos de los otros tipos de diagramas es su contenido particular.

Los objetos se discuten en el Capítulo 13; los enlaces se discuten en el Capítulo 16.

Contenidos

Los diagramas de objetos normalmente contienen:

- Objetos.
- Enlaces.

Al igual que los demás diagramas, los diagramas de objetos pueden contener notas y restricciones.

A veces se colocarán clases en los diagramas de objetos, especialmente cuando se quiera mostrar la clase que hay detrás de cada instancia.

Los diagramas de clases se discuten en el Capítulo 8; los diagramas de interacción se discuten en el Capítulo 19.

Nota: Un diagrama de objetos se relaciona con un diagrama de clases: el diagrama de clases describe la situación general, y el diagrama de instancias describe instancias específicas derivadas del diagrama de clases. Un diagrama de objetos contiene principalmente objetos y enlaces. Los diagramas de despliegue también pueden aparecer en formato genérico o de instancias: los diagramas de despliegue general describen tipos de nodos, y los diagramas de despliegue de instancias describen una configuración concreta de instancias de nodos descritos por esos tipos.

Usos comunes

Las vistas de diseño se discuten en el Capítulo 2.

Los diagramas de objetos se emplean para modelar la vista de diseño estática o la vista de procesos estática de un sistema al igual que se hace con los diagramas de clases, pero desde la perspectiva de instancias reales o prototípicas. Esta vista sustenta principalmente los requisitos funcionales de un sistema (o sea, los servicios que debe proporcionar el sistema a sus usuarios finales). Los diagramas de objetos permiten modelar estructuras de datos estáticas.

Al modelar la vista de diseño estática o la vista de interacción estática de un sistema, normalmente los diagramas de objetos se utilizan para modelar estructuras de objetos.

Los diagramas de interacción se discuten en el Capítulo 19.

El modelado de estructuras de objetos implica tomar una instantánea de los objetos de un sistema en un momento dado. Un diagrama de objetos representa una escena estática dentro de la historia representada por un diagrama de interacción. Los diagramas de objetos se emplean para visualizar, especificar, construir y documentar la existencia de ciertas instancias en el sistema, junto a las relaciones entre ellas. El comportamiento dinámico y la ejecución se pueden representar como una secuencia de escenas.

Técnicas comunes de modelado

Modelado de estructuras de objetos

Cuando se construye un diagrama de clases, de componentes o de despliegue, lo que realmente se está haciendo es capturar un conjunto de abstracciones que son interesantes como grupo y, en ese contexto, exhibir su semántica y las relaciones entre las abstracciones del grupo. Estos diagramas sólo muestran "lo que puede ser". Si la clase A tiene una asociación uno a muchos con la clase B, entonces para una instancia de A podría haber cinco instancias de B; para otra instancia de A podría haber sólo una instancia de B. Además, en un momento dado, esa instancia de A, junto con las instancias de B relacionadas, tendrá ciertos valores para sus atributos y máquinas de estados.

Si se congela un sistema en ejecución o uno se imagina un instante concreto en un sistema modelado, aparecerá un conjunto de objetos, cada uno de ellos en un estado específico, y con unas relaciones particulares con otros objetos. Los diagramas de objetos se pueden usar para visualizar, especificar, construir y documentar la estructura de esas instantáneas. Los diagramas de objetos son especialmente útiles para modelar estructuras de datos complejas.

Cuando se modela la vista de diseño de un sistema, se puede utilizar un conjunto de diagramas de clases para especificar completamente la semántica de las abstracciones y sus relaciones. Con los diagramas de objetos, sin embargo, no se puede especificar completamente la estructura de objetos del sistema. Puede existir una multitud de posibles instancias de una clase particular, y para un conjunto de clases con relaciones entre ellas, pueden existir muchas más configuraciones posibles de esos objetos. Por lo tanto, al utilizar diagramas de objetos sólo se pueden mostrar significativamente conjuntos interesantes de objetos concretos o prototípicos. Esto es lo que significa modelar una estructura de objetos (un diagrama de objetos muestra un conjunto de objetos relacionados entre sí en un momento dado).

Los mecanismos como éstos a menudo están asociados a casos de uso, como se discute en los Capítulos 17 y 29.

Para modelar una estructura de objetos:

- Hay que identificar el mecanismo que se desea modelar. Un mecanismo representa alguna función o comportamiento de la parte del sistema que se está modelando, que resulta de la interacción de una sociedad de clases, interfaces y otros elementos.

- Hay que crear una colaboración para describir un mecanismo.
- Para cada mecanismo, hay que identificar las clases, interfaces y otros elementos que participan en esa colaboración; también hay que identificar las relaciones entre estos elementos.
- Hay que considerar un escenario en el que intervenga este mecanismo. También hay que congelar ese escenario en un momento concreto, y representar cada objeto que participe en el mecanismo.
- Hay que mostrar el estado y los valores de los atributos de cada uno de esos objetos, si son necesarios para comprender el escenario.
- Análogamente, hay que mostrar los enlaces entre esos objetos, que representarán instancias de asociaciones entre ellos.

Por ejemplo, la Figura 14.2 representa un conjunto de objetos extraídos de la implementación de un robot autónomo. Esta figura se centra en algunos de los objetos implicados en el mecanismo utilizado por el robot para calcular un modelo del mundo en el que se mueve. Hay muchos más objetos implicados en un sistema en ejecución, pero este diagrama se centra sólo en aquellas abstracciones implicadas directamente en la creación de esta vista del mundo.

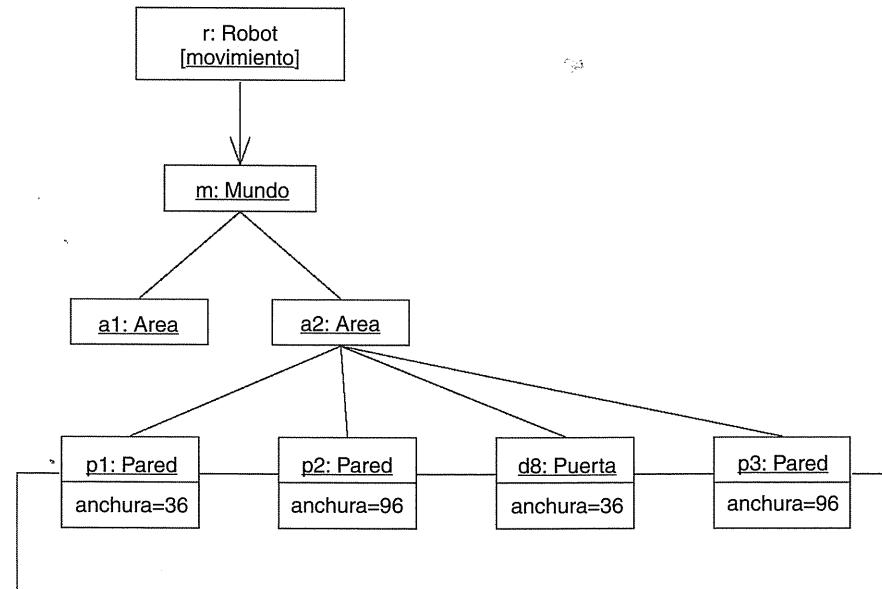


Figura 14.2: Modelado de estructuras de objetos.

Como indica la figura, un objeto representa al propio robot (r, una instancia de Robot), y r se encuentra actualmente en el estado movimiento. Este objeto tiene un enlace con m, una instancia de Mundo, que representa una abstracción del modelo del mundo del robot.

En ese instante, m está enlazado a dos instancias de Área. Una de ellas (a2) se muestra con sus propios enlaces a tres objetos Pared y un objeto Puerta. Cada una de estas paredes está etiquetada con su anchura actual, y cada una se muestra enlazada a sus paredes vecinas. Como sugiere este diagrama de objetos, el robot ha reconocido el área que lo contiene, que tiene paredes en tres lados y una puerta en el cuarto.

Ingeniería inversa

Hacer ingeniería inversa (creación de un modelo a partir del código) con un diagrama de objetos puede ser útil. De hecho, mientras se está depurando el sistema, esto es algo que el programador o las herramientas están haciendo continuamente. Por ejemplo, si se está persiguiendo un enlace perdido, uno dibujará mental o literalmente un diagrama de objetos de los objetos afectados, para ver dónde se invalida, en un momento dado, el estado de un objeto o su relación con otros objetos.

Para hacer ingeniería inversa con un diagrama de objetos:

- Hay que elegir el objetivo al que se desea aplicar la ingeniería inversa. Normalmente, se establecerá el contexto dentro de una operación o en relación con una instancia de una determinada clase.
- Hay que detener la ejecución en un determinado instante, utilizando una herramienta o simplemente recorriendo un escenario.
- Hay que identificar el conjunto de objetos interesantes que colaboran en el contexto, y representarlos en un diagrama de objetos.
- Si es necesario para comprender la semántica, hay que mostrar el estado de estos objetos.
- Si es necesario para comprender la semántica, hay que identificar los enlaces existentes entre estos objetos.
- Si el diagrama termina complicándose en exceso, se puede recortar, eliminando aquellos objetos que no sean pertinentes para las cuestiones sobre el escenario que necesitan respuesta. Si el diagrama es demasiado

simple, hay que expandir los vecinos de los objetos interesantes para mostrar con mayor profundidad el estado de cada objeto.

- Normalmente, habrá que añadir manualmente estructura que no sea explícita en el código destino. La información ausente proporciona el objetivo de diseño que sólo es implícito en el código final.

Sugerencias y consejos

Al crear diagramas de objetos en UML, hay que recordar que cada diagrama de objetos es sólo una representación gráfica de la vista de diseño estática o de la vista de interacción estática de un sistema. Esto significa que un único diagrama de objetos no necesita capturar todo sobre las vistas de diseño o de procesos de un sistema. De hecho, en cualquier sistema no trivial se encontrarán cientos, si no miles, de objetos, la mayoría anónimos. Así que es imposible especificar completamente todos los objetos de un sistema o todas las formas en que pueden asociarse estos objetos. Por consiguiente, los diagramas de objetos reflejan algunos de los objetos concretos o prototípicos de un sistema en ejecución.

Un diagrama de objetos bien estructurado:

- Se centra en comunicar un aspecto de la vista de diseño estática o la vista de procesos estática de un sistema.
- Representa una escena de la historia representada por un diagrama de interacción.
- Contiene sólo aquellos elementos esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con el nivel de abstracción, mostrando sólo aquellos valores de atributos y otros adornos que sean esenciales para su comprensión.
- No es tan minimalista que deje de informar al lector sobre la semántica importante.

Cuando se dibuje un diagrama de objetos:

- Hay que darle un nombre que comunique su propósito.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.

- Hay que organizar sus elementos espacialmente, de modo que los que estén cercanos semánticamente también lo estén físicamente.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre características importantes del diagrama.
- Hay que incluir los valores y el estado de cada objeto, si son necesarios para comunicar el propósito.



Capítulo 15 COMPONENTES

En este capítulo

- Componentes, interfaces y realización.
- Estructura interna, puertos, partes y conectores.
- Enlazar subcomponentes.
- Modelado de una API.

Un componente es una parte lógica y reemplazable de un sistema, que conforma y proporciona la realización de un conjunto de interfaces.

Los buenos componentes definen abstracciones precisas, con interfaces bien definidas, y facilitan la sustitución de los componentes viejos por otros más nuevos y compatibles.

Las interfaces conectan los modelos lógicos y de diseño. Por ejemplo, se puede especificar una interfaz o una clase en un modelo lógico, y esa misma interfaz conducirá a algún componente de diseño que la realice.

Las interfaces permiten implementar un componente utilizando componentes más pequeños mediante la conexión de los puertos de esos componentes.

Introducción

Cuando construimos una casa, podemos optar por instalar un sistema de cine doméstico. Podemos comprar una unidad simple que lo incluya todo: televisor, sintonizador, grabador de vídeo, reproductor de DVD y altavoces. Un sistema así es fácil de instalar y funciona muy bien si satisface nuestras necesidades. Sin embargo, una unidad de una sola pieza no es muy flexible. Estamos sujetos al

La salida del video se puede conectar a la entrada del amplificador porque resultan ser conectores del mismo tipo.

El software tiene la ventaja de que existe un número ilimitado de tipos de "conectores".

conjunto de características proporcionadas por el fabricante. Es posible que no podamos instalar altavoces de alta calidad. Si queremos instalar una nueva televisión de alta definición, habrá que descartar la unidad completa y sustituirla, incluyendo el vídeo y el reproductor de DVD, que puede que aún funcionen bien. Si tenemos una colección de cintas (puede que algún lector recuerde lo que son), mala suerte.

Hay un enfoque mucho más flexible, consistente en construir el sistema de cine a partir de componentes individuales, cada uno dirigido a una funcionalidad específica. Una pantalla muestra las imágenes; los altavoces individuales reproducen el sonido, y pueden ser colocados donde lo permitan la habitación y los oídos; el sintonizador de radio, el vídeo y el reproductor de DVD son cada uno unidades individuales, con sus capacidades ajustadas a nuestros gustos videófilos y a nuestro presupuesto. En lugar de bloquearlos de una manera rígida, colocamos cada componente donde queremos, y los conectamos con cables. Cada cable tiene un tipo específico de enchufe que encaja en un puerto correspondiente, de forma que no se puede conectar el cable de un altavoz en una salida de vídeo. También podemos conectar nuestro viejo tocadiscos si lo deseamos. Cuando queramos actualizar el sistema, podemos reemplazar un componente cada vez, sin deshacernos del sistema entero ni tener que comenzar de nuevo. Los componentes proporcionan más flexibilidad, y permiten obtener más calidad, si uno lo desea y puede permitírselo.

El software es parecido. Podemos construir una aplicación como una gran unidad monolítica, pero entonces será rígida y difícil de modificar cuando cambien las necesidades. Además, no podremos aprovechar las ventajas de las posibilidades existentes. Incluso si un sistema existente tiene la mayoría de la funcionalidad necesaria, también puede tener muchas otras partes no deseadas, y que sean difíciles o imposibles de eliminar. La solución para los sistemas software es la misma que para los sistemas electrónicos: construirlos a partir de componentes bien definidos que puedan enlazarse flexiblemente, y que puedan ser sustituidos cuando cambien los requisitos.

Términos y conceptos

Las interfaces se discuten en el Capítulo 11; las clases se discuten en los Capítulos 4 y 9.

Una *interfaz* es una colección de operaciones que especifican un servicio proporcionado o solicitado por una clase o componente.

Un *componente* es una parte reemplazable de un sistema que conforma y proporciona la implementación de un conjunto de interfaces.

Un *puerto* es una ventana específica en un componente encapsulado, que acepta mensajes hacia y desde el componente, que conforman con las interfaces especificadas.

La *estructura interna* es la implementación de un componente a través de un conjunto de partes conectadas de una manera específica.

Una *parte* es la especificación de un rol que forma parte de la implementación de un componente. En una instancia de un componente, existe una instancia correspondiente a la parte.

Un *conector* es una relación de comunicación entre dos partes o puertos dentro del contexto de un componente.

Componentes e interfaces

Las interfaces se discuten en el Capítulo 11.

El modelado de sistemas distribuidos se discute en el Capítulo 24.

La realización se discute en el Capítulo 10.

Una interfaz es una colección de operaciones que se utiliza para especificar un servicio de una clase o de un componente. La relación entre componente e interfaz es importante. Todos los recursos de los sistemas operativos más comunes (como COM+, CORBA y Enterprise Java Beans) utilizan las interfaces como el pegamento que une a los componentes.

Para construir un sistema basado en componentes, se descompone el sistema especificando interfaces que representan las principales líneas de separación del sistema. A continuación se proporcionan componentes que realizan las interfaces, junto con otros componentes que acceden a los servicios a través de sus interfaces. Este mecanismo permite desplegar un sistema cuyos servicios son en cierto modo independientes de la localización y, como se verá en la sección siguiente, sustituibles.

Una interfaz que es realizada por un componente se denomina una *interfaz proporcionada*, lo que significa que el componente proporciona la interfaz a otros componentes. Un componente puede declarar muchas interfaces proporcionadas. La interfaz que utiliza un componente se denomina *interfaz requerida*, lo que significa una interfaz con la que conforma el componente cuando solicita servicios de otros componentes. Un componente puede conformar con muchas interfaces requeridas. Un componente puede tanto proporcionar como requerir interfaces.

Como se indica en la Figura 15.1, un componente se representa como un rectángulo con un pequeño ícono con dos pestañas en su esquina superior derecha. El nombre del componente aparece en el rectángulo. Un componente puede

tener atributos y operaciones, pero éstos a menudo se ocultan en los diagramas. Un componente puede mostrar una red de estructura interna, como se describe más adelante en este capítulo.

La relación entre un componente y sus interfaces se puede mostrar de dos formas diferentes. La primera (y más frecuente) consiste en representar la interfaz en su formato icónico, abreviado. Una interfaz proporcionada se representa como un círculo unido al componente por una línea (una “piruleta”). Una interfaz requerida se representa como un semicírculo unido al componente por una línea (un “enchufe”). En ambos casos, el nombre de la interfaz se coloca junto al símbolo. La segunda manera representa la interfaz en su formato expandido, mostrando quizás sus operaciones. El componente que realiza la interfaz se conecta a ella con una relación de realización. El componente que accede a los servicios del otro componente a través de la interfaz se conecta a ella mediante una relación de dependencia.

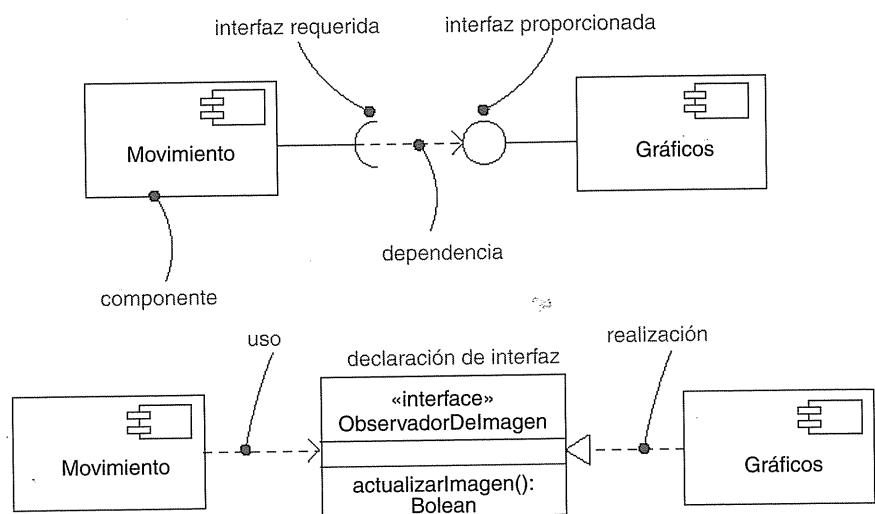


Figura 15.1: Componentes e interfaces.

Una interfaz determinada puede ser proporcionada por un componente y requerida por otro. El hecho de que la interfaz exista entre los dos componentes rompe la dependencia directa entre ellos. Un componente que utiliza una interfaz determinada funcionará bien independientemente de qué componente la realice. Por supuesto, un componente puede ser utilizado en un contexto si y sólo si todas sus interfaces requeridas son realizadas como interfaces proporcionadas por otros componentes.

Nota: Las interfaces se aplican a muchos niveles, al igual que muchos otros elementos. La interfaz a nivel de diseño que es usada o realizada por un componente se corresponderá con una interfaz a nivel de implementación usada o realizada por el artefacto que implementa el componente.

Sustitución

Los artefactos se discuten en el Capítulo 26.

El objetivo básico de cualquier recurso del sistema operativo basado en componentes es permitir el ensamblado de sistemas a partir de artefactos reemplazables. Esto significa que se puede diseñar un sistema utilizando componentes y luego implementar esos componentes utilizando artefactos. Posteriormente, el sistema puede evolucionar añadiendo nuevos componentes y sustituyendo los antiguos, sin reconstruir el sistema. Las interfaces son la clave para conseguir esto. En el sistema ejecutable, podemos usar cualquier artefacto que implemente a un componente que conforma o proporciona una interfaz. Podemos extender el sistema haciendo que los componentes proporcionen nuevos servicios a través de otras interfaces, las cuales pueden ser descubiertas y utilizadas a su vez por otros componentes.

Los sistemas y los subsistemas se discuten en el Capítulo 32.

Un componente es *reemplazable*. Un componente es sustituible (es posible reemplazarlo con otro que conforme con las mismas interfaces). En tiempo de diseño se puede elegir un componente diferente. Normalmente, el mecanismo para insertar o reemplazar un artefacto en tiempo de ejecución es transparente para el usuario del componente, y se consigue gracias a los modelos de objetos (como COM+ y Enterprise Java Beans) que requieren poca o ninguna intervención o por herramientas que automatizan el mecanismo.

Un componente es una *parte de un sistema*. Un componente raramente se encuentra aislado. En vez de ello, un componente dado colabora con otros componentes y al hacer esto existe en el contexto arquitectónico o tecnológico para el que se ha planificado. Un componente es cohesivo tanto lógica como físicamente, y por lo tanto denota un bloque significativo de comportamiento o estructural de un sistema mayor. Un componente puede ser reutilizado en muchos sistemas. Por tanto, un componente representa un bloque de construcción fundamental, que puede utilizarse para diseñar y componer sistemas. Esta definición es recursiva: un sistema a un nivel de abstracción puede ser tan sólo un componente a un nivel de abstracción más alto.

Por último, y tal como se ha descrito en la sección anterior, un componente *conforma y proporciona la realización de un conjunto de interfaces*.

Organización de componentes

Los paquetes se discuten en el Capítulo 12.

Las relaciones se discuten en los Capítulos 5 y 10.

Los componentes pueden organizarse agrupándolos en paquetes de la misma manera en que se organizan las clases.

También se pueden organizar los componentes especificando relaciones de dependencia, generalización, asociación (incluyendo a la agregación) y realización entre ellos.

Los componentes se pueden construir a partir de otros componentes. Véase la descripción de la estructura interna más adelante en este capítulo.

Puertos

Las interfaces son útiles al declarar el comportamiento global de un componente, pero no tienen identidad individual; la implementación del componente debe asegurar simplemente que todas las operaciones de todas las interfaces proporcionadas son implementadas. Para tener más control sobre la implementación, pueden utilizarse puertos.

Un *puerto* es una ventana explícita dentro de un *componente encapsulado*. En un componente encapsulado, todas las interacciones dentro y fuera del componente pasan a través de los puertos. El comportamiento externamente visible del componente es la suma de sus puertos, ni más ni menos. Además, un puerto tiene identidad. Otro componente puede comunicarse con un componente dado a través de un puerto específico. La comunicación queda completamente descrita por las interfaces que soporta el puerto, incluso si el componente soporta otras interfaces. En la implementación, las partes internas del componente pueden interactuar a través de un puerto externo específico, de forma que cada parte puede ser independiente de los requisitos de las otras partes. Los puertos permiten que las interfaces de un componente se dividan en paquetes discretos y que se utilicen de manera independiente. La encapsulación e independencia que proporcionan los puertos permiten un grado mucho mayor de encapsulación y posibilidades de sustitución.

Un puerto se representa como un pequeño cuadrado insertado en el borde de un componente (representa un agujero a través de la frontera de encapsulación del componente). Tanto las interfaces requeridas como las proporcionadas pueden enlazarse al puerto. Una interfaz proporcionada representa un servicio que puede ser solicitado a través de ese puerto. Una interfaz requerida representa un servicio que el puerto necesita obtener de algún otro componente. Cada puerto tiene

un nombre, de forma que pueda ser identificado de manera única, dados el componente y el nombre del puerto. El nombre del puerto puede ser utilizado por algunas partes internas del componente para identificar el puerto al cual enviar y del cual recibir los mensajes. El nombre del componente junto con el nombre del puerto identifican de manera única un puerto específico en un componente específico para que sea utilizado por otros componentes.

Los puertos forman parte de los componentes. Las instancias de los puertos se crean y se destruyen junto con la instancia del componente al que pertenecen. Los puertos también pueden tener multiplicidad: ésta indica el número posible de instancias de un puerto particular dentro de una instancia del componente. Cada puerto en una instancia de un componente tiene un vector de instancias del puerto. Aunque todas las instancias de un puerto que están en un vector satisfacen la misma interfaz y aceptan el mismo tipo de solicitudes, pueden tener diferentes estados y valores para los datos. Por ejemplo, cada instancia en el vector puede tener un nivel de prioridad diferente, de forma que las instancias de puerto más grandes sean servidas antes.

La Figura 15.2 muestra el modelo de un componente *Vendedor de entradas* con puertos. Cada puerto tiene un nombre y, opcionalmente, un tipo que indica la naturaleza del puerto. El componente tiene puertos para la venta de entradas, carga de espectáculos y cobro por tarjeta de crédito.

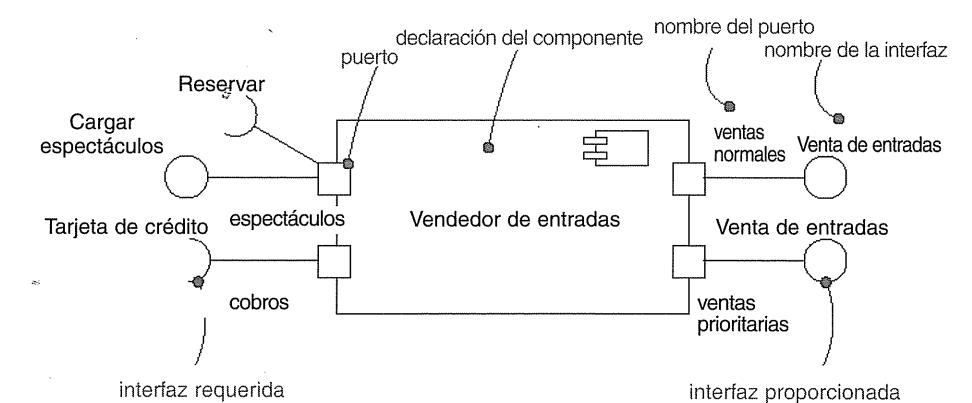


Figura 15.2: Puertos en un componente.

Hay dos puertos para la venta de entradas, uno para los clientes normales y otro para clientes prioritarios. Ambos tienen la misma interfaz de tipo *Venta de*

entradas. El puerto que procesa las tarjetas de crédito tiene una interfaz requerida; cualquier componente que proporcione los servicios especificados puede satisfacerla. A través de la interfaz Cargar espectáculos, una sala puede introducir espectáculos y actuaciones en la base de datos de entradas para vender. A través de la interfaz Reservar, el componente vendedor de entradas puede consultar a las salas sobre la disponibilidad de plazas, y llegar a comprar las entradas.

Estructura interna

Un componente puede implementarse como una única pieza de código, pero en los sistemas más grandes lo deseable es poder construir componentes más grandes utilizando componentes menores como bloques de construcción. La estructura interna de un componente está formada por las partes que componen la implementación del componente junto con las conexiones entre ellos. En muchos casos, las partes internas pueden ser instancias de componentes menores que se enlazan estáticamente a través de sus puertos para proporcionar el comportamiento necesario sin necesidad de que el modelador especifique una lógica adicional.

Una *parte* es una unidad de implementación de un componente. Una parte tiene nombre y un tipo. En una instancia de un componente, hay una o más instancias que se corresponden con cada parte que tiene el tipo especificado. Una parte tiene una multiplicidad dentro del componente. Si la multiplicidad de la parte es mayor que uno, puede haber más de una instancia de la parte en una instancia particular del componente. Si la multiplicidad es distinta de un simple número entero, el número de instancias de la parte puede variar de una instancia del componente a otra. Una instancia de un componente se crea con el mínimo número de partes; las partes adicionales se añaden posteriormente. Un atributo de una clase es una especie de parte: tiene un tipo y una multiplicidad, y cada instancia de la clase tiene una o más instancias del tipo dado.

Los diagramas de componentes se discuten en el Capítulo 15; los diagramas de despliegue se discuten en el Capítulo 31; los diagramas de objetos se discuten en el Capítulo 14.

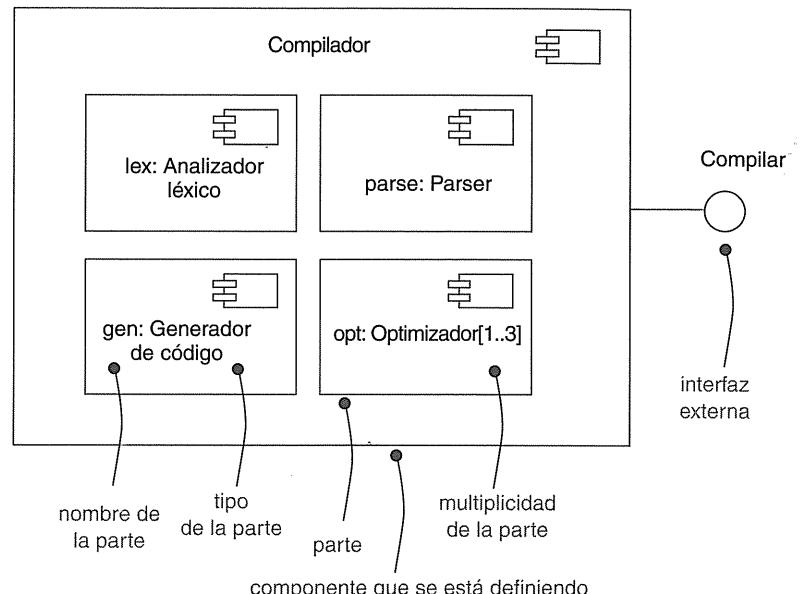


Figura 15.3: Partes dentro de un componente.

La Figura 15.3 muestra un componente de un compilador construido a partir de cuatro tipos de partes. Hay un analizador léxico, un analizador sintáctico, un generador de código, y de uno a tres optimizadores. Algunas versiones más completas del compilador pueden configurarse con distintos niveles de optimización; dentro de una versión particular, puede seleccionarse el optimizador apropiado en tiempo de ejecución.

Hay que tener en cuenta que una parte no es lo mismo que una clase. Cada parte es potencialmente distinguible por su nombre, como lo es cada atributo de una clase. Puede haber más de una parte del mismo tipo, pero pueden diferenciarse por los nombres, y, presumiblemente, cumplen funciones distintas dentro del componente. Por ejemplo, en la Figura 15.4 un componente Venta de Billetes puede tener diferentes partes Ventas para los pasajeros frecuentes y para los clientes normales; ambos trabajan igual, pero la parte de pasajeros frecuentes sólo está disponible para ciertos clientes especiales, y conlleva una menor probabilidad de espera en la fila, y puede incluir algunos privilegios adicionales. Como ambos componentes tienen el mismo tipo, han de tener nombres distintos para poder distinguirse. Los otros dos componentes de tipos AsignaciónDeAsiento y GestiónDeInventario no requieren nombres ya que sólo hay uno de cada tipo dentro del componente Venta de Billetes.

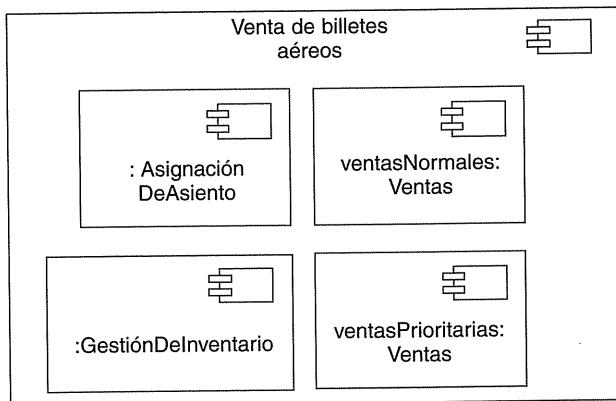


Figura 15.4: Partes del mismo tipo.

Si las partes son componentes con puertos, pueden enlazarse a través de éstos. La regla es bien sencilla: dos componentes pueden conectarse si uno proporciona una interfaz que el otro requiere. Conectar los puertos significa que el puerto que requiere invocará al puerto que proporciona para obtener servicios. La ventaja de los puertos e interfaces es que no hace falta saber nada más; si las interfaces son compatibles, los puertos pueden conectarse. Una herramienta podría generar automáticamente código de invocación desde un componente hacia otro. También se puede reconectar el componente a otros que proporcionen la misma interfaz, si éstos están disponibles. Una línea entre dos puertos es un *conector*. En una instancia del componente global, el conector representa un enlace o un enlace transitorio. Un enlace es una instancia de una asociación común. Un enlace transitorio representa una relación de uso entre dos componentes. En vez de corresponder a una asociación común, el enlace transitorio puede ser proporcionado por un parámetro de procedimiento o una variable local que sirve como destinataria de una operación. La ventaja de los puertos e interfaces es que los dos componentes no tienen que conocer nada acerca del otro en tiempo de diseño, siempre y cuando sus interfaces sean compatibles.

Los conectores se pueden representar de dos formas (Figura 15.5). Si dos componentes se enlazan de manera explícita, ya sea directamente o a través de sus puertos, simplemente se dibuja una línea entre ellos o los puertos. Por otro lado, si dos componentes se conectan porque tienen interfaces compatibles, se puede usar la notación de una junta circular para mostrar que no hay una relación inherente entre los componentes, aunque estén conectados dentro de este componente. Podemos colocar en ese sitio a cualquier otro componente que satisfaga la interfaz.

También se pueden conectar puertos internos a puertos externos del componente global. Esto se denomina un conector de delegación, ya que los mensajes sobre el puerto externo son delegados al puerto interno. Esto se representa con una flecha del puerto interno al externo. Se puede ver de dos maneras, según se prefiera. Por un lado, el puerto interno es el mismo que el puerto externo; ha sido movido a la frontera y se le ha permitido asomarse al exterior. En el segundo enfoque, cualquier mensaje hacia el puerto externo es transmitido inmediatamente hacia el puerto interno, y viceversa. En realidad no importa; el comportamiento es el mismo en ambos casos.

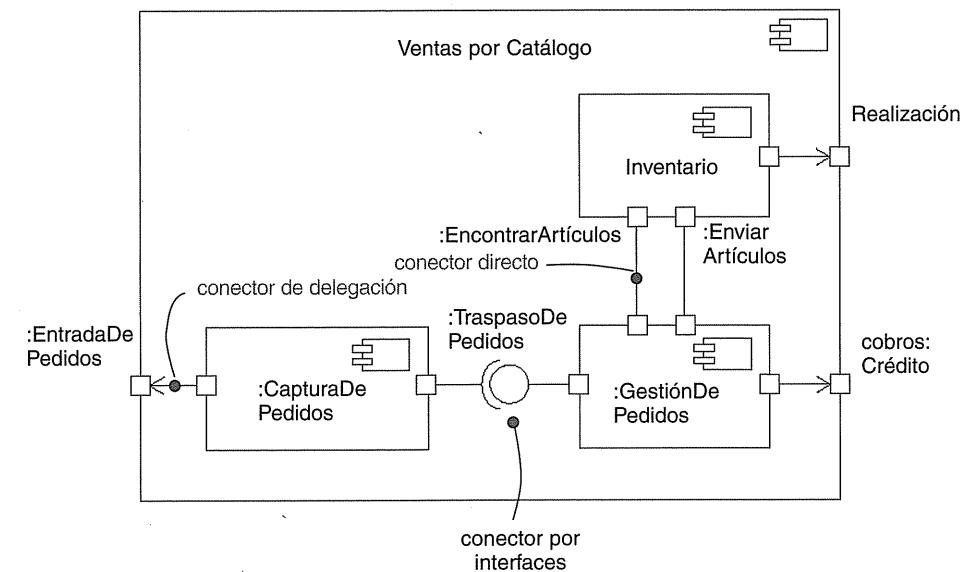


Figura 15.5: Conectores.

La Figura 15.5 muestra un ejemplo con puertos internos y diferentes tipos de conectores. Las solicitudes externas sobre el puerto EntradaDePedidos se delegan hacia el puerto interno del subcomponente CapturaDePedidos. Este componente a su vez envía su salida al puerto TraspasoDePedidos, que está conectado con una junta circular al subcomponente GestiónDePedidos. Este tipo de conexión implica que no existe un conocimiento especial entre ambos componentes; la salida podría estar conectada a cualquier componente que cumpliese la interfaz TraspasoDePedidos. El componente GestiónDePedidos se comunica con el componente Inventario para encontrar los artículos que quedan en el almacén. Esto se representa con un conector directo; como no se muestra ninguna interfaz, se está sugiriendo que la conexión está más fuertemente acoplada. Una vez encontrados los artículos en el almacén, el componente GestiónDePedidos accede a un servicio externo Crédito; esto se muestra por

Los diagramas de interacción se discuten en el Capítulo 19.

el conector de delegación hacia el puerto externo denominado cobros. Una vez que el servicio externo de crédito responde, el componente GestiónDePedidos se comunica con un puerto diferente EnvíoDeArtículos del componente Inventario para preparar el envío del pedido. El componente Inventario accede a un servicio externo Realización para ejecutar finalmente el envío.

Debe notarse que el diagrama de componentes muestra la estructura y los caminos potenciales de los mensajes del componente. El diagrama de componentes por sí mismo no revela la secuenciación de mensajes a través del componente. La secuenciación y otros tipos de información dinámica pueden representarse mediante diagramas de interacción.

Nota: La estructura interna, incluidos los puertos, partes y conectores, puede utilizarse como la implementación de cualquier clase, no sólo componentes. En realidad no hay mucha distinción semántica entre una clase y un componente. Sin embargo, a menudo es útil usar la convención de que los componentes se utilizan para conceptos encapsulados con estructura interna, en especial para esos conceptos que no se corresponden directamente con una clase simple en la implementación.

Técnicas comunes de modelado

Modelado de instancias concretas

Una clase estructurada puede utilizarse para modelar estructuras de datos en las que las partes tienen conexiones contextuales que sólo se aplican dentro de ella. Los atributos o las asociaciones normales pueden definir partes compuestas de una clase, pero puede que las partes no estén relacionadas en un diagrama de clases plano. Una clase cuya estructura interna se muestra con partes y conectores evita este problema.

Para modelar una clase estructurada:

- Hay que identificar las partes internas de la clase: y sus tipos.
 - Hay que dar a cada parte un nombre que indique su propósito dentro de la clase estructurada, no su tipo genérico.

- Hay que dibujar conectores entre las partes que se comunican o tienen relaciones contextuales.
 - Hay que sentirse libre para usar otras clases estructuradas como tipos de las partes, pero recordando que no se pueden hacer conexiones directas a las partes dentro de otra clase; hay que conectar a través de sus puertos externos.

La Figura 15.6 muestra el diseño de la clase estructurada PedidosDeBilletes. Esta clase tiene cuatro partes y un atributo normal, precio. El cliente es un objeto Persona. El cliente puede tener o no tener un estatus de prioridad, así que la parte prioridad se representa con una multiplicidad 0..1; el conector desde cliente a prioridad también tiene la misma multiplicidad. Hay uno o más asientos reservados; asiento tiene un valor de multiplicidad. No es necesario mostrar un conector desde cliente a asiento porque de todos modos están en la misma clase estructurada. Obsérvese que Espectáculo se ha dibujado con un borde discontinuo. Esto significa que la parte es una referencia a un objeto que no está contenido dentro de la clase estructurada. La referencia se crea y se destruye con una instancia de la clase PedidosDeBilletes, pero las instancias de Espectáculo son independientes de la clase PedidosDeBilletes. La parte asiento está conectada a la referencia espectáculo porque el pedido puede incluir asientos para más de un espectáculo, y cada reserva de asiento debe estar conectada a un espectáculo específico. De la multiplicidad del conector podemos deducir que cada reserva de Asiento está conectada exactamente a un objeto Espectáculo.

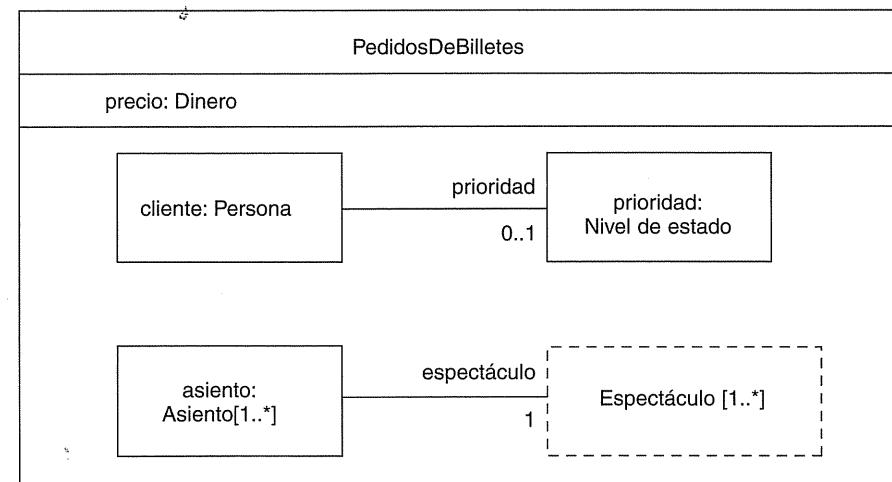


Figura 15.6: Clase estructurada.

Modelado de una API

Los desarrolladores que ensamblan un sistema a partir de partes componentes, a menudo desean ver las interfaces de programación de aplicaciones (API) que utilizan para enlazar esas partes. Las API representan las líneas de separación programáticas de un sistema, y se pueden modelar mediante interfaces y componentes.

Una API es esencialmente una interfaz realizada por uno o más componentes. Los desarrolladores realmente sólo se preocupan de la propia interfaz. El componente que realice las operaciones de la interfaz no es relevante, siempre y cuando haya *algún* componente que las realice. Sin embargo, desde la perspectiva de la gestión de la configuración de un sistema, estas realizaciones son importantes porque hay que asegurar que cuando se publica una API hay alguna realización disponible que cumple las obligaciones de la API. Afortunadamente, con UML podemos modelar ambas perspectivas.

Una API semánticamente rica tendrá muchas operaciones asociadas, por lo que la mayoría de las veces no hará falta visualizar todas las operaciones a la vez. En lugar de ello, nos decantaremos por mantener las operaciones en el trasfondo de los modelos y utilizaremos las interfaces como puntos de agarre con los cuales podamos encontrar esos conjuntos de operaciones. Si queremos construir sistemas ejecutables contra esas API, necesitaremos añadir bastante detalle, de forma que las herramientas de desarrollo puedan compilar frente a las propiedades de las interfaces. Además de las signaturas de las operaciones, probablemente también querremos incluir casos de uso que expliquen cómo utilizar cada interfaz.

Para modelar una API:

- Hay que identificar las líneas de separación del sistema y modelar cada una como una interfaz, recogiendo los atributos y operaciones que forman su frontera.
- Hay que exponer sólo aquellas propiedades de la interfaz que son importantes para comprender dentro del contexto dado; en otro caso, hay que esconder esas propiedades, manteniéndolas en la especificación de la interfaz por referencia, si son necesarias.
- Hay que modelar la realización de cada API sólo en tanto que sea importante para mostrar la configuración de una implementación específica.

La Figura 15.7 muestra la API de un componente de animación. Podemos ver cuatro interfaces que forman la API: `IAplicación`, `IModelos`, `IVisualización` y `IGuiones`. Otros componentes pueden usar una o más de estas interfaces, según las necesiten.

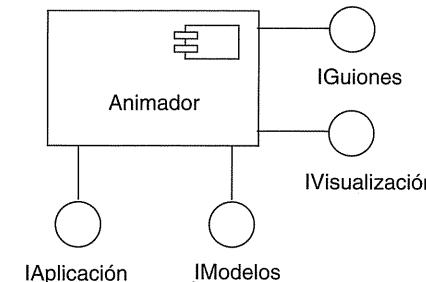


Figura 15.7: Modelado de una API.

Sugerencias y consejos

Los componentes permiten encapsular las partes de un sistema para reducir las dependencias, hacerlas explícitas y mejorar la flexibilidad y la posibilidad de sustitución cuando el sistema debe modificarse en el futuro. Un buen componente:

- Encapsula un servicio que tiene una interfaz y una frontera bien definidas.
- Tiene suficiente estructura interna para que merezca la pena describirla.
- No combina funcionalidades que no estén relacionadas en una única pieza.
- Organiza su comportamiento externo utilizando unas cuantas interfaces y puertos.
- Interactúa sólo a través de los puertos que ha declarado.

Si se quiere mostrar la implementación de un componente utilizando subcomponentes anidados:

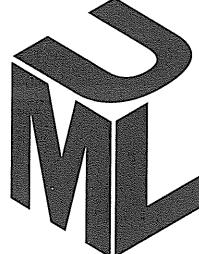
- Hay que utilizar un número pequeño de subcomponentes. Si hay demasiados para que quepan cómodamente en una página, hay que utilizar niveles adicionales de descomposición en algunos de los subcomponentes.

- Hay que asegurarse de que los subcomponentes interactúan sólo a través de los puertos y conectores definidos.
- Hay que determinar qué subcomponentes interactúan directamente con el mundo exterior y deben modelarse éstos con conectores de delegación.

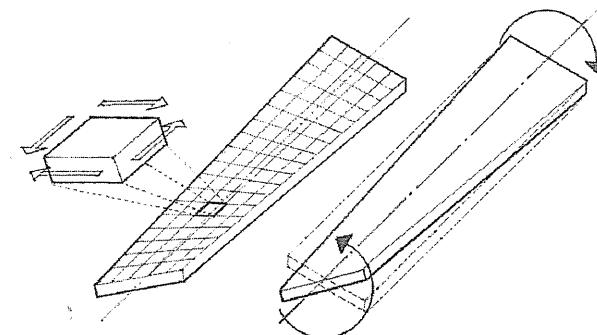
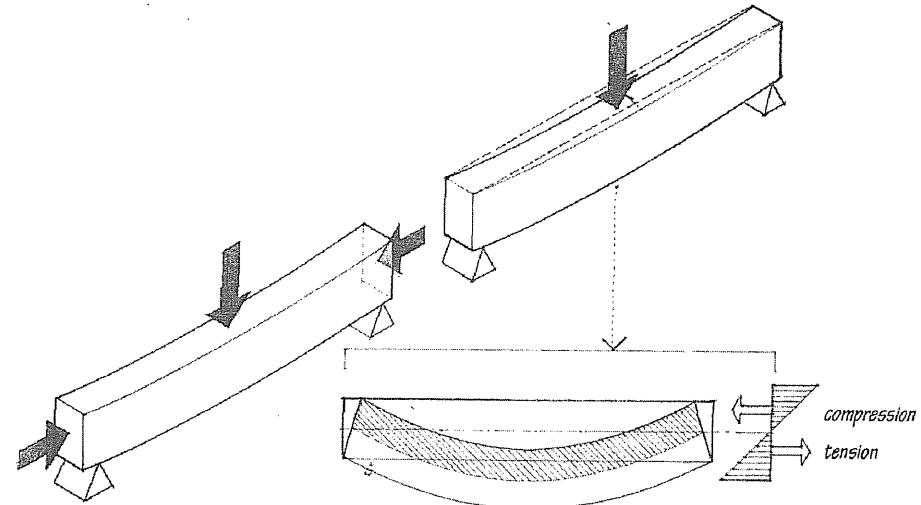
Cuando se dibuje un componente en UML:

- Hay que darle un nombre que indique claramente su propósito. Lo mismo debe hacerse con las interfaces.
- Hay que dar nombres a los subcomponentes y a los puertos si su significado no está claro a partir de sus tipos o si hay varias partes del mismo tipo.
- Hay que ocultar los detalles innecesarios. No hay que mostrar todos los detalles de la implementación sobre el diagrama de componentes.
- Hay que mostrar la dinámica de un componente mediante diagramas de interacción.

LENGUAJE
UNIFICADO DE
MODELADO



Parte 4 MODELADO BÁSICO DEL COMPORTAMIENTO



En este capítulo

- Roles, enlaces, mensajes, acciones y secuencias.
- Modelado de flujos de control.
- Creación de algoritmos bien estructurados.

En cualquier sistema interesante, los objetos interactúan entre sí pasándose mensajes. Una interacción es un comportamiento que incluye un conjunto de mensajes que se intercambian un conjunto de objetos dentro de un contexto para lograr un propósito.

Las interacciones se utilizan para modelar los aspectos dinámicos de las colaboraciones, que representan sociedades de objetos que desempeñan roles específicos, y colaboran entre sí para desarrollar un comportamiento mayor que la suma de los comportamientos de sus elementos. Estos roles representan instancias prototípicas de clases, interfaces, componentes, nodos y casos de uso. Los aspectos dinámicos se visualizan, se especifican, se construyen y se documentan como flujos de control que pueden incluir simples hilos secuenciales a través de un sistema, así como flujos más complejos que impliquen bifurcaciones, iteraciones, recursión y concurrencia. Cada interacción puede modelarse de dos formas: bien destacando la ordenación temporal de los mensajes, bien destacando la secuencia de mensajes en el contexto de una organización estructural de objetos.

Las interacciones bien estructuradas son como los algoritmos bien estructurados: eficientes, sencillas, adaptables y comprensibles.

Introducción

La diferencia entre construir una caseta para el perro y construir un rascacielos se discuten en el Capítulo 1.

Un edificio es algo vivo. Aunque todos los edificios se construyen a partir de materiales inertes, tales como ladrillos, masa, madera, plástico, vidrio y acero, estos elementos colaboran dinámicamente para llevar a cabo un comportamiento que es útil para los usuarios del edificio. Las puertas y ventanas se abren y se cierran. Las luces se encienden y se apagan. La caldera del edificio, el aire acondicionado, el termostato y los conductos de ventilación colaboran para regular la temperatura del edificio. En los edificios inteligentes, los sensores detectan la presencia o ausencia de actividad y ajustan la iluminación, la calefacción, la refrigeración y la música según cambian las condiciones. Los edificios se diseñan para facilitar el paso de gente y materiales de un lugar a otro. A un nivel más sutil, los edificios se diseñan para adaptarse a los cambios en la temperatura, expandiéndose y contrayéndose durante el día y la noche y a lo largo de las estaciones. Todos los edificios bien estructurados están diseñados para reaccionar a fuerzas dinámicas tales como vientos, terremotos y movimientos de sus ocupantes, de forma que siempre se mantenga en equilibrio.

El modelado de los aspectos estructurales de un sistema se discute en las Partes 2 y 3; también se pueden modelar los aspectos dinámicos de un sistema utilizando máquinas de estados, como se discute en el Capítulo 22; los diagramas de objetos se discuten en el Capítulo 14; los diagramas de interacción se discuten en el Capítulo 19; las colaboraciones se discuten en el Capítulo 28.

Los sistemas con gran cantidad de software son similares. El sistema de una compañía aérea puede manejar muchos terabytes de información que la mayor parte del tiempo permanecerán en algún disco sin ser accedidos, hasta que algún evento externo los ponga en acción, como, por ejemplo, una reserva, el movimiento de un avión o la programación de un vuelo. En los sistemas reactivos, como el sistema en el procesador de un horno microondas, la creación de objetos y el trabajo se llevan a cabo cuando se estimula el sistema con un evento, tal como la pulsación de un botón o el paso del tiempo.

En UML, los aspectos estáticos de un sistema se modelan mediante elementos tales como los diagramas de clases y los diagramas de objetos. Estos diagramas permiten especificar, construir y documentar los elementos del sistema, incluyendo clases, interfaces, componentes, nodos y casos de uso e instancias de ellos, así como la forma en la que estos elementos se relacionan entre sí.

En UML, los aspectos dinámicos de un sistema se modelan mediante interacciones. Al igual que un diagrama de objetos, una interacción establece el escenario para su comportamiento presentando todos los objetos que colaboran para realizar alguna acción. Pero, a diferencia de los diagramas de objetos, las interacciones incluyen los mensajes enviados entre objetos. La mayoría de las veces, un mensaje implica la invocación de una operación o el envío de una señal; un mensaje también puede incluir la creación o la destrucción de otros objetos.

Las interacciones se utilizan para modelar el flujo de control dentro de una operación, una clase, un componente, un caso de uso o el sistema completo. Con los diagramas de interacción, se puede razonar de dos formas sobre estos flujos. Una posibilidad es centrarse en cómo fluyen los mensajes a lo largo del tiempo. La otra posibilidad es centrarse en las relaciones estructurales entre los objetos de una interacción y después considerar cómo se pasan los mensajes en el contexto de esa estructura.

UML proporciona una representación gráfica para los mensajes, como se muestra en la Figura 16.1. Esta notación permite visualizar un mensaje de forma que permite destacar sus partes más importantes: nombre, parámetros (si tiene alguno) y secuencia. Gráficamente, un mensaje se representa como una línea dirigida y casi siempre incluye el nombre de su operación.

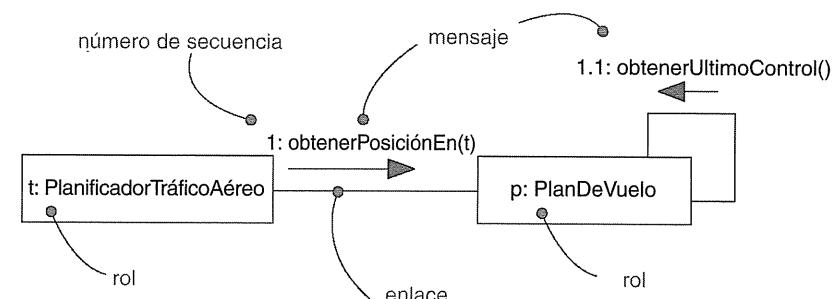


Figura 16.1: Mensajes, enlaces y secuenciación.

Términos y conceptos

Los diagramas de estructura interna muestran las conexiones estructurales entre roles, como se discute en el Capítulo 15; los objetos se discuten en el Capítulo 14; los sistemas y los subsistemas se discuten en el Capítulo 32; las colaboraciones se discuten en el Capítulo 28.

Una *interacción* es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos dentro de un contexto para lograr un propósito. Un *mensaje* es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad.

Contexto

Una interacción puede aparecer siempre que haya unos objetos enlazados a otros. Las interacciones aparecerán en la colaboración de objetos existentes en el contexto de un sistema o subsistema. También aparecerán interacciones en el contexto de una operación. Por último, aparecerán interacciones en el contexto de una clase.

Con mayor frecuencia, las interacciones aparecerán en la colaboración de objetos existentes en el contexto del sistema o de un subsistema. Por ejemplo, en un sistema de comercio electrónico sobre la Web, en el cliente habrá objetos que interactúen entre sí (por ejemplo, instancias de las clases PedidoDeLibro y FormularioDePedido). También habrá objetos en el cliente (de nuevo instancias de PedidoDeLibro) que interactúen con objetos del servidor (por ejemplo, instancias de GestorDePedidos). Por ello, estas interacciones no sólo implican colaboraciones locales de objetos (como las que hay en torno a FormulariodePedido), sino que también podrían atravesar muchos niveles conceptuales del sistema (como las interacciones que tienen que ver con GestorDePedidos).

Las operaciones se discuten en los Capítulos 4 y 9; el modelado de una operación se discute en los Capítulos 20 y 28.

Las interacciones entre objetos también aparecerán en la implementación de una operación. Los parámetros de una operación, sus variables locales y los objetos globales a la operación (pero visibles a ella) pueden interactuar entre sí para llevar a cabo el algoritmo de esa implementación de la operación. Por ejemplo, la invocación de la operación moverAPosicion(p: Posicion) definida en una clase de un robot móvil implicará la interacción de un parámetro (p), un objeto global a la operación (por ejemplo, el objeto posicionActual), y posiblemente varios objetos locales (como las variables locales utilizadas por la operación para calcular puntos intermedios en una trayectoria hasta la nueva posición).

Las clases se discuten en los Capítulos 4 y 9.

Por último, aparecerán interacciones en el contexto de una clase. Las interacciones se pueden utilizar para visualizar, especificar, construir y documentar la semántica de una clase. Por ejemplo, para comprender el significado de la clase AgenteTrazadoRayos, podríamos crear interacciones que mostrasen cómo colaboran entre sí los atributos de la clase (y con los objetos globales y con los parámetros definidos en las operaciones de la clase).

Los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los casos de uso se discuten en el Capítulo 17; el modelado de la realización de un caso de uso se discuten en el Capítulo 28; los clasificadores se discuten en el Capítulo 9.

Nota: Una interacción también puede aparecer en la representación de un componente, un nodo o un caso de uso, cada uno de los cuales es, en realidad, un tipo de clasificador en UML. En el contexto de un caso de uso, una interacción representa un escenario que, a su vez, representa un flujo particular de la acción asociada al caso de uso.

Objetos y roles

Los objetos que participan en una interacción son o bien elementos concretos o bien elementos prototípicos. Como elemento concreto, un objeto representa algo del mundo real. Por ejemplo, p, una instancia de la clase Persona, podría denotar a una persona particular. Alternativamente, como elemento prototípico, p podría representar cualquier instancia de Persona.

Nota: En una colaboración, los participantes son normalmente elementos prototípicos que desempeñan roles particulares, no objetos específicos del mundo real, aunque a veces es útil describir colaboraciones entre objetos particulares.

Las clases abstractas se discuten en el Capítulo 4; las interfaces se discuten en el Capítulo 11.

Las instancias se discuten en el Capítulo 13; los diagramas de objetos se discuten en el Capítulo 14.

Las asociaciones se discuten en los Capítulos 5 y 10; los conectores y los roles se discuten en el Capítulo 15.

En el contexto de una interacción podemos encontrar instancias de clases, componentes, nodos y casos de uso. Aunque las clases abstractas y las interfaces, por definición, no pueden tener instancias directas, podemos representar instancias de estos elementos en una interacción. Tales instancias no representarán instancias directas de la clase abstracta o de la interfaz, pero pueden representar, respectivamente, instancias indirectas (o prototípicas) de cualquier clase hija concreta de la clase abstracta o de alguna clase concreta que realice la interfaz.

Un diagrama de objetos se puede ver como una representación del aspecto estático de una interacción, que configura el escenario para la interacción al especificar todos los objetos que colaboran. Una interacción va más allá, al introducir una secuencia dinámica de mensajes que pueden fluir a través de los enlaces que conectan a esos objetos.

Enlaces y conectores

Un enlace es una conexión semántica entre objetos. En general, un enlace es una instancia de una asociación. Como se muestra en la Figura 16.2, siempre que una clase tenga una asociación con otra clase, puede existir un enlace entre las instancias de las dos clases; siempre que haya un enlace entre dos objetos, un objeto puede enviar un mensaje al otro.

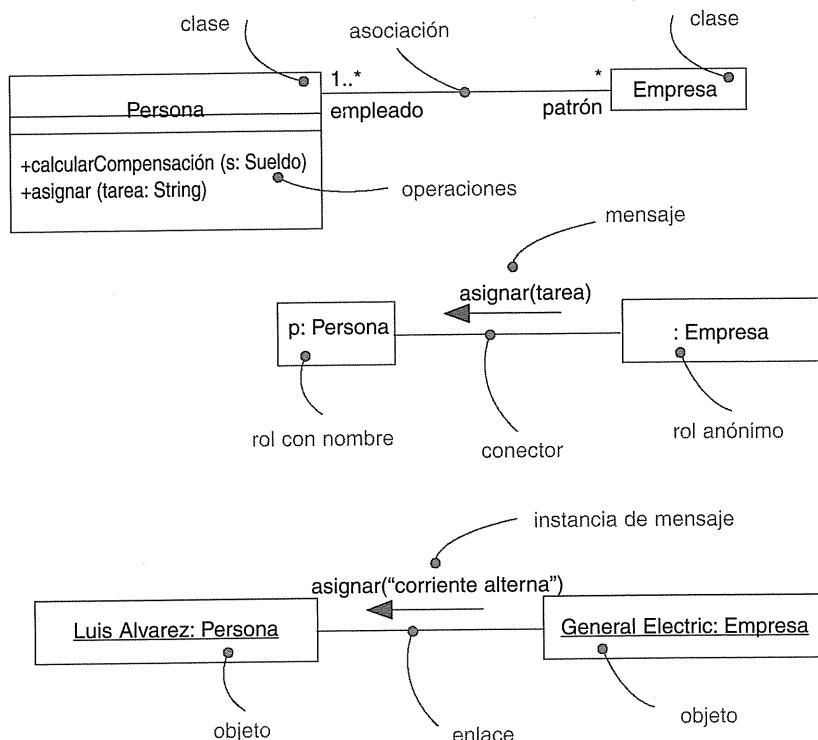


Figura 16.2: Asociaciones, enlaces y conectores.

Un enlace especifica un camino a lo largo del cual un objeto puede enviar un mensaje a otro objeto (o a sí mismo). La mayoría de las veces es suficiente con especificar que existe ese camino. Si es necesario precisar más acerca de cómo existe ese camino, se puede adornar el extremo apropiado del enlace con alguno de los siguientes estereotipos estándar.

- **association** Especifica que el objeto correspondiente es visible por existir una asociación.
- **self** Especifica que el objeto correspondiente es visible porque es el invocador de la operación.
- **global** Especifica que el objeto correspondiente es visible porque su ámbito contiene al actual.
- **local** Especifica que el objeto correspondiente es visible porque su ámbito es local.
- **parameter** Especifica que el objeto correspondiente es visible porque es un parámetro.

Los roles, los conectores y la estructura interna se discuten en el Capítulo 15; las colaboraciones se discuten en el Capítulo 28.

Nota: Como instancia de una asociación, un enlace se puede representar con la mayoría de los adornos propios de las asociaciones, tales como el nombre, nombres de roles, navegación y agregación. Sin embargo, la multiplicidad no se aplica a los enlaces, porque éstos son instancias de una asociación.

En la mayoría de los modelos, estaremos más interesados en objetos prototípicos y enlaces en un contexto que en objetos y enlaces particulares. Un objeto prototípico se denomina *rol*; un enlace prototípico se denomina *conector*; el contexto es una colaboración o la estructura interna de un clasificador. La multiplicidad de los roles y sus conectores se define en relación con el contexto que los contiene. Por ejemplo, una multiplicidad de 1 sobre un rol significa que un objeto representa el rol para cada objeto que representa el contexto. Una colaboración o una estructura interna puede utilizarse muchas veces, al igual que la declaración de una clase. En cada uso, el contexto, los roles y los conectores se ligan a un conjunto diferente de objetos y enlaces.

La Figura 16.2 muestra un ejemplo. La parte superior de la figura muestra un diagrama de clases que declara las clases Persona y Empresa y la asociación muchos-a-muchos empleado-patrón entre ellas. La parte media muestra el contenido de una colaboración AsignarTrabajo que asigna un empleado a un trabajo. Esta colaboración tiene dos roles y un conector entre ellos. La parte inferior muestra una instancia de esta colaboración, en la que hay objetos y enlaces ligados a los roles y los conectores. Un mensaje concreto en la parte inferior representa a la declaración prototípica del mensaje en la colaboración.

Los diagramas de objetos se discuten en el Capítulo 14.

Las operaciones se discuten en los Capítulos 4 y 9; los eventos se discuten en el Capítulo 21; las instancias se discuten en el Capítulo 13.

Mensajes

Supongamos que disponemos de un conjunto de objetos y un conjunto de enlaces que los conectan. Si esto es todo, entonces tenemos un modelo completamente estático que puede representarse mediante un diagrama de objetos. Los diagramas de objetos modelan el estado de una sociedad de objetos en un momento dado y son útiles cuando se quiere visualizar, especificar, construir o documentar una estructura de objetos estática.

Supongamos que queremos modelar los cambios de estado en una sociedad de objetos a lo largo de un período de tiempo. Podemos pensar en esto como

el rodaje de una película sobre un conjunto de objetos, donde los fotogramas representan momentos sucesivos en la vida de los objetos. Si estos objetos no son totalmente pasivos, se verá cómo se pasan mensajes los unos a los otros, cómo se envían eventos y cómo invocan operaciones. Además, en cada fotograma, se puede mostrar explícitamente el estado actual y el rol de cada instancia individual.

Un mensaje es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad. La recepción de una instancia de un mensaje puede considerarse una ocurrencia de un evento. (Una *ocurrencia* es el nombre de UML para la instancia de un evento).

Cuando se pasa un mensaje, su recepción produce normalmente una acción. Una acción puede producir un cambio en el estado del destinatario y en los objetos accesibles desde él.

En UML, se pueden modelar varios tipos de acciones.

Las operaciones se discuten en los Capítulos 4 y 9; las señales se discuten en el Capítulo 21.

La creación y la destrucción se representan con estereotipos, que se discuten en el Capítulo 6; la distinción entre los mensajes síncronos y asíncronos es relevante sobre todo en el contexto de la concurrencia, como se discute en el Capítulo 23.

- Llamada Invoca una operación sobre un objeto; un objeto puede enviarse un mensaje a sí mismo, lo que da lugar a la invocación local de una operación.
- Retorno Devuelve un valor al invocador.
- Envío Envía una señal a un objeto.
- Creación Crea un objeto.
- Destrucción Destruye un objeto; un objeto puede “suicidarse” al destruirse a sí mismo.

Nota: En UML también se pueden modelar acciones complejas. Además de los cinco tipos básicos de acciones listados anteriormente, se pueden incluir acciones sobre objetos individuales. UML no especifica la sintaxis ni la semántica de tales acciones; se espera que las herramientas proporcionen varios lenguajes de acciones o utilicen la sintaxis de los lenguajes de programación.

UML permite distinguir visualmente los diferentes tipos de mensajes, como se muestra en la Figura 16.3.

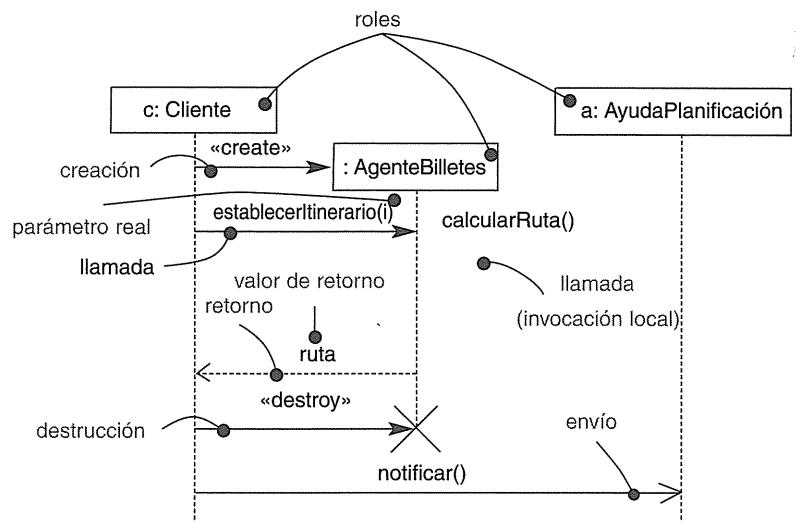


Figura 16.3: Mensajes.

Las clases se discuten en los Capítulos 4 y 9.

El tipo más común de mensaje que se modelará será la llamada, en la cual un objeto invoca una operación de otro objeto (o de él mismo). Un objeto no puede invocar cualquier operación al azar. Si un objeto, como *c* en el anterior ejemplo, invoca la operación *establecerItinerario* sobre una instancia de la clase *AgenteBillete*, la operación *establecerItinerario* no sólo debe estar definida en la clase *AgenteBillete* (o sea, debe estar declarada en la clase *AgenteBillete* o en una de sus clases padres), sino que también debe ser visible al objeto invocador *c*.

Nota: Los lenguajes como C++ tienen comprobación estática de tipos (aunque sean polimórficos), lo que significa que la validez de una llamada se comprueba en tiempo de compilación. Los lenguajes como Smalltalk, sin embargo, tienen comprobación dinámica de tipos, lo que significa que no se puede determinar la validez del envío de un mensaje a un objeto hasta el momento de su ejecución. En UML, un modelo bien formado puede, en general, ser comprobado estáticamente por una herramienta, ya que el desarrollador normalmente conoce el propósito de la operación en tiempo de modelado.

Es posible proporcionar parámetros reales al mensaje que se envía cuando un objeto invoca una operación o envía una señal a otro objeto. Análogamente, cuando un objeto devuelve el control a otro, también es posible representar el valor de retorno.

Las interfaces se discuten en el Capítulo 11.

Los mensajes también pueden corresponderse con el envío de señales. Una señal es un valor objeto que se comunica de manera asíncrona a un objeto destinatario. Después de enviar la señal, el objeto que la envió continúa su propia ejecución. Cuando el objeto destinatario recibe la señal, él decide qué hacer con ella. Normalmente las señales disparan transiciones en la máquina de estados del receptor. Disparar una transición hace que el destinatario ejecute acciones y cambie a un nuevo estado. En un sistema con paso asíncrono de mensajes, los objetos que se comunican se ejecutan de manera concurrente e independiente. Ambos comparten valores sólo a través del paso de mensajes, así que no hay peligro de conflicto en la memoria compartida.

Nota: También se puede calificar una operación por la clase o la interfaz en la que se declara. Por ejemplo, la invocación de la operación matricular sobre una instancia de Estudiante podría invocar polimórficamente cualquier operación que coincidiera con ese nombre en la jerarquía de clases de Estudiante; la invocación de IMiembro::matricular llamaría a la operación especificada en la interfaz IMiembro (y realizada por alguna clase apropiada, también en la jerarquía de clases de Estudiante).

Secuenciación

Los procesos e hilos se discuten en el Capítulo 23.

Cuando un objeto envía un mensaje a otro (delegando alguna acción en el receptor), el objeto receptor puede, a su vez, enviar un mensaje a otro objeto, el cual puede enviar un mensaje a otro objeto diferente, y así sucesivamente. Este flujo de mensajes forma una secuencia. Cualquier secuencia ha de tener un comienzo; el inicio de cada secuencia tiene su origen en algún proceso o hilo. Además, una secuencia continuará mientras exista el proceso o hilo que la contiene. Un sistema de funcionamiento continuo, como el que podemos encontrar en el control de dispositivos de tiempo real, continuará ejecutándose mientras el nodo en el que se ejecuta esté en funcionamiento.

Los sistemas se discuten en el Capítulo 32.

Cada proceso e hilo dentro de un sistema definen flujos de control separados, y dentro de cada flujo los mensajes se ordenan en secuencia conforme se suceden en el tiempo. Para visualizar mejor la secuencia de mensajes, se puede expresar

explícitamente la posición de un mensaje con relación al inicio de la secuencia, precediéndolo de un número de secuencia, separado por dos puntos.

Un diagrama de comunicación muestra el flujo de mensajes entre roles dentro de una colaboración. Los mensajes fluyen a través de las conexiones de la colaboración, como se ve en la Figura 16.4.

Lo más frecuente es especificar un flujo de control procedimental o anidado, representado con una punta de flecha rellena, como se muestra en la Figura 16.4. En este caso, el mensaje encontrarEn ha sido especificado como el primer mensaje anidado en el segundo mensaje de la secuencia (2.1).

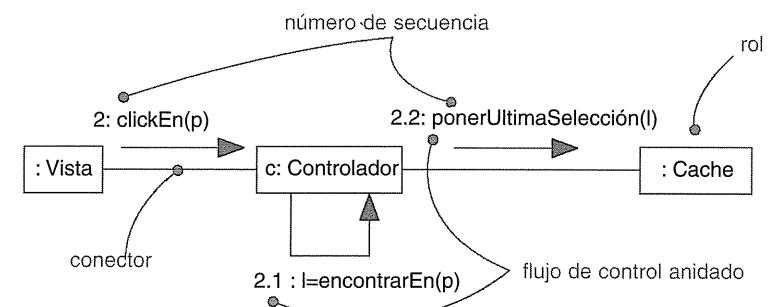


Figura 16.4: Secuencia procedural.

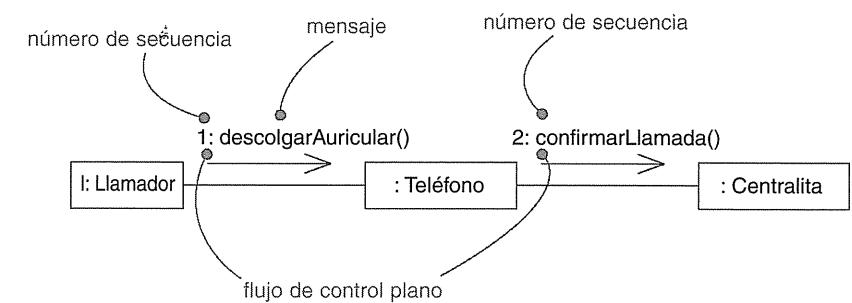


Figura 16.5: Secuencia plana.

Como muestra la Figura 16.5, también es posible, aunque menos frecuente, especificar un flujo de control plano sin anidamiento, representado con una punta de flecha en pico, para modelar la progresión no procedimental del control paso a paso. En este caso, el mensaje confirmarLlamada se ha especificado como el segundo mensaje de la secuencia.

Nota: La distinción entre las secuencias asíncronas y las procedimentales es importante en el mundo actual de la computación concurrente. Para mostrar el comportamiento global de un sistema de objetos concurrentes, hay que utilizar paso de mensajes asíncronos. Éste es el caso más general. Cuando el objeto invocador puede hacer una solicitud y esperar la respuesta, debe utilizarse el flujo de control procedimental. El flujo de control procedimental es conocido de los lenguajes de programación tradicionales, pero hay que tener presente que una serie de invocaciones a procedimientos anidadas puede dar como resultado una pila de objetos bloqueados que temporalmente serán incapaces de hacer nada, así que no es muy útil si éstos representan servidores o recursos compartidos.

Los procesos y los hilos se discuten en el Capítulo 23; también se pueden especificar flujos de control asíncronos, que se representan con una flecha con media punta, como se discute en el Capítulo 23.

Cuando se modelan interacciones que involucran varios flujos de control, es especialmente importante identificar el proceso o el hilo que envió un mensaje particular. En UML se puede distinguir un flujo de control de otro escribiendo el número de secuencia de un mensaje precedido del nombre del proceso o del hilo que es el origen de la secuencia. Por ejemplo, la expresión

D5 : abrirEscotilla(3)

especifica que la operación `abrirEscotilla` se envía (con el parámetro real 3) como el quinto mensaje de la secuencia cuya raíz es el proceso o el hilo denominado D.

No sólo se pueden mostrar los parámetros reales enviados junto a una operación o una señal en el contexto de una interacción, sino que también se pueden mostrar los valores de retorno de una función. Como se observa en la siguiente expresión, el valor `p` es devuelto por la operación `buscar`, enviada con el parámetro real “Alicia”. Se trata de una secuencia anidada; la operación es enviada como el segundo mensaje anidado en el tercer mensaje anidado en el primer mensaje de la secuencia. En el mismo diagrama, `p` puede usarse entonces como parámetro real en otros mensajes.

1.3.2 : p := buscar("Alicia")

La interacción, la ramificación y los mensajes con guardas se discuten en el Capítulo 19; las marcas de tiempo se discuten en el Capítulo 23; los estereotipos y las restricciones se discuten en el Capítulo 6.

Nota: En UML también se pueden modelar formas de secuenciación más complejas, tales como iteraciones, bifurcaciones y mensajes con guardas. Además, para modelar restricciones de tiempo como las que aparecen en los sistemas de tiempo real, se pueden asociar marcas de tiempo a una secuencia. Otras formas menos frecuentes de envíos de mensajes, tales como la sincronización sin espera (*balking*) y la sincronización con límite de tiempo (*time out*), pueden modelarse definiendo un estereotipo de mensaje apropiado.

Creación, modificación y destrucción

La mayoría de las veces, los objetos que participan en una interacción existen durante todo el tiempo que dura la interacción. Sin embargo, algunas interacciones pueden conllevar la creación (mensaje `create`) y destrucción (mensaje `destroy`) de objetos. Esto también se aplica a los enlaces: las relaciones entre objetos pueden aparecer y desaparecer. Para especificar si un objeto o un enlace aparece y/o desaparece durante una interacción, se puede asociar una nota a su rol dentro del diagrama de comunicación.

Las líneas de vida se discuten en el Capítulo 19.

A lo largo de una interacción, normalmente un objeto cambia los valores de sus atributos, su estado o sus roles. Podemos representar la modificación de un objeto en un diagrama de secuencia mostrando el estado o los valores sobre su línea de vida.

En un diagrama de secuencia se representan de manera explícita la vida, la creación y la destrucción de objetos o roles a través de la longitud vertical de sus líneas de vida. Dentro de un diagrama de comunicación, la creación y la destrucción deben indicarse con notas. Los diagramas de secuencia deben utilizarse si es importante mostrar las vidas de los objetos.

Representación

Cuando se modela una interacción, normalmente se incluirán tanto roles (cada uno representando a objetos que aparecen en una instancia de la interacción) como mensajes (cada uno representando una comunicación entre objetos, que desencadena alguna acción).

Los diagramas de interacción se discuten en el Capítulo 19.

Los roles y los mensajes implicados en una interacción se pueden visualizar de dos formas: destacando la ordenación temporal de sus mensajes, o destacando la organización estructural de los roles que envían y reciben los mensajes. En UML, el primer tipo de representación se llama diagrama de secuencia; el segundo tipo de representación se llama diagrama de comunicación. Tanto los diagramas de secuencia como los de comunicación son tipos de diagramas de interacción. (UML también tiene un tipo de diagrama de interacción más especializado llamado *diagrama de tiempo*, que muestra los momentos exactos en los que los roles se intercambian los mensajes. Este diagrama no se trata en este libro. Para más información véase el *Manual de Referencia de UML*.)

Los diagramas de secuencia y de comunicación son similares, lo que significa que podemos partir de uno de ellos y transformarlo en el otro, aunque a veces muestran información diferente, así que puede no ser tan útil pasar de uno a otro continuamente. Hay algunas diferencias visuales. Por una parte, los diagramas de secuencia permiten modelar la línea de vida de un objeto. La línea de vida de un objeto representa la existencia de un objeto durante un período de tiempo, posiblemente cubriendo la creación y destrucción del objeto. Por otra parte, los diagramas de comunicación permiten modelar los enlaces estructurales que pueden existir entre los objetos de la interacción.

Técnicas comunes de modelado

Modelado de un flujo de control

Los casos de uso se discuten en el Capítulo 17; los patrones y los frameworks se discuten en el Capítulo 29; las clases y las operaciones se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27;

La mayoría de las veces, las interacciones se utilizan con el propósito de modelar el flujo de control que caracteriza el comportamiento de un sistema, incluyendo casos de uso, patrones, mecanismos y *frameworks*, o el comportamiento de una clase o una operación individual. Mientras que las clases, las interfaces, los componentes, los nodos y sus relaciones modelan los aspectos estáticos del sistema, las interacciones modelan los aspectos dinámicos.

Cuando se modela una interacción, lo que se hace esencialmente es construir una historia de las acciones que tienen lugar entre un conjunto de objetos. Las técnicas como las tarjetas CRC son particularmente útiles para ayudar a descubrir y pensar sobre tales interacciones.

Para modelar un flujo de control:

- Hay que establecer el contexto de la interacción, si es el sistema global, una clase o una operación individual.

también se pueden modelar los aspectos dinámicos de un sistema utilizando máquinas de estados, como se discute en el Capítulo 22.

- Hay que establecer el escenario para la interacción, identificando qué objetos juegan un rol; se deben establecer sus propiedades iniciales, incluyendo los valores de sus atributos, estado y rol. Hay que dar nombre a los roles.
- Si el modelo destaca la organización estructural de esos objetos, hay que identificar los enlaces que los conectan y que sean relevantes para los trayectos de comunicación que tienen lugar en la interacción. Si es necesario, hay que especificar la naturaleza de los enlaces con los estereotipos estándar y las restricciones de UML.
- Hay que especificar los mensajes que pasan de un objeto a otro mediante una ordenación temporal. Si es necesario, hay que distinguir los diferentes tipos de mensajes; se deben incluir parámetros y valores de retorno para expresar los detalles necesarios de la interacción.
- Para expresar los detalles de la interacción que sean necesarios, habrá que adornar cada objeto con su estado y rol, siempre que sea preciso.

Por ejemplo, la Figura 16.6 representa un conjunto de objetos que interactúan en el contexto de un mecanismo de publicación y suscripción (una instancia del patrón de diseño *observador*). Esta figura incluye tres roles: r (un *ResponsablePrecioStock*), s1 y s2 (ambos instancias de *SuscriptorPrecioStock*). Esta figura es un ejemplo de un diagrama de secuencia, que resalta la ordenación temporal de los mensajes.

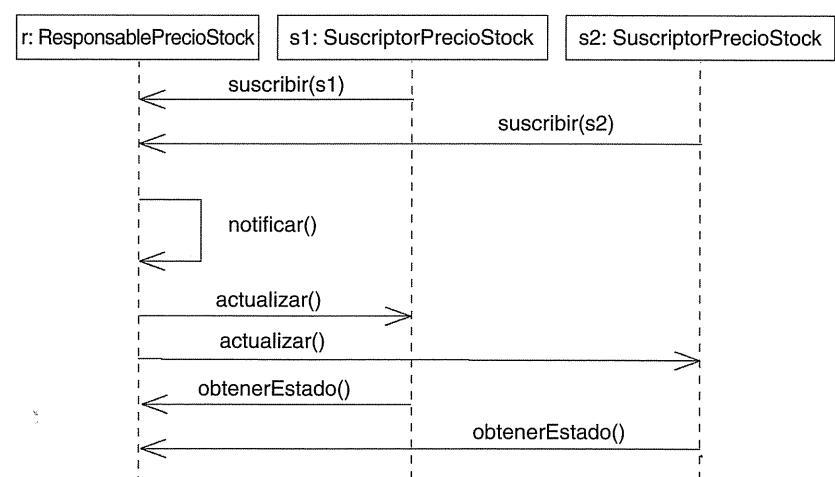


Figura 16.6: Flujo de control por tiempo.

Los diagramas de comunicación se discuten en el Capítulo 19.

La Figura 16.7 es semánticamente equivalente a la figura anterior, pero ha sido dibujada como un diagrama de comunicación, el cual destaca la organización estructural de los objetos. Esta figura representa el mismo flujo de control, pero también muestra los enlaces entre esos objetos.

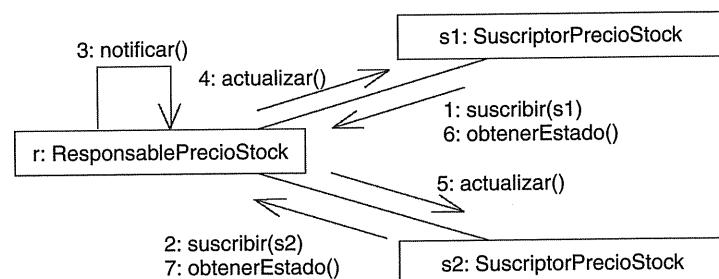


Figura 16.7: Flujo de control por organización.

Sugerencias y consejos

Cuando se modelan interacciones en UML, hay que recordar que cada interacción representa el aspecto dinámico de una sociedad de objetos. Una interacción bien estructurada:

- Es sencilla y debe incluir sólo aquellos objetos que colaboran para llevar a cabo algún comportamiento mayor que la suma de los comportamientos de esos elementos.
- Tiene un contexto claro y puede representar la interacción de objetos en el contexto de una operación, una clase o el sistema completo.
- Es eficiente y debe llevar a cabo su comportamiento con un equilibrio óptimo de tiempo y recursos.
- Es adaptable y los elementos más proclives a cambiar deberían aislar-
se para que puedan ser fácilmente modificados.
- Es comprensible y debería ser sencilla, y no debe incluir trucos, efectos laterales ocultos ni tener una semántica confusa.

Cuando se dibuje una interacción en UML:

- Hay que elegir el aspecto a destacar en la interacción. Se puede destaca-
r la ordenación temporal de los mensajes o la secuencia de los men-

sajes en el contexto de alguna organización estructural de objetos. No se pueden hacer ambas cosas al mismo tiempo.

- Hay que tener en cuenta que los eventos en subsecuencias separadas sólo están parcialmente ordenados. Cada subsecuencia está ordenada, pero los tiempos relativos de los eventos en diferentes subsecuencias no están fijados.
- Hay que mostrar sólo aquellas propiedades de cada objeto (valores de atributos, rol y estado) que sean importantes para comprender la interacción en su contexto.
- Hay que mostrar sólo aquellas propiedades de los mensajes (parámetros, semántica de concurrencia, valor de retorno) que sean importantes para comprender la interacción en su contexto.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 17

CASOS DE USO

En este capítulo

- Casos de uso, actores, relaciones de inclusión y extensión.
- Modelado del comportamiento de un elemento.
- Realización de casos de uso con colaboraciones.

Ningún sistema se encuentra aislado. Cualquier sistema interesante interactúa con actores humanos o mecánicos que lo utilizan con algún objetivo y que esperan que el sistema funcione de forma predecible. Un caso de uso especifica el comportamiento de un sistema o de una parte de éste, y es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor.

Los casos de uso se emplean para capturar el comportamiento deseado del sistema en desarrollo, sin tener que especificar cómo se implementa ese comportamiento. Los casos de uso proporcionan un medio para que los desarrolladores, los usuarios finales del sistema y los expertos del dominio lleguen a una comprensión común del sistema. Además, los casos de uso ayudan a validar la arquitectura y a verificar el sistema mientras evoluciona a lo largo del desarrollo. Conforme se desarrolla el sistema, los casos de uso son realizados por colaboraciones, cuyos elementos cooperan para llevar a cabo cada caso de uso.

Los casos de uso bien estructurados denotan sólo comportamientos esenciales del sistema o de un subsistema, y nunca deben ser excesivamente genéricos ni demasiado específicos.

Introducción

Una casa bien diseñada es algo más que un montón de paredes, puestas juntas para sostener en alto un techo que proteja de las inclemencias del tiempo. Cuando una persona discute con un arquitecto el diseño de una casa para su familia, tiene muy en cuenta cómo se utilizará la casa. Si le gusta organizar fiestas para los amigos, debería considerar el tránsito de personas por el salón, con el fin de facilitar que los invitados puedan conversar y no existan puntos donde se produzcan aglomeraciones. También procurará facilitar la preparación de comidas para la familia, con un diseño de la cocina que permita una colocación eficiente de la despensa y los electrodomésticos. Incluso podría considerar cómo llegar del garaje a la cocina para descargar la compra, lo cual afectará a la forma de conectar las distintas habitaciones. Si se trata de una familia grande, habrá que pensar en el uso del cuarto de baño. Si en el diseño se establece el número y posición de los cuartos de baño lo antes posible, se reducirá enormemente el riesgo de cuellos de botella por las mañanas cuando la familia se dirige al trabajo o al colegio. Si hay hijos adolescentes, esta cuestión es especialmente importante, porque emocionalmente puede suponer un gran trastorno.

El análisis de cómo una familia utilizará una casa es un ejemplo de análisis basado en casos de uso. Se consideran las diferentes formas en que se utilizará la casa, y estos casos de uso guían la arquitectura. Muchas familias tendrán las mismas categorías de casos de uso (las casas se utilizan para comer, dormir, criar a los niños y guardar recuerdos). Pero cada familia también tendrá sus propios casos de uso, que podrán ser especiales o variaciones de los básicos. Las necesidades de una familia grande, por ejemplo, son diferentes de las de un adulto soltero que acaba de finalizar sus estudios. Estas variaciones son las que mayor impacto tienen en la forma que finalmente tendrá la casa.

Un factor clave al definir casos de uso como los mencionados es que no se especifica cómo se implementan. Por ejemplo, se puede especificar cómo debería comportarse un cajero automático enunciando mediante casos de uso cómo interactúan los usuarios con el sistema; pero no se necesita saber nada del interior del cajero. Los casos de uso especifican un comportamiento deseado tal y como se verá desde afuera, pero no imponen cómo se llevará a cabo internamente ese comportamiento. Lo más importante es que permiten que los usuarios finales y los expertos del dominio se comuniquen con los desarrolladores (quienes construyen sistemas que satisfacen sus requisitos) sin quedarse atascados en los detalles. Estos detalles llegarán, pero los casos de uso permiten centrarse en las cuestiones más importantes para el usuario final.

Las interacciones se discuten en el Capítulo 16; los requisitos se discuten en el Capítulo 6.

A nivel del sistema, un caso de uso describe un conjunto de secuencias, donde cada secuencia representa la interacción de los elementos externos al sistema (sus actores) con el propio sistema (y con sus abstracciones claves). En realidad, estos comportamientos son funciones a nivel del sistema que se utilizan durante la captura de requisitos y el análisis para visualizar, especificar, construir y documentar el comportamiento esperado del sistema. Un caso de uso representa un requisito funcional del sistema global. Por ejemplo, un caso de uso fundamental en un banco es el procesamiento de préstamos.

Un caso de uso involucra la interacción de actores y el sistema u otros sujetos. Un actor representa un conjunto coherente de roles que juegan los usuarios de los casos de uso al interactuar con éstos. Los actores pueden ser personas o pueden ser sistemas automáticos. Por ejemplo, en el modelado de un banco, el procesamiento de un préstamo implica, entre otras cosas, la interacción entre un cliente y un responsable de préstamos.

Un caso de uso puede tener variantes. En cualquier sistema interesante, se pueden encontrar casos de uso que son versiones especializadas de otros casos de uso, casos de uso incluidos como parte de otros, y casos de uso que extienden el comportamiento de otros casos de uso básicos. Se puede factorizar el comportamiento común y reutilizable de un conjunto de casos de uso organizándolos según estos tres tipos de relaciones. Por ejemplo, cuando se modela un banco aparecen muchas variaciones del caso de uso básico de procesar un préstamo, tales como las diferencias entre procesar una gran hipoteca frente a un pequeño préstamo comercial. En cada caso, sin embargo, estos casos de uso comparten algo de comportamiento, como el caso de uso de aprobar el préstamo para ese cliente, un comportamiento que es parte del procesamiento de cualquier tipo de préstamo.

Un caso de uso realiza cierto trabajo cuyo efecto es tangible. Desde la perspectiva de un actor determinado, un caso de uso produce algo de valor para algún actor, como el cálculo de un resultado, la generación de un nuevo objeto, o el cambio del estado de otro objeto. Por ejemplo, en el modelado de un banco, el procesamiento de un préstamo produce un préstamo aceptado, que se concreta en una cantidad de dinero entregada al cliente.

Los subsistemas se discuten en el Capítulo 32; las clases se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11.

Los casos de uso se pueden aplicar al sistema completo. También se pueden aplicar a partes del sistema, incluyendo subsistemas e incluso clases e interfaces individuales. En cada caso, estos casos de uso no sólo representan el comportamiento esperado de estos elementos, sino que también pueden utilizarse como la base para establecer casos de prueba para esos elementos mientras evolucionan durante el desarrollo del sistema. Los casos de uso, aplicados a los subsistemas, son una fuente excelente de pruebas de regresión; los casos de uso

aplicados al sistema completo son una fuente excelente de pruebas del sistema y de integración. UML proporciona una representación gráfica de un caso de uso y un actor, como se muestra en la Figura 17.1. Esta notación permite visualizar un caso de uso independientemente de su realización y en un contexto con otros casos de uso.

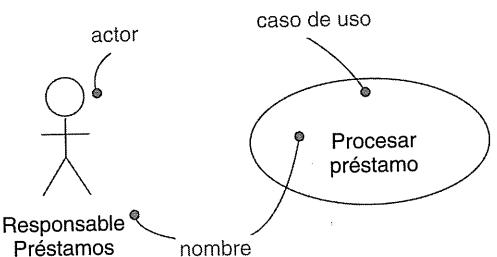


Figura 17.1: Actores y caso de uso.

Términos y conceptos

La notación de los casos de uso es similar a la de las colaboraciones, como se discute en el Capítulo 28.

Un *caso de uso* es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor. Gráficamente, un caso de uso se representa como una elipse.

Sujeto

El *sujeto* es una clase descrita por un conjunto de casos de uso. Normalmente la clase es un sistema o un subsistema. El caso de uso representa aspectos del comportamiento de la clase. Los actores representan aspectos de otras clases que interactúan con el sujeto. Uniéndolos todos, los casos de uso describen el comportamiento completo del sujeto.

Nombres

Cada caso de uso debe tener un nombre que lo distinga de otros casos de uso. Un *nombre* es una cadena de texto. Ese nombre solo se llama *nombre simple*; un *nombre calificado* consta del nombre del caso de uso precedido del nombre del paquete en el que se encuentra. Normalmente, un caso de uso se dibuja mostrando sólo su nombre, como se ve en la Figura 17.2.

El nombre de un caso de uso debe ser único dentro del paquete que lo contiene, como se discuten en el Capítulo 12.

Los artefactos se discuten en el Capítulo 26.

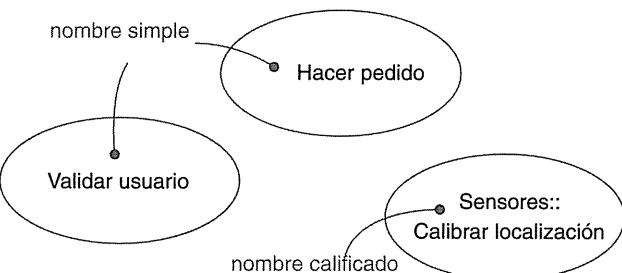


Figura 17.2: Nombres simple y calificado

Nota: El nombre de un caso de uso puede constar de texto con cualquier número de letras, números y la mayoría de los signos de puntuación (excepto signos como los dos puntos, utilizados para separar el nombre de un caso de uso del nombre del paquete que lo contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de los casos de uso son expresiones verbales que describen algún comportamiento del vocabulario del sistema que se está modelando.

Casos de uso y actores

Un actor representa un conjunto coherente de roles que los usuarios de los casos de uso representan al interactuar con éstos. Normalmente, un actor representa un rol que es desempeñado por una persona, un dispositivo hardware o incluso otro sistema al interactuar con nuestro sistema. Por ejemplo, si una persona trabaja para un banco, podría ser un *ResponsablePrestamos*. Si tiene sus cuentas personales en ese banco, está desempeñando también el rol de *Cliente*. Una instancia de un actor, por lo tanto, representa una interacción individual con el sistema de una forma específica. Aunque se utilizan actores en los modelos, éstos no forman parte del sistema. Están fuera de la aplicación, en el entorno que la rodea.

En un sistema en ejecución, los actores no han de existir como entidades separadas. Un objeto puede representar el papel de varios actores. Por ejemplo, una *Persona* puede ser tanto *ResponsablePrestamos* como *Cliente*.

La generalización se discute en los Capítulos 5 y 10.

Como se muestra en la Figura 17.3, los actores se representan como monigotes. Se pueden definir categorías generales de actores (como *Cliente*) y especializarlos (como *ClienteComercial*) a través de relaciones de generalización.

Los estereotipos se discuten en el Capítulo 6.

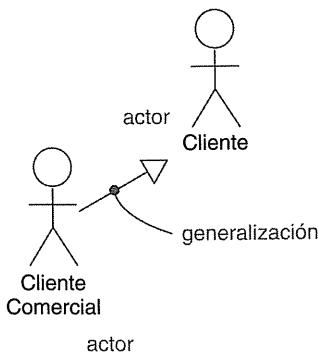


Figura 17.3: Actores.

Las relaciones de asociación se discuten en los Capítulos 5 y 10; los mensajes se discuten en el Capítulo 16.

Nota: Los mecanismos de extensibilidad de UML se pueden utilizar para estereotipar un actor, con el fin de proporcionar un icono diferente que ofrezca una señal visual más apropiada a los objetivos que se persiguen.

Los actores sólo se pueden conectar a los casos de uso a través de asociaciones. Una asociación entre un actor y un caso de uso indica que el actor y el caso de uso se comunican entre sí, y cada uno puede enviar y recibir mensajes.

Casos de uso y flujo de eventos

Un caso de uso describe *qué* hace un sistema (o un subsistema, una clase o una interfaz), pero no especifica *cómo* lo hace. Cuando se modela, es importante tener clara la separación de intereses entre las vistas externa e interna.

El comportamiento de un caso de uso se puede especificar describiendo un flujo de eventos de forma textual, lo suficientemente claro para que alguien ajeno al sistema lo entienda fácilmente. Cuando se escribe este flujo de eventos se debe incluir cómo y cuándo empieza y acaba el caso de uso, cuándo interactúa con los actores y qué objetos se intercambian, el flujo básico y los flujos alternativos del comportamiento.

Por ejemplo, en el contexto de un cajero automático, se podría describir el caso de uso *ValidarUsuario* de la siguiente forma:

Flujo de eventos principal: El caso de uso comienza cuando el sistema pide al *Cliente* un número de identificación personal (PIN). El *Cliente* puede introducir un PIN a través del teclado. El *Cliente* acepta la

entrada pulsando el botón Enter. El sistema comprueba entonces este PIN para ver si es válido. Si el PIN es válido, el sistema acepta la entrada, y así acaba el caso de uso.

Flujo de eventos excepcional: El *Cliente* puede cancelar una transacción en cualquier momento pulsando el botón Cancelar, y de esta forma reinicia el caso de uso. No se efectúa ningún cambio a la cuenta del *Cliente*.

Flujo de eventos excepcional: El *Cliente* puede borrar un PIN en cualquier momento antes de introducirlo, y volver a teclear un nuevo PIN.

Flujo de eventos excepcional: Si el *Cliente* introduce un PIN inválido, el caso de uso vuelve a empezar. Si esto ocurre tres veces en una sesión, el sistema cancela la transacción completa, lo que impide que el *Cliente* utilice el cajero durante 60 segundos.

Nota: El flujo de eventos de un caso de uso se puede especificar de muchas formas, incluyendo texto estructurado informal (como el ejemplo anterior), texto estructurado formal (con pre y postcondiciones), máquinas de estados (especialmente para los sistemas reactivos), diagramas de actividad (especialmente para los flujos de trabajo) y pseudocódigo.

Casos de uso y escenarios

Los diagramas de interacción, que incluyen los diagramas de secuencia y de colaboración, se discuten en el Capítulo 19.

Normalmente, primero se describe el flujo de eventos de un caso de uso mediante texto. Sin embargo, conforme se mejora la comprensión de los requisitos del sistema estos flujos se pueden especificar gráficamente mediante diagramas de interacción. Normalmente, se usa un diagrama de secuencia para especificar el flujo principal de un caso de uso, y se usan variaciones de ese diagrama para especificar los flujos excepcionales del caso de uso.

Conviene separar el flujo principal de los flujos alternativos, porque un caso de uso describe un conjunto de secuencias, no una única secuencia, y sería imposible expresar todos los detalles de un caso de uso no trivial en una única secuencia. Por ejemplo, en un sistema de recursos humanos, podría aparecer el caso de uso *Contratar Empleado*. Esta función básica del sistema podría tener muchas variantes. Podría contratarse a una persona de otra empresa (el escenario más frecuente); podría trasladarse una persona de un departamento a otro (algo frecuente en las compañías internacionales) o podría contratarse a un extranje-

ro (lo que conlleva sus reglas específicas). Cada una de estas variantes se puede expresar en una secuencia diferente.

Las instancias se discuten en el Capítulo 13.

Este caso de uso (Contratar Empleado), en realidad describe un conjunto de secuencias, donde cada secuencia representa un posible flujo a través de todas las variantes. Cada secuencia se denomina escenario. Un escenario es una secuencia específica de acciones que ilustra un comportamiento. Los escenarios son a los casos de uso lo que las instancias a las clases; es decir, un escenario es básicamente una instancia de un caso de uso.

Nota: Hay un efecto de expansión de los casos de uso a los escenarios. Un sistema modestamente complejo puede tener unas pocas decenas de casos de uso que capturen su comportamiento, y cada caso de uso puede expandirse en varias decenas de escenarios. Para cada caso de uso, puede haber escenarios principales (definen secuencias esenciales) y escenarios secundarios (definen secuencias alternativas).

Casos de uso y colaboraciones

Las colaboraciones se discuten en el Capítulo 28.

Un caso de uso captura el comportamiento esperado del sistema (o subsistema, clase o interfaz) que se está desarrollando, sin tener que especificar cómo se implementa ese comportamiento. Esta separación es importante porque el análisis de un sistema (que especifica un comportamiento) no debería estar influenciando, mientras sea posible, por cuestiones de implementación (que especifican cómo se lleva a cabo el comportamiento). No obstante, un caso de uso debe implementarse al fin y al cabo, y esto se hace creando una sociedad de clases y otros elementos que colaborarán para llevar a cabo el comportamiento del caso de uso. Esta sociedad de elementos, incluyendo tanto su estructura estática como la dinámica, se modela en UML como una colaboración.

La realización se discute en los Capítulos 9 y 10.

Como se muestra en la Figura 17.4, la realización de un caso de uso puede especificarse explícitamente mediante una colaboración. Pero, aunque la mayoría de las veces un caso de uso es realizado exactamente por una colaboración, no será necesario mostrar explícitamente esta relación.

Nota: Aunque no se puede visualizar esta relación explícitamente, las herramientas que se utilicen para gestionar los modelos probablemente la mantendrán.

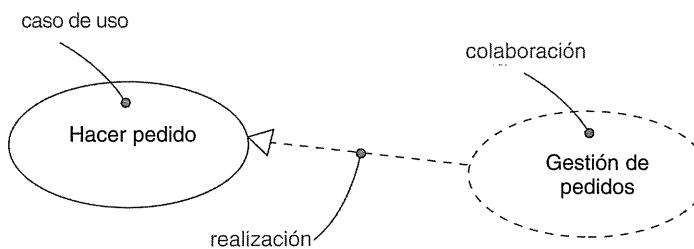


Figura 17.4: .4: Casos de uso y colaboraciones.

La arquitectura se discute en el Capítulo 2.

Nota: El objetivo de la arquitectura de un sistema es encontrar el conjunto mínimo de colaboraciones bien estructuradas que satisfacen el flujo de eventos, especificado en todos los casos de uso del sistema.

Los paquetes se discuten en el Capítulo 12.

Organización de los casos de uso

Los casos de uso pueden organizarse agrupándolos en paquetes, de la misma manera que se organizan las clases.

Los casos de uso también pueden organizarse especificando relaciones de generalización, inclusión y extensión entre ellos. Estas relaciones se utilizan para factorizar el comportamiento común (extrayendo ese comportamiento de los casos de uso en los que se incluye) y para factorizar variantes (poniendo ese comportamiento en otros casos de uso que lo extienden).

La generalización se discute en los Capítulos 5 y 10.

La generalización entre casos de uso es como la generalización entre clases. Aquí significa que el caso de uso hijo hereda el comportamiento y el significado del caso de uso padre; el hijo puede añadir o redefinir el comportamiento del padre; el hijo puede ser colocado en cualquier lugar donde aparezca el padre (tanto el padre como el hijo pueden tener instancias concretas). Por ejemplo, en un sistema bancario puede tenerse el caso de uso *Validar Usuario*, responsable de verificar la identidad del usuario. Además, podría haber dos hijos especializados de este caso de uso (*Comprobar clave* y *Examinar retina*), los cuales se comportarían como *Validar Usuario* y podrían utilizarse dondequiera que apareciera *Validar Usuario*, aunque ambos tengan su propio comportamiento (el primero comprobando una clave textual, el segundo comprobando los patrones diferenciadores en la retina del usuario). Como se muestra en la Figura 17.5, la generalización entre casos de uso se representa con una línea continua con una punta de flecha vacía, al igual que la generalización entre clases.

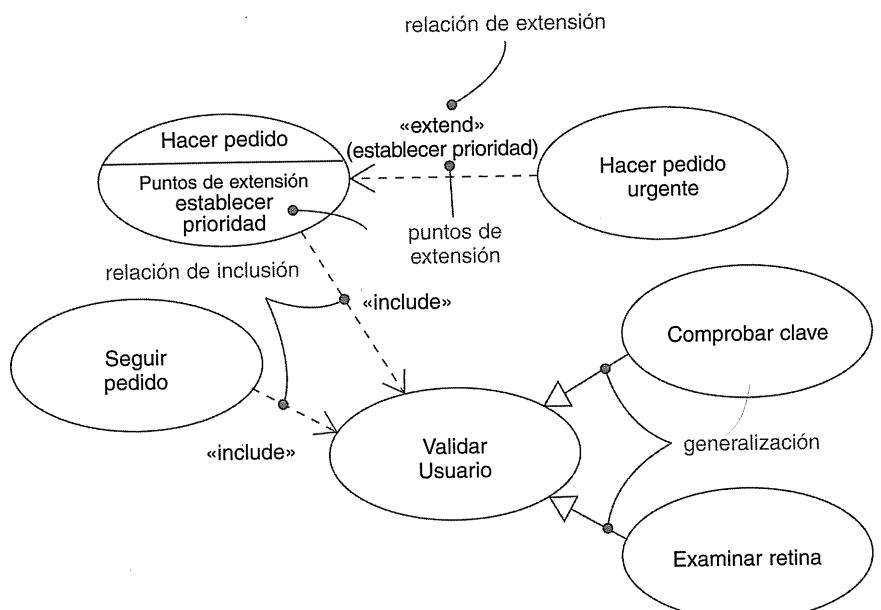


Figura 17.5: Generalización, inclusión y extensión.

Una relación de inclusión entre casos de uso significa que un caso de uso base incorpora explícitamente el comportamiento de otro caso de uso en el lugar especificado en el caso base. El caso de uso incluido nunca se encuentra aislado, sino que es instanciado sólo como parte de algún caso de uso base más amplio que lo incluye. La inclusión puede verse como que el caso de uso base toma el comportamiento del caso de uso proveedor.

La relación de inclusión se usa para evitar describir el mismo flujo de eventos repetidas veces, poniendo el comportamiento común en un caso de uso aparte (que será incluido por un caso de uso base). La relación de inclusión es básicamente un ejemplo de delegación: se toma un conjunto de responsabilidades del sistema y se capturan en un sitio (el caso de uso a incluir en otros); a continuación se permite que otras partes del sistema (otros casos de uso) incluyan la nueva agregación de responsabilidades, siempre que se necesite usar esa funcionalidad.

Una relación de inclusión se representa como una dependencia, estereotipada con `include`. Para especificar la posición en un flujo de eventos en la cual el caso de uso base incluye el comportamiento de otro caso de uso, simplemente se debe escribir `include` seguido del nombre del caso de uso que se quiere incluir, como en el siguiente flujo para Seguir pedido.

Las relaciones de dependencia se discuten en los Capítulos 5 y 10; los estereotipos se discuten en el Capítulo 6.

Seguir pedido:

- obtener y verificar el número de pedido;
- include 'Validar usuario';
- para cada parte del pedido,
- consultar el estado del pedido;
- informar del estado global al usuario.

Nota: No existe una notación predefinida en UML para expresar los escenarios de los casos de uso. La sintaxis utilizada aquí es un tipo de lenguaje natural estructurado. Varios autores han sugerido que es mejor una notación informal, porque los casos de uso no deberían verse como especificaciones rígidas que generen código automáticamente; otros han propuesto notaciones formales.

Una relación de extensión entre casos de uso significa que un caso de uso base incorpora implícitamente el comportamiento de otro caso de uso en el lugar especificado indirectamente por el caso de uso que extiende al base. El caso de uso base puede estar aislado pero, en algunas condiciones, su comportamiento puede extenderse con el comportamiento de otro caso de uso. Este caso de uso base puede extenderse sólo en ciertos puntos, llamados, como era de esperar, puntos de extensión. La extensión se puede ver como que el caso de uso que extiende incorpora su comportamiento en el caso de uso base.

Una relación de extensión se utiliza para modelar la parte de un caso de uso que el usuario puede ver como comportamiento opcional del sistema. De esta forma, se separa el comportamiento opcional del obligatorio. También se puede utilizar una relación de extensión para modelar un subflujo separado que se ejecuta sólo bajo ciertas condiciones. Por último, se puede utilizar una relación de extensión para modelar varios flujos que se pueden insertar en un punto dado, controlados por la interacción explícita con un actor. También se puede usar una relación de extensión para distinguir las partes opcionales de un sistema que se va a implementar. El significado es que el sistema puede existir con o sin las diferentes extensiones.

Las relaciones de dependencia se discuten en los Capítulos 5 y 10; los estereotipos y los comportamientos extras se discuten en el Capítulo 6.

Una relación de extensión se representa como una dependencia, estereotipada como `extend`. Los puntos de extensión del caso de uso base se pueden listar en un compartimento extra. Estos puntos de extensión sólo son etiquetas que pueden aparecer en el flujo del caso de uso base. Por ejemplo, el flujo Hacer pedido podría escribirse como sigue:

Hacer pedido:

```
include 'Validar usuario';
capturar los artículos del pedido del usuario;
establecer prioridad: punto de extensión;
enviar el pedido para ser procesado.
```

En este ejemplo, establecer prioridad es un punto de extensión. Un caso de uso puede tener más de un punto de extensión (que puede aparecer más de una vez), y éstos siempre se identifican por el nombre. En circunstancias normales, este caso de uso base se ejecutará sin importar la prioridad del pedido. Por otra parte, si se trata de una instancia de un pedido con prioridad, el flujo para este caso base se desarrollará según se ha indicado antes. Pero en el punto de extensión establecer prioridad, se ejecutará el comportamiento del caso de uso que extiende Hacer pedido urgente, y después se continuará con el flujo normal. Si hay varios puntos de extensión, el caso de uso que extiende al caso base simplemente combinará sus flujos en orden.

Nota: La organización de los casos de uso, considerando la extracción de comportamientos comunes (a través de relaciones de inclusión) y la distinción de variantes (a través de relaciones de extensión), es muy importante para la creación de un conjunto de casos de uso del sistema que sea sencillo, equilibrado y comprensible.

Otras características

Los atributos y las operaciones se discuten en el Capítulo 4; las máquinas de estados se discuten en el Capítulo 22.

Los casos de uso también son clasificadores, de forma que pueden tener operaciones y atributos que se pueden representar igual que en las clases. Se puede pensar en estos atributos como en los objetos dentro del caso de uso que son necesarios para describir el comportamiento externo. Análogamente, las operaciones se pueden ver como las acciones del sistema que son necesarias para describir un flujo de eventos. Estos objetos y operaciones pueden utilizarse en los diagramas de interacción para especificar el comportamiento del caso de uso.

Como clasificadores que son, también se pueden asociar máquinas de estados a los casos de uso. Las máquinas de estados se pueden usar como otra forma de describir el comportamiento representado por un caso de uso.

Técnicas comunes de modelado**Modelado del comportamiento de un elemento**

Los sistemas y los subsistemas se discuten en el Capítulo 32; las clases se discuten en los Capítulos 4 y 9.

La mayoría de las veces, los casos de uso se utilizan para el modelado del comportamiento de un elemento, ya sea un sistema completo, un subsistema o una clase. Cuando se modela el comportamiento de estos elementos, es importante centrarse en lo que hace el elemento, no en cómo lo hace.

Es importante aplicar de esta forma los casos de uso a los elementos por tres razones. Primera, el modelado del comportamiento de un elemento, mediante casos de uso, permite a los expertos del dominio especificar su vista externa del sistema a nivel suficiente para que los desarrolladores construyan su vista interna. Los casos de uso proporcionan un foro en el que pueden intercambiar opiniones los expertos del dominio, los usuarios finales y los desarrolladores. Segunda, los casos de uso proporcionan a los desarrolladores una forma de abordar y comprender un elemento. Un sistema, un subsistema o una clase pueden ser complejos y estar repletos de operaciones y otras partes. Cuando se especifican los casos de uso de un elemento, se ayuda a que los usuarios de ese elemento lo puedan abordar directamente, de acuerdo con el modo en el que ellos utilizarán el sistema. En ausencia de estos casos de uso, los usuarios tendrían que descubrir por su cuenta cómo usar el elemento. Los casos de uso permiten que el creador de un elemento comunique su intención sobre cómo se debería usar. Tercera, los casos de uso sirven de base para probar cada elemento conforme evoluciona durante el desarrollo. Al probar continuamente cada elemento frente a sus casos de uso, se está validando su implementación a lo largo del desarrollo. Estos casos de uso no sólo sirven como punto de partida para las pruebas de regresión, sino que cada vez que se añade un caso de uso a un elemento, hay que reconsiderar la implementación, para asegurarse de que ese elemento es flexible al cambio. Si no lo es, la arquitectura debe reorganizarse del modo adecuado.

Para modelar el comportamiento de un elemento:

- Hay que identificar los actores que interactúan con el elemento. Los actores candidatos pueden incluir grupos que requieran un cierto comportamiento para ejecutar sus tareas o que se necesiten directa o indirectamente para ejecutar las funciones del elemento.
- Hay que organizar los actores identificando tanto los roles más generales como los más especializados.
- Hay que considerar las formas más importantes que tiene cada actor de interactuar con el elemento. También deben considerarse las interacciones

que implican el cambio de estado del elemento o su entorno o que involucran una respuesta ante algún evento.

- Hay que considerar también las formas excepcionales en las que cada actor puede interactuar con el elemento.
- Hay que organizar estos comportamientos como casos de uso, utilizando las relaciones de inclusión y extensión para factorizar el comportamiento común y distinguir el comportamiento excepcional.

Por ejemplo, un sistema de venta interactuará con clientes, que efectuarán pedidos y querrán llevar un seguimiento. A su vez, el sistema enviará pedidos y facturas al cliente. Como se muestra en la Figura 17.6, el comportamiento de ese sistema se puede modelar declarando estos comportamientos como casos de uso (*Hacer pedido*, *Seguir pedido*, *Enviar pedido* y *Facturar al cliente*). El comportamiento común puede factorizarse (*Validar cliente*) y también pueden distinguirse sus variantes (*Enviar pedido parcial*). Para cada caso de uso se incluirá una especificación del comportamiento, ya sea a través de texto, una máquina de estados o interacciones.

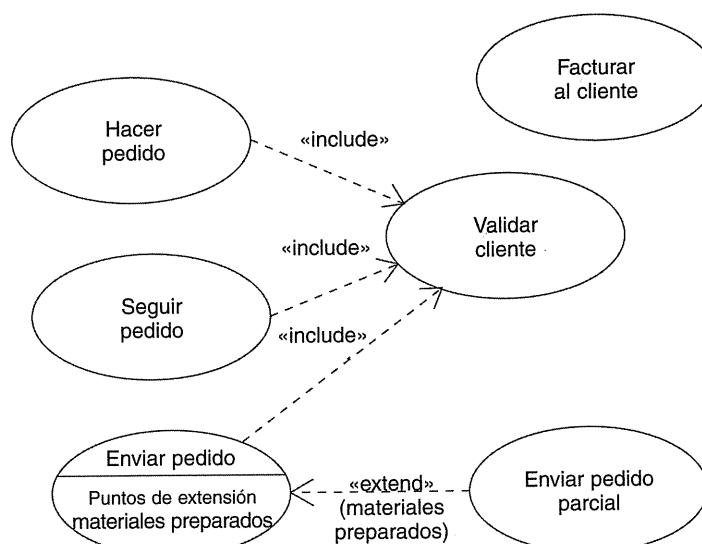


Figura 17.6: Modelado del comportamiento de un elemento.

Los paquetes se discuten en el Capítulo 12.

Conforme crecen los modelos, se descubrirá que los casos de uso tienden a juntarse en grupos relacionados conceptual y semánticamente. En UML, se pueden utilizar los paquetes para modelar estas agrupaciones de clases.

Sugerencias y consejos

Cuando se modelan los casos de uso en UML, cada caso de uso debe representar un comportamiento distinto e identificable del sistema o de una parte de éste. Un caso de uso bien estructurado:

- Asigna un nombre a un comportamiento simple, identificable y razonablemente atómico del sistema o parte del sistema.
- Factoriza el comportamiento común, incorporando ese comportamiento desde otros casos de uso que incluye.
- Factoriza las variantes, colocando ese comportamiento en otros casos de uso que lo extienden.
- Describe el flujo de eventos de forma suficientemente clara para que alguien externo al sistema lo entienda fácilmente.
- Se describe por un conjunto mínimo de escenarios que especifican la semántica normal y de las variantes del caso de uso.

Cuando se dibuje un caso de uso en UML:

- Hay que mostrar sólo aquellos casos de uso que sean importantes para comprender el comportamiento del sistema o parte del sistema en su contexto.
- Hay que mostrar sólo aquellos actores relacionados con ese caso de uso.



Capítulo 18

DIAGRAMAS DE CASOS DE USO

En este capítulo

- Modelado del contexto de un sistema.
- Modelado de los requisitos de un sistema.
- Ingeniería directa e inversa.

Los diagramas de actividades se discuten en el Capítulo 20; los diagramas de estados se discuten en el Capítulo 25; los diagramas de secuencia y de comunicación se discuten en el Capítulo 19.

Los diagramas de casos de uso son uno de los tipos de diagramas de UML que se utilizan para modelar los aspectos dinámicos de un sistema. (Los diagramas de actividades, de estados, de secuencia y de comunicación son otros cuatro tipos de diagramas de UML para modelar los aspectos dinámicos de un sistema). Los diagramas de casos de uso son importantes para modelar el comportamiento de un sistema, un subsistema o una clase. Cada uno muestra un conjunto de casos de uso, actores y sus relaciones.

Los diagramas de casos de uso se emplean para modelar la vista de casos de uso de un sistema. La mayoría de las veces, esto implica modelar el contexto del sistema, subsistema o clase, o el modelado de los requisitos de comportamiento de esos elementos.

Los diagramas de casos de uso son importantes para visualizar, especificar y documentar el comportamiento de un elemento. Estos diagramas facilitan que los sistemas, subsistemas y clases sean abordables y comprensibles, al presentar una vista externa de cómo pueden utilizarse estos elementos en un contexto dado. Los diagramas de casos de uso también son importantes para probar sistemas ejecutables a través de ingeniería directa y para comprender sistemas ejecutables a través de ingeniería inversa.

Introducción

Supongamos que alguien nos da una caja. En un lado hay algunos botones y una pequeña pantalla de cristal líquido. Aparte de esto, no viene ninguna descripción

con la caja; ni siquiera disponemos de ningún indicio acerca de cómo usarla. Podríamos pulsar aleatoriamente los botones y ver qué ocurre; pero nos veríamos obligados a imaginar qué hace la caja o cómo utilizarla correctamente, a menos que dedicásemos mucho tiempo a hacer pruebas de ensayo y error.

Los sistemas con gran cantidad de software pueden ser parecidos. A un usuario se le podría dar una aplicación y pedirle que la utilizara. Si la aplicación sigue las convenciones habituales del sistema operativo al que el usuario está acostumbrado, podría ser capaz de hacer algo útil después de un rato, pero de esta forma nunca llegaría a entender su comportamiento más complejo y útil. Análogamente, a un desarrollador se le podría dar una aplicación ya existente o un conjunto de componentes y decirle que los use. El desarrollador se vería presionado ante la necesidad de conocer cómo usar estos elementos hasta que se formara un modelo conceptual para su uso.

Con UML, los diagramas de casos de uso se emplean para visualizar el comportamiento de un sistema, un subsistema o una clase, de forma que los usuarios puedan comprender cómo utilizar ese elemento y de forma que los desarrolladores puedan implementarlo. Como se muestra en la Figura 18.1, se puede proporcionar un diagrama de casos de uso para modelar el comportamiento de esa caja (que la mayoría de la gente llamaría un teléfono móvil).

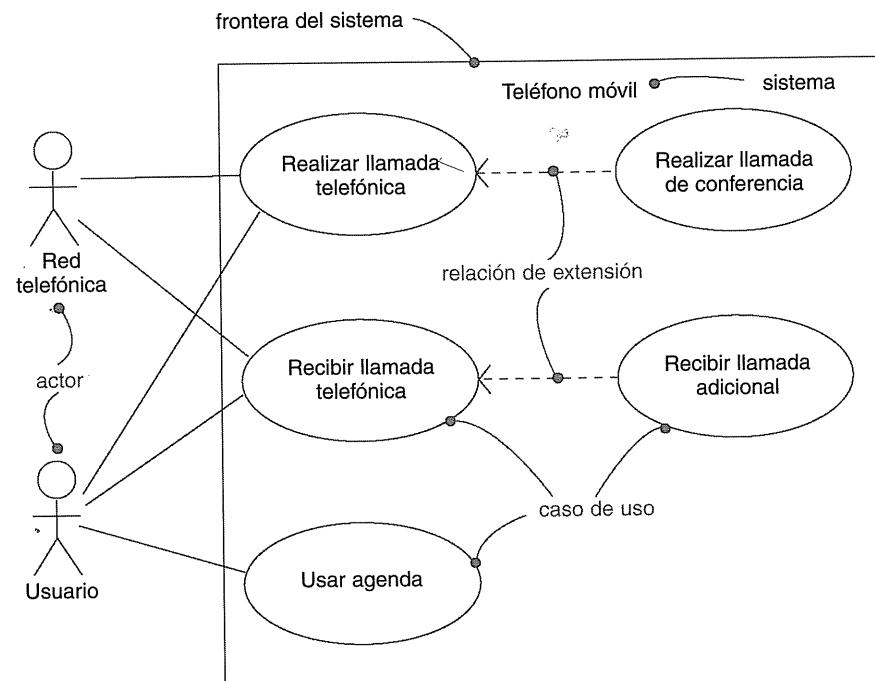


Figura 18.1: Un diagrama de casos de uso.

Términos y conceptos

Un *diagrama de casos de uso* es un diagrama que muestra un conjunto de casos de uso, actores y sus relaciones.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de casos de uso es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de casos de uso de los otros tipos de diagramas es su contenido particular.

Contenidos

Los casos de uso y los actores se discuten en el Capítulo 17; las relaciones se discuten en los Capítulos 5 y 10; los paquetes se discuten en el Capítulo 12; las instancias se discuten en el Capítulo 13.

Normalmente, un diagrama de casos de uso contiene:

- Sujetos.
- Casos de uso.
- Actores.
- Relaciones de dependencia, generalización y asociación.

Al igual que los demás diagramas, los diagramas de casos de uso pueden contener notas y restricciones.

Los diagramas de casos de uso también pueden contener paquetes, que se emplean para agrupar elementos del modelo en partes mayores. De vez en cuando, se pueden incluir instancias de casos de uso en los diagramas, especialmente cuando se quiera visualizar un sistema específico en ejecución.

Notación

El sujeto se representa como un rectángulo que contiene un conjunto de elipses que son los casos de uso. El nombre del sujeto se coloca dentro del rectángulo. Los actores se muestran como monigotes fuera del rectángulo, con el nombre debajo. Las líneas conectan los iconos de los actores con las elipses de los casos de uso con los que se comunican. Las relaciones entre los casos de uso (como la extensión y la inclusión) se dibujan dentro del rectángulo.

La vista de casos de uso se discute en el Capítulo 2.

Usos comunes

Los diagramas de casos de uso se emplean para modelar la vista de casos de uso de un sujeto, como un sistema. Esta vista abarca principalmente el comportamiento externo del sujeto (los servicios visibles externamente que el sujeto proporciona en el contexto de su entorno).

Cuando se modela la vista de casos de uso estática de un sujeto, normalmente se emplean los diagramas de casos de uso de una de las dos formas siguientes:

1. Para modelar el contexto de un sujeto.

Modelar el contexto de un sujeto implica dibujar una línea alrededor de todo el sistema e indicar qué actores quedan fuera del sistema e interactúan con él. Aquí, se emplean los diagramas de casos de uso para especificar los actores y el significado de sus roles.

2. Para modelar los requisitos de un sujeto.

El modelado de los requisitos de un sujeto implica especificar qué debería hacer ese sujeto (desde un punto de vista externo), independientemente de cómo lo haga. Aquí se emplean los diagramas de casos de uso para especificar el comportamiento deseado del sistema. De esta forma, un diagrama de casos de uso permite ver el sujeto entero como una caja negra; se puede ver qué hay fuera del sujeto y cómo reacciona a los elementos externos, pero no se puede ver cómo funciona por dentro.

Los requisitos se discuten en los Capítulos 4 y 6.

Técnicas comunes de modelado

Modelado del contexto de un sistema

Dado un sistema (cualquier sistema), algunos elementos se encuentran dentro de él y otros fuera. Por ejemplo, en un sistema de validación de tarjetas de crédito existen elementos como cuentas, transacciones y agentes de detección de fraudes dentro del sistema. También existen cosas como clientes de tarjetas de crédito y comercios fuera del sistema. Los elementos del sistema son responsables de llevar a cabo el comportamiento que esperan los elementos externos. Todos estos elementos externos que interactúan con el sistema constituyen su contexto. Este contexto define el entorno en el que reside el sistema.

Los sistemas se discuten en el Capítulo 32.

En UML se puede modelar el contexto de un sistema con un diagrama de casos de uso, destacando los actores en torno al sistema. La decisión acerca de qué incluir como un actor es importante, porque al hacer eso se especifica un tipo de cosas que interactúan con el sistema. La decisión acerca de qué no incluir es igualmente importante, si no más, porque restringe el entorno para que sólo incluya a aquellos actores necesarios en la vida del sistema.

Para modelar el contexto de un sistema:

- Hay que identificar las fronteras del sistema decidiendo los comportamientos que formarán parte de él y cuáles serán ejecutados por entidades externas. Esto define el sujeto.
- Hay que identificar los actores en torno al sistema, considerando qué grupos requieren ayuda del sistema para llevar a cabo sus tareas; qué grupos son necesarios para ejecutar las funciones del sistema; qué grupos interactúan con el hardware externo o con otros sistemas software; y qué grupos realizan funciones secundarias de administración y mantenimiento.
- Hay que organizar los actores similares en jerarquías de generalización/especialización.
- Hay que proporcionar un estereotipo para cada uno de esos actores, si así se ayuda a entender el sistema.

Hay que introducir esos actores en un diagrama de casos de uso y especificar las vías de comunicación de cada actor con los casos de uso del sistema.

Por ejemplo, la Figura 18.2 muestra el contexto de un sistema de validación de tarjetas de crédito, destacando los actores en torno al sistema. Se puede ver que existen Clientes, de los cuales hay dos categorías (Cliente individual y Cliente corporativo). Estos actores representan los roles que juegan las personas que interactúan con el sistema. En este contexto, también hay actores que representan a otras instituciones, tales como Comercio (con el cual un Cliente realiza una transacción con tarjeta para comprar un artículo o un servicio) y Entidad financiera (que presta servicio como sucursal bancaria para la cuenta de la tarjeta de crédito). En el mundo real, estos dos últimos actores probablemente serán a su vez sistemas con gran cantidad de software.

Los subsistemas se discuten en el Capítulo 32.

Esta misma técnica se utiliza para el modelado del contexto de un subsistema. Un sistema a un nivel dado de abstracción es a menudo un subsistema de un sistema mayor a un nivel de abstracción mayor. Por lo tanto, el modelado del contexto de un subsistema es útil al construir sistemas de sistemas interconectados.

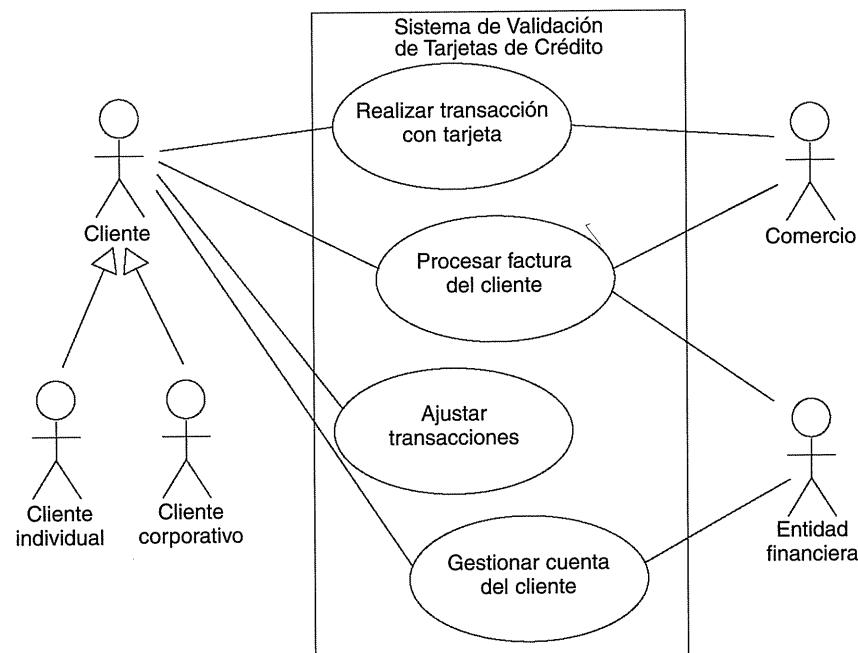


Figura 18.2: Modelado del contexto de un sistema.

Modelado de los requisitos de un sistema

Un requisito es una característica de diseño, una propiedad o un comportamiento de un sistema. Cuando se enuncian los requisitos de un sistema se está estableciendo un contrato entre los elementos externos al sistema y el propio sistema, que establece lo que se espera que haga el sistema. La mayoría de las veces no importa cómo lo hace, sólo importa *que* lo hace. Un sistema con un comportamiento correcto llevará a cabo todos sus requisitos de manera fiel, predecible y fiable. Al construir un sistema, es importante que al comenzar exista un acuerdo sobre qué debería hacer el sistema, aunque, con total seguridad, la comprensión de los requisitos evolucionará conforme se vaya implementando el sistema de manera iterativa e incremental. Análogamente, cuando se le proporciona un sistema a alguien para que lo use, es esencial saber cómo se comporta para utilizarlo correctamente.

Las notas pueden utilizarse para escribir los requisitos, como se discute en el Capítulo 6.

Los requisitos se pueden expresar de varias formas, desde texto sin estructura hasta expresiones en un lenguaje formal, pasando por cualquier otra forma intermedia. La mayoría de los requisitos funcionales de un sistema, si no todos,

El modelado de la dinámica para el balanceo de la carga y la reconfiguración de una red se discute en el Capítulo 24.

se pueden expresar con casos de uso, y los diagramas de casos de uso de UML son fundamentales para manejar esos requisitos.

Para modelar los requisitos de un sistema:

- Hay que establecer el contexto del sistema, identificando los actores a su alrededor.
- Hay que considerar el comportamiento que cada actor espera del sistema o requiere que éste le proporcione.
- Hay que nombrar esos comportamientos comunes como casos de uso.
- Hay que factorizar el comportamiento común en nuevos casos de uso que puedan ser utilizados por otros; hay que factorizar el comportamiento variante en nuevos casos de uso que extiendan los flujos principales.
- Hay que modelar esos casos de uso, actores y relaciones en un diagrama de casos de uso.
- Hay que adornar esos casos de uso con notas que enuncien los requisitos no funcionales; puede que haya que asociar varias de estas notas al sistema global.

La Figura 18.3 extiende el anterior diagrama de casos de uso. Aunque omite las relaciones entre los actores y los casos de uso, añade casos de uso adicionales que son invisibles para el cliente normal, aunque son comportamientos fundamentales del sistema. Este diagrama es valioso porque ofrece un punto de partida común para los usuarios finales, los expertos del dominio y los desarrolladores para visualizar, especificar, construir y documentar sus decisiones sobre los requisitos funcionales del sistema. Por ejemplo, Detectar fraude de tarjeta es un comportamiento importante tanto para el Comercio como para la Entidad financiera. Análogamente, Informe de estado de la cuenta es otro comportamiento requerido del sistema por varias entidades del contexto.

El requisito modelado por el caso de uso Gestión de corte de fluido eléctrico es un poco diferente de los demás, porque representa un comportamiento secundario del sistema necesario para un funcionamiento fiable y continuo.

Una vez determinada la estructura del caso de uso, hay que describir su comportamiento. Normalmente podemos escribir uno o más diagramas de secuencia para la línea principal de cada caso. A continuación habría que escribir diagramas de secuencia para las variaciones. Por último, habría que escribir al menos

un diagrama de secuencia para ilustrar cada tipo de excepción o condición de error. La gestión de los errores es parte del caso de uso y debería planificarse junto con comportamiento normal.

Los subsistemas se discuten en el Capítulo 32.

Esta misma técnica se utiliza para el modelado de los requisitos de un subsistema.

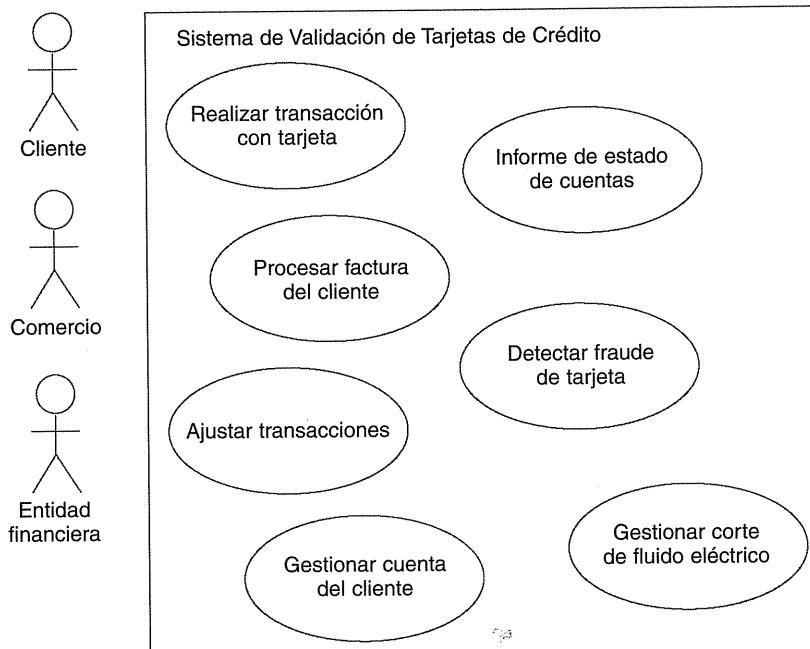


Figura 18.3: Modelado de los requisitos de un sistema.

Ingeniería directa e inversa

Los diagramas se discuten en el Capítulo 7; los casos de uso se discuten en el Capítulo 17.

La mayoría de los otros diagramas de UML, incluyendo los diagramas de clases, de componentes y de estados, son claros candidatos para la ingeniería directa e inversa, porque cada uno tiene un análogo en un sistema ejecutable. Los diagramas de casos de uso son algo diferentes, puesto que reflejan, más que especifican, la implementación de un sistema, un subsistema o una clase. Los casos de uso describen cómo se comporta un elemento, no cómo se implementa ese comportamiento, de forma que no se les puede aplicar directamente ingeniería directa o inversa.

La *ingeniería directa* es el proceso de transformar un modelo en código a través de una correspondencia con un lenguaje de implementación. A un diagra-

ma de casos de uso se le puede aplicar ingeniería directa para generar pruebas del elemento al que se aplica. Cada caso de uso de un diagrama especifica un flujo de eventos (y variantes de ese flujo), y esos flujos especifican cómo se espera que se comporte el elemento (algo que merece la pena probar). Un caso de uso bien estructurado especificará incluso pre y poscondiciones para definir el estado inicial de una prueba y los criterios de éxito. Para cada caso de uso de un diagrama, puede crearse un caso de prueba que se puede ejecutar cada vez que se produce una nueva versión del elemento, con el fin de confirmar que funciona como se requería, antes de que otros elementos confíen en él.

Para hacer ingeniería directa con un diagrama de casos de uso:

- Hay que identificar los objetos que interactúan con el sistema. También deben identificarse los diferentes roles que puede desempeñar cada entidad externa.
- Hay que caracterizar a un actor para que represente cada uno de los diferentes roles de interacción.
- Hay que identificar el flujo de eventos de cada caso de uso, y su flujo de eventos excepcional.
- Según el grado en el que se deseé profundizar con las pruebas, hay que generar un guión de prueba para cada flujo, con la precondición del flujo como el estado inicial de la prueba y sus poscondiciones como los criterios de éxito.
- Si es necesario, hay que generar una estructura de prueba para representar cada actor que interactúa con el caso de uso. Los actores que envían información al elemento o aquellos sobre los que actúa el sistema pueden ser simulados o sustituidos por sus equivalentes del mundo real.
- Hay que usar herramientas que ejecuten estas pruebas cada vez que se genere una nueva versión del elemento al que se aplica el diagrama de casos de uso.

La *ingeniería inversa* es el proceso de transformar código en un modelo a través de una correspondencia con un lenguaje de implementación específico. Conseguir un diagrama de casos de uso a través de ingeniería inversa automáticamente es impensable con la tecnología actual, simplemente porque hay una pérdida de información al pasar de la especificación de cómo se debe comportar un elemento al cómo se implementa. Sin embargo, se puede estudiar un sistema existente y discernir su comportamiento de forma manual, para ponerlo después en forma de diagrama de casos de uso. En realidad, esto

es bastante parecido a lo que se hace cada vez que uno recibe un software que no está documentado. Los diagramas de casos de uso de UML proporcionan simplemente un lenguaje estándar y expresivo con el que plasmar lo que se ha descubierto.

Para hacer un diagrama de casos de uso mediante ingeniería inversa:

- Hay que identificar cada actor que interactúe con el sistema.
- Hay que considerar la forma en que cada actor interactúa con el sistema, cambia el estado del sistema o de su entorno, o responde a algún evento.
- Hay que relacionar el flujo de eventos del sistema ejecutable con cada actor. Se debe comenzar con los flujos principales, declarando el correspondiente caso de uso, y considerar, posteriormente, los caminos alternativos.
- Hay que agrupar los flujos relacionados declarando el caso de uso correspondiente. Debe considerarse el modelado de variantes con relaciones de extensión, y hay que considerar los flujos comunes aplicando relaciones de inclusión.
- Hay que representar esos actores y casos de uso en un diagrama de casos de uso, y establecer sus relaciones.

Sugerencias y consejos

Cuando se crean diagramas de casos de uso en UML, hay que recordar que cada diagrama de casos de uso es una representación gráfica de la vista de casos de uso estática de un sistema. Esto significa que un único diagrama de casos de uso no tiene por qué capturar toda la vista de casos de uso del sistema. En conjunto, todos los diagramas de casos de uso representan la vista de casos de uso estática completa de un sistema; individualmente, cada uno representa sólo un aspecto.

Un diagrama de casos de uso bien estructurado:

- Se ocupa de modelar un aspecto de la vista de casos de uso estática de un sistema.
- Contiene sólo aquellos casos de uso y actores esenciales para comprender ese aspecto.

- Proporciona detalles de forma consistente con su nivel de abstracción; sólo se deben mostrar aquellos adornos (como los puntos de extensión) esenciales para su comprensión.
- No es tan minimalista que no ofrezca información al lector sobre los aspectos importantes de la semántica.

Cuando se dibuje un diagrama de casos de uso:

- Hay que darle un nombre que comunique su propósito.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que organizar sus elementos espacialmente para que los comportamientos y roles semánticamente cercanos se encuentren cercanos físicamente.
- Hay que utilizar las notas y los colores como señales visuales para llamar la atención sobre las características importantes del diagrama.
- Hay que intentar no mostrar demasiados tipos de relaciones. En general, si se tienen relaciones de extensión e inclusión complicadas, esos elementos se deberían llevar a otro diagrama.

Capítulo 19

DIAGRAMAS DE INTERACCIÓN



LENGUAJE
UNIFICADO DE
MODELADO

En este capítulo

- Modelado de flujos de control por ordenación temporal.
- Modelado de flujos de control por organización.
- Ingeniería directa e inversa.

Los diagramas de actividades, los diagramas de estados y los diagramas de casos de uso son otros tres tipos de diagramas que se utilizan en UML para modelar los aspectos dinámicos de los sistemas; los diagramas de actividades se discuten en el Capítulo 20; los diagramas de estados se discuten en el Capítulo 25; los diagramas de casos de uso se discuten en el Capítulo 18.

Los diagramas de secuencia y los diagramas de comunicación (ambos llamados diagramas de interacción) son dos de los tipos de diagramas de UML que se utilizan para modelar los aspectos dinámicos de los sistemas. Un diagrama de interacción muestra una interacción, que consiste en un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar entre ellos. Un diagrama de secuencia es un diagrama de interacción que destaca la ordenación temporal de los mensajes; un diagrama de comunicación es un diagrama de interacción que destaca la organización estructural de los objetos que envían y reciben mensajes.

Los diagramas de interacción se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar instancias concretas o prototípicas de clases, interfaces, componentes y nodos, junto con los mensajes enviados entre ellos, todo en el contexto de un escenario que ilustra un comportamiento. Los diagramas de interacción pueden servir para visualizar, especificar, construir y documentar la dinámica de una sociedad particular de objetos, o se pueden utilizar para modelar un flujo de control particular de un caso de uso.

Los diagramas de interacción no son sólo importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables por medio de ingeniería directa e inversa.

Introducción

Cuando vemos una película en vídeo o televisión, nuestra mente nos engaña. En lugar de ver un movimiento continuo como en la acción real, realmente se ve una serie de imágenes estáticas reproducidas tan rápidamente que producen la ilusión de un movimiento continuo.

Cuando los directores y los animadores planifican una película, utilizan la misma técnica pero con menor fidelidad. Al representar gráficamente mediante viñetas (*storyboarding*) los fotogramas clave de la película, están construyendo un modelo de cada escena, con el detalle suficiente para transmitir su intención a todos los integrantes del equipo de producción. De hecho, este tipo de representación es una actividad importante en el proceso de producción, que ayuda al equipo a visualizar, especificar, construir y documentar un modelo de la película conforme evoluciona desde su concepción inicial hasta la materialización y distribución final.

El modelado de los aspectos estructurales de un sistema se discute en las Partes 2 y 3.

Cuando se modelan sistemas con gran cantidad de software se tiene un problema similar: ¿cómo modelar sus aspectos dinámicos? Imaginemos, por un momento, cómo podría visualizarse un sistema en ejecución. Si disponemos de un depurador interactivo asociado al sistema, podríamos ver una sección de la memoria y observar cómo cambia su contenido a lo largo del tiempo. Fijándonos un poco más, incluso podríamos realizar el seguimiento de varios objetos de interés. A lo largo del tiempo, veríamos la creación de objetos, los cambios en el valor de sus atributos y la destrucción de algunos de ellos.

El valor de visualizar así los aspectos dinámicos de un sistema es bastante limitado, especialmente si se trata de un sistema distribuido con múltiples flujos de control concurrentes. También se podría intentar comprender el sistema circulatorio humano mirando la sangre que pasa a través de un punto de una arteria a lo largo del tiempo. Una forma mejor de modelar los aspectos dinámicos de un sistema es construir representaciones gráficas de escenarios que impliquen la interacción de ciertos objetos interesantes y los mensajes que se envían.

En UML, estas representaciones gráficas se modelan con los diagramas de interacción. Como se muestra en la Figura 19.1, estas representaciones se pueden construir de dos formas: destacando la ordenación temporal de los mensajes y destacando la relación estructural entre los objetos que interactúan. En cualquier caso, los diagramas son semánticamente equivalentes; se puede pasar de uno a otro sin pérdida de información.

Las interacciones se discuten en el Capítulo 16; los requisitos se discuten en el Capítulo 6.

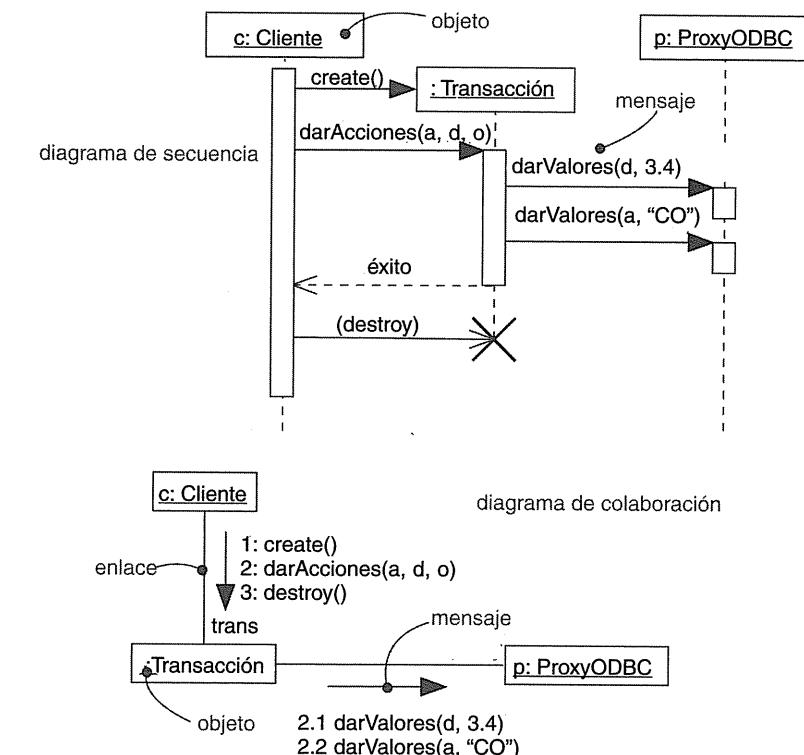


Figura 19.1: Diagramas de interacción.

Términos y conceptos

Un *diagrama de interacción* muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar entre ellos. Un *diagrama de secuencia* es un diagrama de interacción que destaca la ordenación temporal de los mensajes. Gráficamente, un diagrama de secuencia es una tabla que representa objetos, dispuestos a lo largo del eje X, y mensajes, ordenados según se suceden en el tiempo, a lo largo del eje Y. Un *diagrama de comunicación* es un diagrama de interacción que destaca la organización estructural de los objetos que envían y reciben mensajes. Gráficamente, un diagrama de comunicación es una colección de nodos y arcos.

Los subsistemas se discuten en el Capítulo 32; las clases se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11.

Los objetos se discuten en el Capítulo 13; los enlaces se discuten en los Capítulos 14 y 16; la estructura interna se discute en el Capítulo 15; las colaboraciones se discuten en el Capítulo 28.

Propiedades comunes

Un diagrama de interacción es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de interacción de los otros tipos de diagramas es su contenido particular.

Contenidos

Normalmente, los diagramas de interacción contienen:

- Roles u objetos.
- Comunicaciones o enlaces.
- Mensajes.

Nota: Un diagrama de interacción es básicamente una proyección de los elementos de una interacción. La semántica del contexto de una interacción, los objetos y roles, enlaces y conectores, mensajes y secuenciación se aplican a los diagramas de interacción.

Al igual que los demás diagramas, los diagramas de interacción pueden contener notas y restricciones.

Diagramas de secuencia

Un diagrama de secuencia destaca la ordenación temporal de los mensajes. Como se muestra en la Figura 19.2, un diagrama de secuencia se forma colocando en primer lugar los objetos o roles que participan en la interacción en la parte superior del diagrama, a lo largo del eje horizontal. Normalmente, se coloca a la izquierda el objeto o rol que inicia la interacción, y los objetos o roles subordinados a la derecha. A continuación, se colocan los mensajes que estos objetos envían y reciben a lo largo del eje vertical, en orden de sucesión en el tiempo, de arriba abajo. Esto ofrece al lector una señal visual clara del flujo de control a lo largo del tiempo.

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de comunicación.

En primer lugar, está la línea de vida. La línea de vida de un objeto es la línea discontinua vertical que representa la existencia de un objeto a lo largo de un pe-

ríodo de tiempo. La mayoría de los objetos que aparecen en un diagrama de interacción existirán mientras dure la interacción, así que los objetos se colocan en la parte superior del diagrama, con sus líneas de vida dibujadas desde arriba hasta abajo.

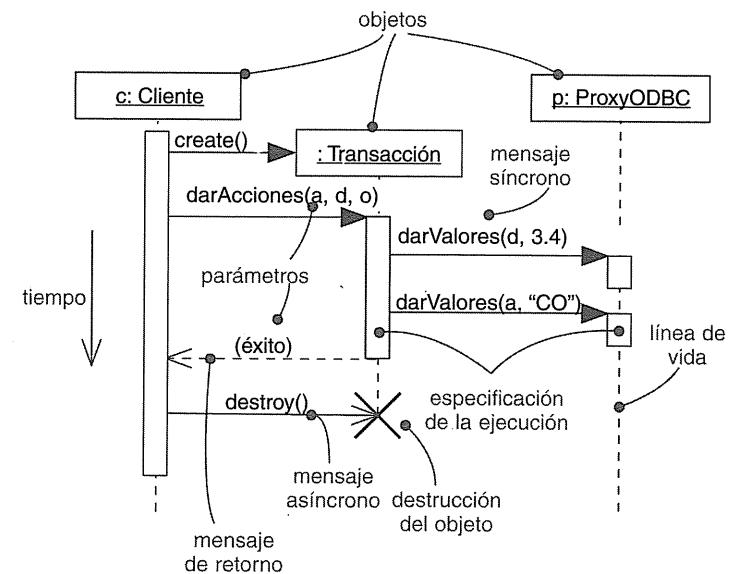


Figura 19.2: Diagrama de secuencia.

Pueden crearse objetos durante la interacción. Sus líneas de vida comienzan con la recepción del mensaje estereotipado como `create` (que va hasta el rectángulo donde empieza la línea de vida). Los objetos pueden destruirse durante la interacción. Sus líneas de vida acaban con la recepción del mensaje estereotipado como `destroy` (además se muestra la señal visual de una gran X que marca el final de sus vidas).

Nota: Si un objeto cambia el valor de sus atributos, su estado o sus roles, se puede colocar una copia del icono del objeto sobre su línea de vida en el punto en el que ocurre el cambio, mostrando esas modificaciones.

En segundo lugar, está el foco de control. El foco de control es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de un procedimiento subordinado. La parte superior del rectángulo se alinea con el comienzo de la acción; la inferior se alinea con su terminación (y puede marcarse con un

mensaje de retorno). También puede mostrarse el anidamiento de un foco de control (que puede estar causado por recursión, una llamada a una operación propia, o una llamada desde otro objeto) colocando otro foco de control ligeramente a la derecha de su foco padre (esto se puede hacer a cualquier nivel de profundidad). Si se quiere ser especialmente preciso acerca de dónde se encuentra el foco de control, también se puede sombrear la región del rectángulo durante la cual el método del objeto está ejecutándose y el control no ha pasado a otro objeto, pero esto es ser bastante meticuloso.

El contenido principal de los diagramas de secuencia son los mensajes. Un mensaje se representa con una flecha que va de una línea de vida hasta otra. La flecha apunta al receptor. Si el mensaje es asíncrono, la flecha es abierta. Si el mensaje es síncrono (una llamada), la flecha es un triángulo relleno. Una respuesta a un mensaje síncrono (el retorno de una llamada) se representa con una flecha abierta discontinua. El mensaje de retorno puede omitirse, ya que de manera implícita hay un retorno después de cada llamada, pero a menudo es útil para reflejar valores de retorno.

El orden del tiempo a lo largo de una línea de vida es significativo. Por lo general, no importa la distancia exacta; las líneas de vida sólo muestran secuencias relativas, es decir, no son diagramas del tiempo a escala. Por lo general, las posiciones de los mensajes en parejas distintas de líneas de vida no conllevan ninguna información sobre el orden en la secuencia; los mensajes pueden ocurrir en cualquier orden. El conjunto entero de mensajes en líneas de vida separadas forma un orden parcial. Una serie de mensajes establece una cadena de causalidad, de forma que cualquier punto sobre otra línea de vida al final de la cadena debe seguir siempre al punto en la línea de vida original al inicio de la cadena.

Control estructurado en los diagramas de secuencia

Una secuencia de mensajes está bien para mostrar una secuencia sencilla y lineal, pero a menudo necesitamos mostrar condicionales y bucles. A veces queremos mostrar la ejecución concurrente de varias secuencias. Este tipo de control de alto nivel puede mostrarse mediante operadores de control estructurados en los diagramas de secuencia.

Un operador de control se representa como una región rectangular dentro del diagrama de secuencia. El operador tiene una etiqueta (un texto dentro de un pequeño pentágono en la esquina superior izquierda) para indicar qué tipo de operador de control es. El operador se aplica a las líneas de vida que lo cruzan. Esto se considera el cuerpo del operador. Si una línea de vida no se aplica al

operador, puede ser interrumpida al principio del operador y continuada al final. Los siguientes tipos de controles son los más habituales:

Ejecución opcional. La etiqueta es opt. El cuerpo del operador de control se ejecuta si una condición de guarda es cierta cuando se entra en el operador. La condición de guarda es una expresión booleana que puede aparecer entre corchetes encima de cualquier línea de vida dentro del cuerpo y puede referenciar a los atributos del objeto.

Ejecución condicional. La etiqueta es alt. El cuerpo del operador de control se divide en varias subregiones con líneas discontinuas horizontales. Cada subregión representa una rama de la condición. Cada subregión tiene una condición de guarda. Si la condición de guarda para una subregión es cierta, la subregión se ejecuta. Sin embargo, sólo se ejecuta una subregión como máximo; si son ciertas más de una condición de guarda, la elección de la subregión no es determinista y podría variar de ejecución en ejecución. Si no es cierta ninguna condición de guarda, entonces el control continúa después del operador de control. Puede haber una subregión con una condición de guarda especial [else]; esta subregión se ejecuta si no es cierta ninguna de las otras condiciones de guarda.

Ejecución paralela. La etiqueta es par. El cuerpo del operador de control se divide en varias subregiones con líneas discontinuas horizontales. Cada subregión representa una computación paralela (concurrente). En la mayoría de los casos, cada subregión implica diferentes líneas de vida. Cuando se entra en el operador de control, todas las subregiones se ejecutan concurrentemente. La ejecución de mensajes en cada subregión es secuencial, pero el orden relativo de los mensajes en las subregiones paralelas es completamente arbitrario. Este constructor no debe usarse si las diferentes computaciones interactúan. Sin embargo, hay muchas situaciones del mundo real que se descomponen en actividades paralelas e independientes, así que éste es un operador muy útil.

Ejecución en bucle (iterativa). La etiqueta es loop. Una condición de guarda aparece sobre una línea de vida dentro del cuerpo. El cuerpo del bucle se ejecuta repetidamente mientras la condición de guarda sea cierta antes de cada iteración. Cuando la condición de guarda sea cierta en la parte superior del cuerpo, el control continúa fuera del operador de control.

Hay muchos otros tipos de operadores, pero éstos son los más útiles.

Para proporcionar una indicación clara de la frontera, un diagrama de secuencia puede encerrarse en un rectángulo, con una etiqueta en la esquina superior izquierda. La etiqueta es sd, que puede estar seguida del nombre del diagrama.

La Figura 19.3 muestra un ejemplo simplificado que ilustra algunos operadores de control. El usuario inicia la secuencia. El primer operador es un operador de bucle. Los números entre paréntesis (1,3) indican el número mínimo y máximo de veces que debe ejecutarse el cuerpo del bucle. Como el mínimo es uno, el cuerpo siempre se ejecuta al menos una vez antes de que se compruebe la condición. En el bucle, el usuario introduce la clave y el sistema la verifica. El bucle continúa mientras la clave sea incorrecta. Sin embargo, después de tres veces, el bucle termina en cualquier caso.

El siguiente operador es un operador opcional. El cuerpo opcional se ejecuta si la clave es válida; en otro caso, se ignora el resto del diagrama de secuencia. El cuerpo opcional contiene un operador paralelo. Los operadores pueden anidarse como se muestra en la figura.

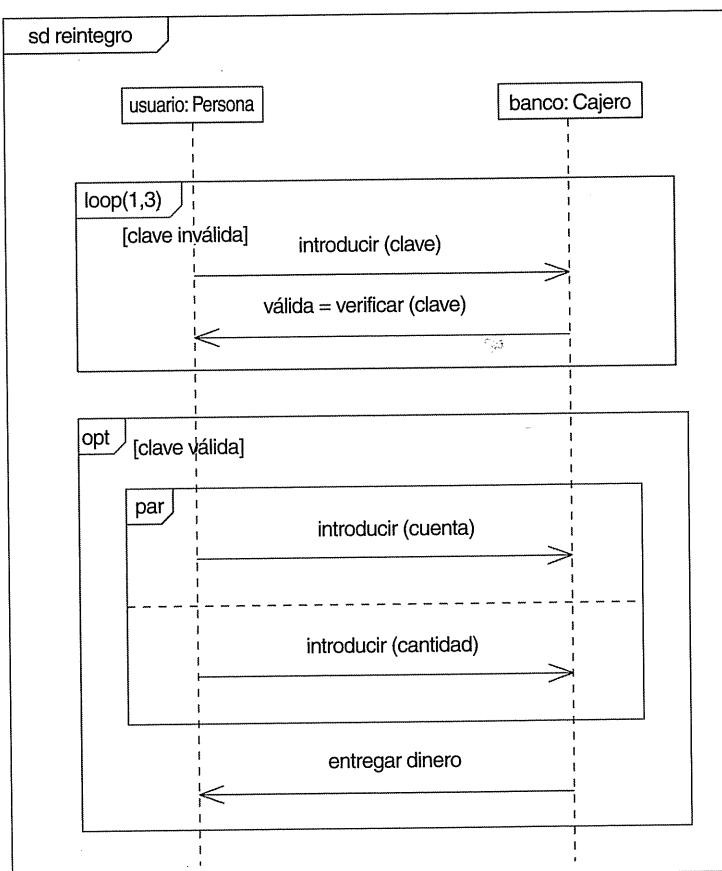


Figura 19.3: Operadores de control estructurados.

El operador paralelo tiene dos subregiones: una permite que el usuario introduzca una cuenta y la otra permite que el usuario introduzca una cantidad. Al ser paralelas, no se requiere un orden para capturar ambos datos; pueden introducirse en cualquier orden. Esto destaca el hecho de que la concurrencia no siempre implica una ejecución que sea físicamente simultánea. La concurrencia en realidad significa que dos acciones no están coordinadas y pueden ocurrir en cualquier orden. Si son acciones verdaderamente independientes, pueden solaparse; si son acciones secuenciales, pueden ocurrir en cualquier orden.

Una vez que ambas acciones se han ejecutado, el operador paralelo se ha completado. En la siguiente acción dentro del operador opcional, el banco entrega el dinero al usuario. El diagrama de secuencia ha terminado aquí.

Diagramas de actividad anidados

Los diagramas de actividad que son demasiado grandes pueden ser difíciles de comprender. Las secciones estructuradas de una actividad pueden organizarse en una actividad subordinada, especialmente si la actividad subordinada se utiliza más de una vez dentro de la actividad principal. La actividad principal y las actividades subordinadas se representan en diagramas diferentes. Dentro del diagrama de actividad principal, el uso de una actividad subordinada se representa con un rectángulo con la etiqueta `ref` en su esquina superior izquierda; el nombre del comportamiento subordinado se representa dentro del rectángulo. El comportamiento subordinado no se restringe a un diagrama de actividad; también podría ser una máquina de estados, un diagrama de secuencia u otra especificación de comportamiento. La Figura 19.4 muestra el diagrama de la Figura 19.3 reorganizado, del que se han extraído dos secciones a diagramas de actividad separados que son referenciados desde el diagrama principal.

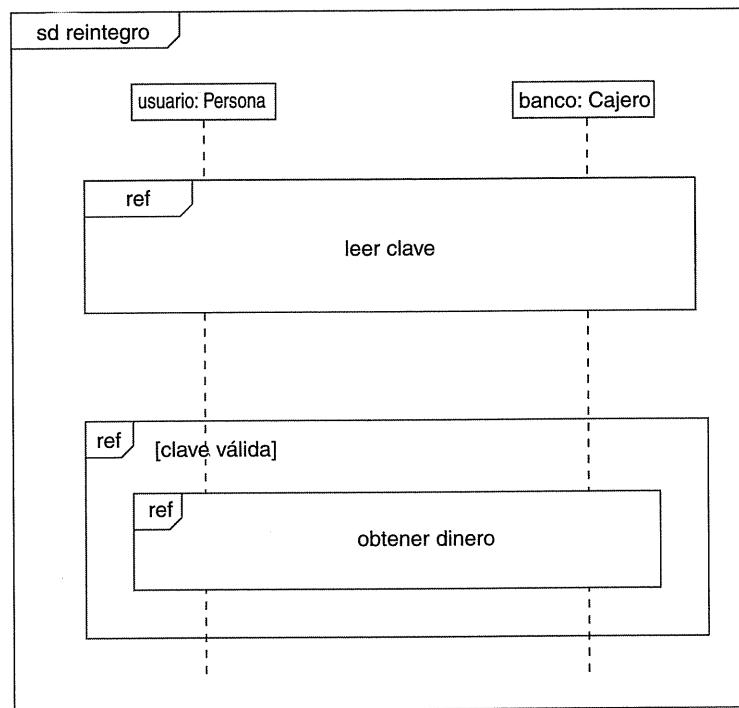


Figura 19.4: Diagrama de actividades anidado.

Diagramas de comunicación

Un diagrama de comunicación destaca la organización de los objetos que participan en una interacción. Como se muestra en la Figura 19.5, un diagrama de comunicación se construye colocando en primer lugar los objetos que participan en la colaboración como nodos del grafo. A continuación se representan los enlaces que conectan esos objetos como arcos del grafo. Los enlaces pueden tener nombres de roles para identificarlos. Por último, estos enlaces se adornan con los mensajes que los objetos envían y reciben. Esto da al lector una señal visual clara del flujo de control en el contexto de la organización estructural de los objetos que colaboran.

Se puede utilizar una forma avanzada de los números de secuencia para distinguir los flujos de control concurrentes, como se discute en el Capítulo 23; los estereotipos de camino se discuten en el Capítulo 18; las ramificaciones y las iteraciones complejas se pueden especificar más fácilmente en los diagramas de actividades, como se discute en el Capítulo 20.

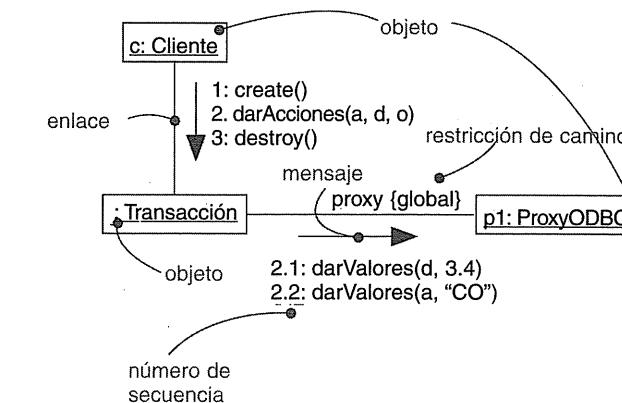


Figura 19.5: Diagrama de comunicación.

Nota: Al contrario que en un diagrama de secuencia, en un diagrama de colaboración no se muestra explícitamente la línea de vida de un objeto, aunque se pueden representar los mensajes *create* y *destroy*. Además, el foco de control no se muestra explícitamente en un diagrama de colaboración, aunque el número de secuencia de cada mensaje puede indicar el anidamiento.

Los diagramas de comunicación tienen dos características que los distinguen de los diagramas de secuencia.

En primer lugar, el camino. El camino se dibuja haciéndolo corresponder con una asociación. También se dibujan caminos haciéndolos corresponder con variables locales, parámetros, variables globales o accesos propios. Un camino representa una fuente de conocimiento de un objeto.

En segundo lugar, está el número de secuencia. Para indicar la ordenación temporal de un mensaje, se precede de un número (comenzando con el mensaje número 1), que se incrementa secuencialmente por cada nuevo mensaje en el flujo de control (2, 3, etc.). Para representar el anidamiento, se utiliza la numeración decimal de Dewey (1 es el primer mensaje; 1.1 es el primer mensaje dentro del mensaje 1; 1.2 es el segundo mensaje dentro del mensaje 1; etc.). El anidamiento se puede representar a cualquier nivel de profundidad. También hay que tener en cuenta que se pueden mostrar varios mensajes a través del mismo enlace (posiblemente enviados desde distintas direcciones), y cada uno tendrá un número de secuencia único.

La mayoría de las veces se modelarán flujos de control simples y secuenciales. Sin embargo, también se pueden modelar flujos más complejos, que impliquen

iteración y bifurcación. Una iteración representa una secuencia repetida de mensajes. Para modelar una iteración, el número de secuencia de un mensaje se precede de una expresión de iteración, como en `* [i := 1 .. n]` (o sólo `*` si se quiere indicar iteración pero no se desea especificar los detalles). Una iteración indica que el mensaje (y cualquier mensaje anidado) se repetirá de acuerdo con la expresión dada. Análogamente, una condición representa un mensaje cuya ejecución depende de la evaluación de una expresión booleana. Para modelar una condición, el número de secuencia de un mensaje se precede de una cláusula de condición, como `[x > 0]`. Los distintos caminos alternativos de una bifurcación tendrán el mismo número de secuencia, pero cada camino debe ser distinguible de forma única por una condición que no se solape con las otras.

Tanto para la iteración como para la bifurcación, UML no impone el formato de la expresión entre corchetes; se puede utilizar pseudocódigo o la sintaxis de un lenguaje de programación específico.

Nota: Los enlaces entre los objetos no se muestran explícitamente en un diagrama de secuencia. Tampoco se representa explícitamente el número de secuencia de un mensaje en un diagrama de secuencia: va implícito en la ordenación temporal de los mensajes desde la parte superior a la inferior del diagrama. No obstante, también pueden mostrarse la iteración y la bifurcación utilizando las estructuras de control de los diagramas de secuencia.

Equivalencia semántica

Los diagramas de secuencia y de comunicación son semánticamente equivalentes, ya que ambos derivan de la misma información del metamodelo de UML. Como consecuencia de esto, se puede partir de un diagrama en una forma y convertirlo a la otra sin pérdida de información, como se muestra en las dos figuras anteriores, que son semánticamente equivalentes. Sin embargo, esto no significa que ambos diagramas visualicen explícitamente la misma información. Por ejemplo, en las dos figuras anteriores, el diagrama de comunicación muestra cómo se enlazan los objetos (nótese las anotaciones `{local}` y `{global}`), mientras que el diagrama de secuencia correspondiente no lo muestra. Análogamente, el diagrama de secuencia muestra el retorno de mensajes (nótese el valor de retorno `exito`), pero el diagrama de comunicación correspondiente no lo hace. En ambos casos, los dos diagramas comparten el mismo modelo subyacente, pero cada uno puede representar cosas que el otro no representa. No obstante, un modelo introducido en un formato puede carecer de parte de la información reflejada en el otro formato; así

que, aunque el modelo subyacente pueda incluir ambos tipos de información, los dos tipos de diagramas pueden conducir a modelos diferentes.

Usos comunes

Las cinco vistas de una arquitectura se discuten en el Capítulo 2; las instancias se discuten en el Capítulo 13; las clases se discuten en los Capítulos 4 y 9; las clases activas se discuten en el Capítulo 23; las interfaces se discuten en el Capítulo 11; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los sistemas y los subsistemas se discuten en el Capítulo 32; las operaciones se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Los diagramas de interacción se utilizan para modelar los aspectos dinámicos de un sistema. Estos aspectos dinámicos pueden involucrar la interacción de cualquier tipo de instancia en cualquier vista de la arquitectura de un sistema, incluyendo instancias de clases (incluso clases activas), interfaces, componentes y nodos.

Cuando se usa un diagrama de interacción para modelar algún aspecto dinámico de un sistema, se hace en el contexto de un sistema global, un subsistema, una operación o una clase. También se pueden asociar diagramas de interacción a los casos de uso (para modelar un escenario) y a las colaboraciones (para modelar los aspectos dinámicos de una sociedad de objetos).

Cuando se modelan los aspectos dinámicos de un sistema, normalmente se utilizan los diagramas de interacción de dos formas.

1. Para modelar flujos de control por ordenación temporal.

Para ello se utilizan los diagramas de secuencia. El modelado de un flujo de control por ordenación temporal destaca el paso de mensajes tal y como se desarrolla a lo largo del tiempo, lo que es una forma útil de visualizar el comportamiento dinámico en el contexto de un escenario de un caso de uso. Los diagramas de secuencia hacen un papel mejor que los diagramas de comunicación para mostrar las iteraciones y las bifurcaciones sencillas.

2. Para modelar flujos de control por organización.

Para ello se utilizan los diagramas de comunicación. El modelado de un flujo de control por organización destaca las relaciones estructurales entre las instancias de la interacción, a través de las cuales pasan los mensajes.

Técnicas comunes de modelado

Modelado de flujos de control por ordenación temporal

Los sistemas y los subsistemas se discuten en el Capítulo 32; las operaciones y las clases se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Considérense los objetos existentes en el contexto de un sistema, un subsistema, una operación o una clase. Considérense también los objetos y roles que participan en un caso de uso o en una colaboración. Para modelar un flujo de control que discurre entre esos objetos y roles se utiliza un diagrama de interacción; para destacar el paso de mensajes conforme se desarrollan en el tiempo se utiliza un diagrama de secuencia, un tipo de diagramas de interacción.

Para modelar un flujo de control por ordenación temporal:

- Hay que establecer el contexto de la interacción, bien sea un sistema, un subsistema, una operación, o una clase, o bien un escenario de un caso de uso o de una colaboración.
- Hay que establecer el escenario de la interacción, identificando qué objetos juegan un rol en ella. Los objetos deben organizarse en el diagrama de secuencia de izquierda a derecha, colocando los objetos más importantes a la izquierda y sus objetos vecinos a la derecha.
- Hay que establecer la línea de vida de cada objeto. En la mayoría de los casos, los objetos persistirán durante la interacción completa. Para aquellos objetos creados y destruidos durante la interacción, hay que establecer sus líneas de vida según corresponda, e indicar explícitamente su creación y destrucción con mensajes estereotipados apropiadamente.
- A partir del mensaje que inicia la interacción, hay que ir colocando los mensajes subsiguientes de arriba abajo entre las líneas de vida, mostrando las propiedades de cada mensaje (tales como sus parámetros), según sea necesario para explicar la semántica de la interacción.
- Si es necesario visualizar el anidamiento de mensajes o el intervalo de tiempo durante el cual tiene lugar la computación, hay que adornar la línea de vida de cada objeto con su foco de control.
- Si es necesario especificar restricciones de tiempo o espacio, hay que adornar cada mensaje con una marca de tiempo y asociar las restricciones apropiadas.
- Si es necesario especificar este flujo de control más formalmente, hay que asociar pre y poscondiciones a cada mensaje.

Las marcas de tiempo se discuten en el Capítulo 24; las pre y postcondiciones se discuten en el Capítulo 4; los paquetes se discuten en el Capítulo 12.

Las señales se discuten en el Capítulo 21; las marcas de tiempo se discuten en el Capítulo 24; las restricciones se discuten en el Capítulo 6; las responsabilidades se discuten en el Capítulo 4; las notas se discuten en el Capítulo 6.

Un único diagrama de secuencia sólo puede mostrar un flujo de control (aunque es posible mostrar concurrencia estructurada utilizando los constructores de control estructurados). Normalmente, se realizarán varios diagramas de interacción, algunos de los cuales serán los principales y los demás mostrarán caminos alternativos o condiciones excepcionales. Se pueden utilizar paquetes para organizar estas colecciones de diagramas de secuencia, dando a cada diagrama un nombre adecuado para distinguirlo del resto.

Por ejemplo, la Figura 19.6 representa un diagrama de secuencia que especifica el flujo de control para iniciar una simple llamada telefónica entre dos partes. A este nivel de abstracción existen cuatro objetos involucrados: dos Interlocutores (s y r), una Centralita de teléfonos sin nombre, y c, la materialización de la Conversación entre ambas partes. Debe notarse que el diagrama modela los cuatro roles; cada instancia del diagrama tiene objetos específicos ligados a cada uno de los roles. El mismo patrón de interacción se aplica a cada instancia del diagrama.

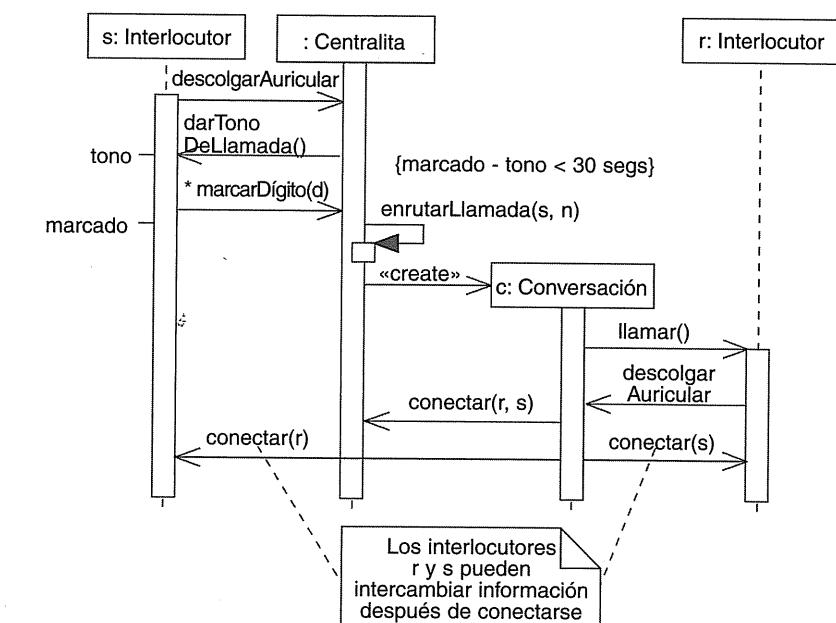


Figura 19.6: Modelado de flujos de control por ordenación temporal.

La secuencia comienza cuando un Interlocutor (*s*) emite una señal (*descolgarAuricular*) al objeto Centralita. A su vez, la Centralita llama a darTonoDeLlamada sobre este Interlocutor, y el Interlocutor itera sobre el mensaje marcarDigito. Nótese que esta secuencia debe durar menos de 30 segundos, tal y como indica la restricción. El diagrama no indica qué ocurre si se viola la restricción temporal. Para ello podría incluirse una bifurcación o un diagrama de secuencia aparte. El objeto Centralita se llama a sí mismo con el mensaje enrutarLlamada. A continuación crea un objeto Conversacion (*c*), al cual delega el resto del trabajo. Aunque no se representa en esta interacción, *c* tendrá la responsabilidad adicional de formar parte del sistema de contabilidad de la centralita (lo cual se expresaría en otro diagrama de interacción). El objeto Conversacion (*c*) llama al Interlocutor (*r*), el cual envía asíncronamente el mensaje descolgarAuricular. Entonces, el objeto Conversacion indica a la Centralita que debe conectar la llamada, y luego indica a los dos objetos Interlocutor que pueden conectar, tras lo cual pueden intercambiar información, como se indica en la nota adjunta.

Un diagrama de secuencia puede comenzar o terminar en cualquier punto de una secuencia. Una traza completa del flujo de control sería increíblemente compleja, así que es razonable dividir las partes de un flujo grande en diagramas separados.

Modelado de flujos de control por organización

Los sistemas y los subsistemas se discuten en el Capítulo 32; las operaciones y las clases se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Considérense los objetos existentes en el contexto de un sistema, un subsistema, una operación o una clase. Considérense también los objetos y roles que participan en un caso de uso o una colaboración. Para modelar un flujo de control que discurre entre esos objetos y roles se utiliza un diagrama de interacción; para mostrar el paso de mensajes en el contexto de esa estructura se utiliza un diagrama de comunicación, un tipo de diagramas de interacción.

Para modelar el flujo de control por organización:

- Hay que establecer el contexto de la interacción, bien sea un sistema, un subsistema, una operación, o una clase, o bien un escenario de un caso de uso o de una colaboración.
- Hay que establecer el escenario de la interacción, identificando qué objetos juegan un rol en ella. Estos objetos deben organizarse en el diagrama de comunicación como los nodos de un grafo, colocando los objetos más importantes en el centro del diagrama y sus vecinos hacia el exterior.

Las relaciones de dependencia se discuten en los Capítulos 5 y 10; las restricciones de camino se discuten en el Capítulo 16.

Las marcas de tiempo se discuten en el Capítulo 24; las pre y postcondiciones se discuten en el Capítulo 4; los paquetes se discuten en el Capítulo 12.

- Hay que especificar los enlaces entre esos objetos, junto a los mensajes que pueden pasar.
 1. Colocar los enlaces de asociaciones en primer lugar; éstos son los más importantes, porque representan conexiones estructurales.
 2. Colocar los demás enlaces a continuación, y adornarlos con las anotaciones de camino adecuadas (como global y local) para especificar explícitamente cómo se conectan estos objetos entre sí.
- Comenzando por el mensaje que inicia la interacción, hay que asociar cada mensaje subsiguiente al enlace apropiado, estableciendo su número de secuencia. Los anidamientos se representan con la numeración decimal de Dewey.
- Si es necesario especificar restricciones de tiempo o espacio, hay que adornar cada mensaje con una marca de tiempo y asociar las restricciones adecuadas.
- Si es necesario especificar el flujo de control más formalmente, hay que asociar pre y poscondiciones a cada mensaje.

Al igual que los diagramas de secuencia, un único diagrama de comunicación sólo puede mostrar un flujo de control (aunque se pueden representar variaciones sencillas utilizando la notación de UML para la iteración y la bifurcación). Normalmente se realizarán varios diagramas de interacción, algunos de los cuales serán principales y otros mostrarán caminos alternativos o condiciones excepcionales. Los paquetes se pueden utilizar para organizar estas colecciones de diagramas de comunicación, dando a cada diagrama un nombre para distinguirlo del resto.

Por ejemplo, la Figura 19.7 muestra un diagrama de comunicación que especifica el flujo de control para matricular un nuevo estudiante en una universidad, destacando las relaciones estructurales entre los objetos. Se ven cuatro roles: un EncargadoMatriculas (*r*), un Estudiante (*e*), un Curso (*c*) y un objeto Universidad sin nombre. El flujo de control está numerado explícitamente. La acción comienza cuando el EncargadoMatriculas crea un objeto Estudiante, añade el estudiante a la universidad (mensaje añadirEstudiante), y a continuación dice al objeto Estudiante que se matricule. El objeto Estudiante invoca a obtenerPlanEstudios sobre sí mismo, de donde obtiene los objetos Curso en los que se debe matricular. Después, el objeto Estudiante se añade a sí mismo a cada objeto Curso.

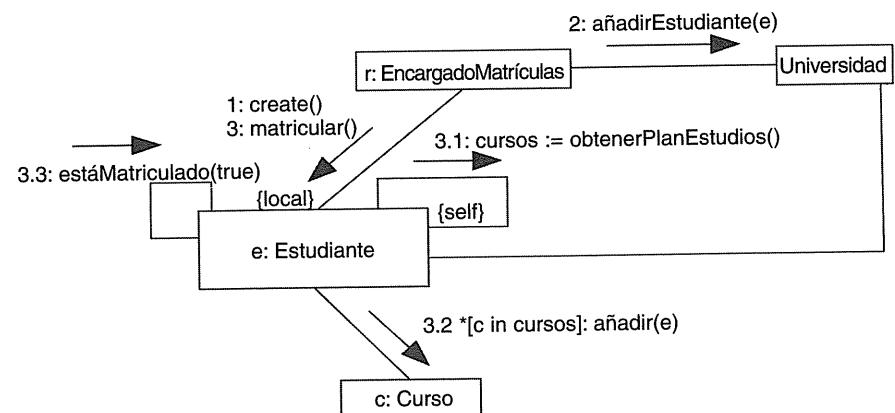


Figura 19.7: Modelado de flujos de control por organización.

Ingeniería directa e inversa

La *ingeniería directa* (creación de código a partir de un modelo) es posible tanto para los diagramas de secuencia como para los de comunicación, especialmente si el contexto del diagrama es una operación. Por ejemplo, con el diagrama de comunicación anterior, una herramienta de ingeniería directa razonablemente inteligente podría generar el siguiente código Java para la operación *matricular*, asociada a la clase *Estudiante*.

```

public void matricular() {
    ColeccionDeCursos c = obtenerPlanEstudios();
    for (int i = 0; i < c.size(); i++)
        c.item(i).añadir(this);
    this.matriculado = true;
}
  
```

“Razonablemente inteligente” significa que la herramienta debería darse cuenta de que *obtenerPlanEstudios* devuelve un objeto *ColecciónDeCursos*, lo que podría determinar mirando la firma de la operación. Al recorrer el contenido de este objeto con una construcción de iteración estándar (la cual podría ser conocida implícitamente por la herramienta), el código podría generalizarse a cualquier número de cursos ofrecidos.

La *ingeniería inversa* (creación de un modelo a partir de código) también es posible tanto para los diagramas de secuencia como para los de comunicación,

especialmente si el contexto del código es el cuerpo de una operación. Algunas partes del diagrama anterior podrían haber sido producidas por una herramienta a partir de una ejecución prototípica de la operación matricular.

Nota: La ingeniería directa es inmediata; la ingeniería inversa es difícil. Es fácil obtener un montón de información mediante simple ingeniería inversa, de forma que la parte difícil consiste en ser inteligente para saber qué detalles conservar.

Sin embargo, más interesante que la ingeniería inversa de un modelo a partir de código es la animación de un modelo durante la ejecución del sistema desarrollado. Por ejemplo, dado el diagrama anterior, una herramienta podría animar los mensajes del diagrama mientras se producen en un sistema en ejecución. Mejor aún, teniendo esta herramienta bajo el control de un depurador, se podría controlar la velocidad de ejecución, introduciendo posiblemente puntos de ruptura para detener la acción en puntos de interés, con el fin de examinar los valores de los atributos individuales.

Sugerencias y consejos

Cuando se crean diagramas de interacción en UML, hay que recordar que tanto los diagramas de secuencia como los de comunicación son proyecciones del mismo modelo de los aspectos dinámicos de un sistema. Un único diagrama de interacción no puede capturar todo acerca de la dinámica de un sistema. En lugar de ello, se utilizarán varios diagramas de interacción para modelar la dinámica del sistema global, así como la de sus subsistemas, operaciones, clases, casos de uso y colaboraciones.

Un diagrama de interacción bien estructurado:

- Se ocupa de modelar un aspecto de la dinámica de un sistema.
- Contiene sólo aquellos elementos esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con su nivel de abstracción y sólo debe mostrar aquellos adornos que son esenciales para su comprensión.
- No es tan minimalista que no ofrezca información al lector sobre los aspectos importantes de la semántica.

Cuando se dibuje un diagrama de interacción:

- Hay que darle un nombre que comunique su propósito.
- Si se quiere destacar la ordenación temporal de los mensajes, hay que utilizar un diagrama de secuencia. Si se quiere destacar la organización de los objetos implicados en la interacción, hay que utilizar un diagrama de comunicación.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre las características importantes del diagrama.
- Hay que usar la bifurcación de forma moderada; las bifurcaciones complejas se pueden representar mucho mejor con los diagramas de actividades.



Capítulo 20 DIAGRAMAS DE ACTIVIDADES

En este capítulo

- Modelado de un flujo de trabajo.
- Modelado de una operación.
- Ingeniería directa e inversa.

Los diagramas de secuencia, los diagramas de comunicación, los diagramas de estados y los diagramas de casos de uso también modelan los aspectos dinámicos de un sistema. Los diagramas de secuencia y los diagramas de comunicación se discuten en el Capítulo 19; los diagramas de estados se discuten en el Capítulo 25; los diagramas de casos de uso se discuten en el Capítulo 18; las acciones se discuten en el Capítulo 16.

Los diagramas de actividades son uno de los cinco tipos de diagramas de UML que se utilizan para el modelado de los aspectos dinámicos de los sistemas. Un diagrama de actividades es fundamentalmente un diagrama de flujo que muestra el flujo de control entre actividades. Al contrario que un diagrama de flujo clásico, un diagrama de actividad muestra tanto la concurrencia como las bifurcaciones del control.

Los diagramas de actividades se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar los pasos secuenciales (y posiblemente concurrentes) de un proceso computacional. Con un diagrama de actividades también se puede modelar el flujo de valores entre los distintos pasos. Los diagramas de actividades pueden servir para visualizar, especificar, construir y documentar la dinámica de una sociedad de objetos, o pueden emplearse para modelar el flujo de control de una operación. Mientras que los diagramas de interacción destacan el flujo de control entre objetos, los diagramas de actividades destacan el flujo de control entre los distintos pasos. Una actividad es una ejecución estructurada en curso de un comportamiento. La ejecución de una actividad se descompone finalmente en la ejecución de varias acciones individuales, cada una de las cuales puede cambiar el estado del sistema o puede comunicar mensajes.

Los diagramas de actividades no son sólo importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables a través de ingeniería directa e inversa.

Introducción

Considérese el flujo de trabajo asociado a la construcción de una casa. En primer lugar se selecciona un sitio. A continuación, se contrata a un arquitecto para diseñarla. Después de llegar a un acuerdo en el plano, el constructor pide presupuestos para establecer el precio. Una vez acordados un precio y un plano, puede comenzar la construcción. Se obtienen los permisos, se mueven tierras, se echan los cimientos, se erige la estructura, etcétera, hasta que todo queda hecho. Entonces se entregan las llaves y un certificado de vivienda, y el propietario toma posesión de la casa.

Aunque ésta es una tremenda simplificación de lo que realmente ocurre en un proceso de construcción, captura el camino crítico del flujo de trabajo. En un proyecto real, hay muchas actividades paralelas entre los distintos profesionales. Por ejemplo, los electricistas pueden trabajar a la vez que los fontaneros y los carpinteros. También podemos encontrar condiciones y bifurcaciones. Por ejemplo, dependiendo del resultado de las pruebas de las tierras, se tendrá que excavar, usar explosivos o colocar una estructura que soporte oscilaciones. Incluso podría haber iteraciones. Por ejemplo, una inspección podría revelar violaciones de las leyes que produjesen como resultado algo de escombro y repetición de trabajo.

En la industria de la construcción se utilizan frecuentemente técnicas como los diagramas de Gantt y los diagramas Pert para visualizar, especificar, construir y documentar el flujo de trabajo del proyecto.

El modelado de los aspectos estructurales de un sistema se discute en las Partes 2 y 3; los diagramas de interacción se discuten en el Capítulo 19.

Cuando se modelan sistemas con gran cantidad de software aparece un problema similar. ¿Cuál es la mejor forma de modelar un flujo de trabajo o una operación, que son ambos aspectos de la dinámica del sistema? La respuesta es que existen dos elecciones básicas, similares al uso de diagramas de Gantt y diagramas Pert.

Por un lado, se pueden construir representaciones gráficas de escenarios que involucren la interacción de ciertos objetos interesantes y los mensajes que se pueden enviar entre ellos. En UML se pueden modelar estas representaciones de dos formas: destacando la ordenación temporal de los mensajes (con diagramas de secuencia) o destacando las relaciones estructurales entre los objetos que interactúan (con diagramas de comunicación). Los diagramas de interacción son similares a los diagramas de Gantt, los cuales se centran en los objetos (recursos) que llevan a cabo alguna actividad a lo largo del tiempo.

Las acciones se discuten en el Capítulo 16.

Por otro lado, estos aspectos dinámicos se pueden modelar con diagramas de actividades, que se centran en las actividades que tienen lugar entre los objetos,

como se muestra en la Figura 20.1. En este sentido, los diagramas de actividades son similares a los diagramas Pert. Un diagrama de actividades es esencialmente un diagrama de flujo que destaca la actividad que tiene lugar a lo largo del tiempo. Se puede pensar en un diagrama de actividades como en un diagrama de interacción al que se le ha dado la vuelta. Un diagrama de interacción muestra objetos que pasan mensajes; un diagrama de actividades muestra las operaciones que se pasan entre los objetos. La diferencia semántica es sutil, pero tiene como resultado una forma muy diferente de mirar el mundo.

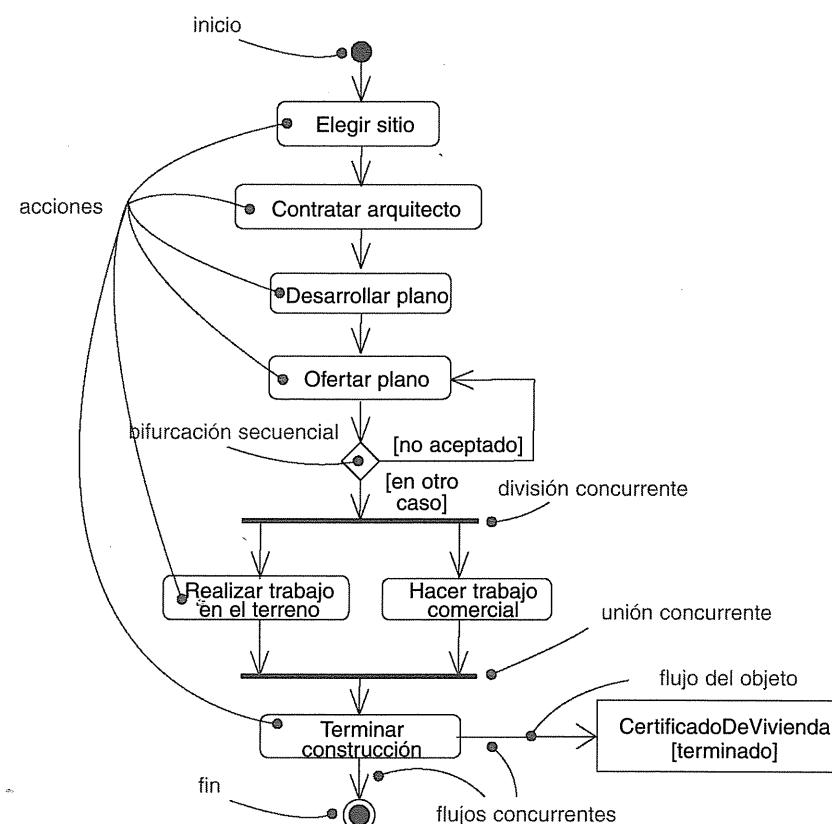


Figura 20.1: Diagrama de actividades.

Términos y conceptos

Un *diagrama de actividades* muestra el flujo de actividades. Una *actividad* es una ejecución no atómica en curso, dentro de una máquina de estados. Las actividades producen finalmente la ejecución de *acciones* individuales, cada una de las cuales

puede producir un cambio en el estado del sistema o la comunicación de mensajes. Las acciones incluyen llamadas a otras operaciones, envío de señales, creación o destrucción de objetos o simples cálculos, como la evaluación de una expresión. Gráficamente, un diagrama de actividades es una colección de nodos y arcos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de actividades es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de actividades de los otros tipos de diagramas es su contenido.

Los estados, las transiciones y las máquinas de estados se discuten en el Capítulo 22; los objetos se discuten en el Capítulo 13.

Contenidos

Normalmente, los diagramas de actividades contienen:

- Acciones.
- Nodos de actividad.
- Flujos.
- Objetos valor.

Al igual que el resto de los diagramas, los diagramas de actividades pueden contener notas y restricciones.

Los atributos y las operaciones se discuten en los Capítulos 4 y 9; las señales se discuten en el Capítulo 21; la creación y la destrucción de objetos se discuten en el Capítulo 16; los estados y las máquinas de estados se discuten en el Capítulo 22.

Acciones y nodos de actividad

En el flujo de control modelado por un diagrama de actividades suceden cosas. Por ejemplo, se podría evaluar una expresión que estableciera el valor de un atributo o que devolviera algún valor. También se podría invocar una operación sobre un objeto, enviar una señal a un objeto o incluso crear o destruir un objeto. Estas computaciones ejecutables y atómicas se llaman acciones. Como se muestra en la Figura 20.2, una acción se representa con un rectángulo con los laterales redondeados. Dentro de esa figura se puede escribir cualquier expresión.

Nota: UML no especifica el lenguaje de esas expresiones. De forma abstracta, se podría utilizar texto estructurado; de forma más precisa, se podría utilizar la sintaxis y la semántica de un lenguaje de programación específico.

El modelado del tiempo y el espacio se discute en el Capítulo 24.

Las acciones no se pueden descomponer. Además, las acciones son atómicas, lo que significa que pueden ocurrir eventos, pero el comportamiento interno de la acción no es visible. No se puede ejecutar una parte de una acción; o se ejecuta completamente o no se ejecuta. Por último, se considera generalmente que la ejecución de una acción lleva un tiempo insignificante, aunque algunas acciones pueden tener una duración sustancial.

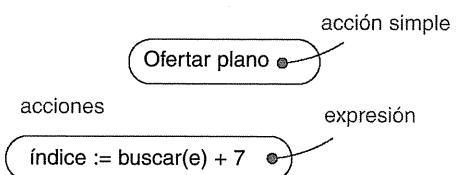


Figura 20.2: Acciones.

Nota: En el mundo real, por supuesto, cada computación conlleva algo de tiempo y espacio. Es importante modelar estas propiedades, especialmente en los sistemas de tiempo real muy exigentes.

Un *nodo de actividad* es una unidad organizativa dentro de una actividad. En general, los nodos de actividad son agrupaciones anidadas de acciones o de otros nodos de actividad. Además, los nodos de actividad tienen una subestructura visible; en general, se considera que se necesita un cierto tiempo para que se completen. Una acción puede verse como un caso especial de un nodo de actividad. Una acción es un nodo de actividad que no puede descomponerse más. Análogamente, un nodo de actividad puede ser visto como un elemento compuesto, cuyo flujo de control se compone de otros nodos de actividad y acciones. Si se entra en los detalles de un nodo de actividad se encontrará otro diagrama de actividades. Como se muestra en la Figura 20.3, no hay distinción en cuanto a la notación de los nodos de actividad y las acciones, excepto que un nodo de actividad puede tener partes adicionales, que normalmente una herramienta de edición mantendrá asociadas al nodo internamente.

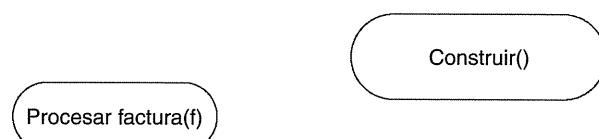


Figura 20.3: Nodos de actividad.

Flujos de control

Cuando se completa una acción o un nodo de actividad, el flujo de control pasa inmediatamente a la siguiente acción o nodo de actividad. Este flujo se especifica con flechas que muestran el camino de control de un nodo de actividad o una acción al siguiente. En UML, un flujo se representa como una flecha des de la acción predecesora hacia la sucesora, sin una etiqueta de evento, como se muestra en la Figura 20.4.

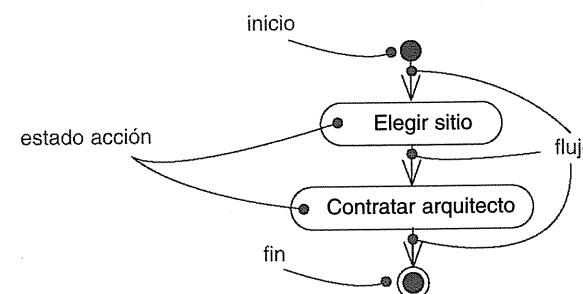


Figura 20.4: Transiciones de finalización.

En realidad, un flujo de control tiene que empezar y parar en algún sitio (a menos, por supuesto, que sea un flujo infinito, en cuyo caso tendrá un principio pero no un final). Por lo tanto, como se aprecia en la figura, se puede especificar el inicio (un círculo relleno) y la terminación (un círculo relleno dentro de una circunferencia).

Bifurcación

Las bifurcaciones son un recurso de notación, equivalente semánticamente a varias transiciones con guardas, como se discute en el Capítulo 22.

Los flujos secuenciales son frecuentes, pero no son el único tipo de camino que se necesita para modelar un flujo de control. Como en los diagramas de flujo, se puede incluir una bifurcación, que especifica caminos alternativos, elegidos en función del valor de una expresión booleana. Como se muestra en la Figura 20.5, una bifurcación se representa con un rombo. Una bifurcación puede tener un flujo de entrada y dos o más de salida. En cada flujo de salida se coloca una expresión booleana, que se evalúa al entrar en la bifurcación. Las guardas de los flujos de salida no deben solaparse (de otro modo, el flujo de control sería ambiguo), pero deberán cubrir todas las posibilidades (de otro modo, el flujo de control se vería interrumpido).

Por comodidad, se puede utilizar la palabra clave `else` para marcar un flujo de salida, la cual representa el camino elegido si ninguna de las otras expresiones de guarda toma el valor *verdadero*.

La bifurcación y la iteración pueden aparecer en los diagramas de interacción, como se discute en el Capítulo 19.

Cuando dos caminos de control vuelven a encontrarse, también se puede utilizar un rombo con dos entradas y una salida. En este caso no se especifican guardas.

Se puede lograr el efecto de la iteración utilizando una acción que establezca el valor de la variable de control de una iteración, otra acción que incremente el valor de la variable y una bifurcación que evalúe si se ha terminado la iteración. UML incluye tipos de nodos para los bucles, pero a menudo éstos se suelen expresar más fácilmente con texto que con gráficos.

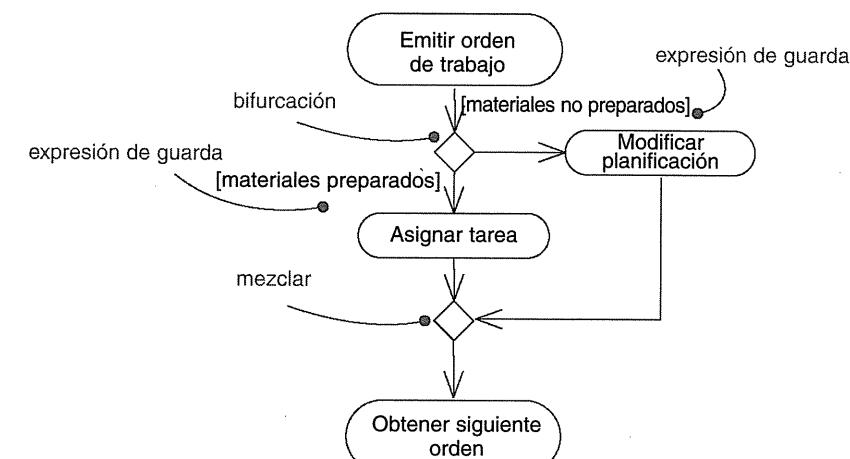


Figura 20.5: Bifurcación

Nota: UML no especifica el lenguaje de estas expresiones. De forma abstracta, se podría utilizar texto estructurado; de forma más concreta, se podría utilizar la sintaxis y la semántica de un lenguaje de programación específico.

Un flujo de control concurrente se encuentra a menudo en el contexto de un objeto activo independiente, el cual se modela normalmente como un proceso o un hilo, como se discute en el Capítulo 23; los nodos se discuten en el Capítulo 27.

División y unión

Los flujos secuenciales y las bifurcaciones son los caminos más utilizados en los diagramas de actividades. Sin embargo, también es posible encontrar flujos concurrentes, especialmente cuando se modelan flujos de trabajo de procesos de negocio. En UML se utiliza una barra de sincronización para especificar la división y unión de estos flujos de control paralelos. Una barra de sincronización se representa como una línea horizontal o vertical ancha.

Por ejemplo, considérense los flujos de control implicados en el control de un dispositivo electrónico que imite la voz y los gestos humanos. Como se muestra en la Figura 20.6, una división representa la separación de un flujo de control sencillo en

dos o más flujos de control concurrentes. Una división puede tener una transición de entrada y dos o más transiciones de salida, cada una de las cuales representa un flujo de control independiente. Después de la división, las actividades asociadas a cada uno de estos caminos continúan en paralelo. Conceptualmente, las actividades de cada uno de estos flujos son verdaderamente concurrentes, aunque en un sistema en ejecución, estos flujos pueden ser realmente concurrentes (en el caso de un sistema instalado en varios nodos) o secuenciales y entrelazados (en el caso de un sistema instalado en un único nodo), dando la ilusión de concurrencia real.

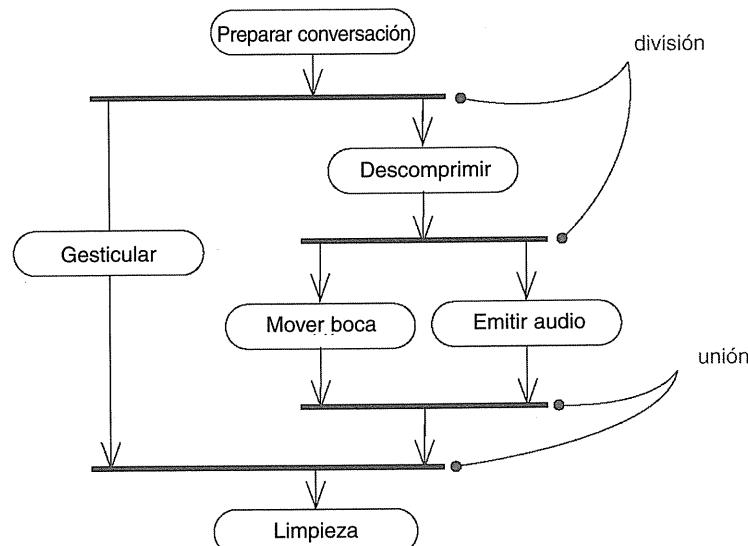


Figura 20.6: División y unión.

Como también se muestra en la figura, una unión representa la sincronización de dos o más flujos de control concurrentes. Una unión puede tener dos o más transiciones de entrada y una transición de salida. Antes de llegar a la unión, las actividades asociadas con cada uno de los caminos continúa en paralelo. En la unión, los flujos concurrentes se sincronizan, es decir, cada uno espera hasta que todos los flujos de entrada han alcanzado la unión, y a partir de ahí continúa un único flujo de control que sale de la unión.

Los objetos activos se discuten en el Capítulo 23; las señales se discuten en el Capítulo 21.

Nota: Las uniones y las divisiones deben equilibrarse, es decir, el número de flujos que parten de una división debe coincidir con el número de flujos que entran en la unión correspondiente. Además, las actividades que se encuentran en flujos de control paralelos pueden comunicarse con las otras por medio de señales. Este estilo de comunicación de procesos secuenciales se llama corrutina. La mayoría de las veces, este estilo de comunicación se modela con objetos activos.

Una calle es un tipo de paquete. Los paquetes se discuten en el Capítulo 12; las clases se discuten en los Capítulos 4 y 9; los procesos y los hilos se discuten en el Capítulo 23.

Calles

Una cosa especialmente útil cuando se modelan flujos de trabajo de procesos del negocio es particionar los nodos de actividad de un diagrama de actividades en grupos, donde cada uno representa la parte de la organización responsable de esas actividades. En UML cada grupo se denomina *calle* porque, visualmente, cada grupo se separa de sus vecinos por una línea vertical continua, como se muestra en la Figura 20.7. Una calle especifica un conjunto de actividades que comparten alguna propiedad organizativa.

Cada calle tiene un nombre único dentro del diagrama. Una calle realmente no tiene una semántica profunda, excepto que puede representar alguna entidad del mundo real, como un departamento de una empresa, por ejemplo. Cada calle representa una responsabilidad de alto nivel de una parte de la actividad global de un diagrama de actividades, y cada calle puede ser implementada en última instancia por una o más clases. En un diagrama de actividades organizado en calles, cada actividad pertenece a una única calle, pero los flujos pueden cruzar las calles.

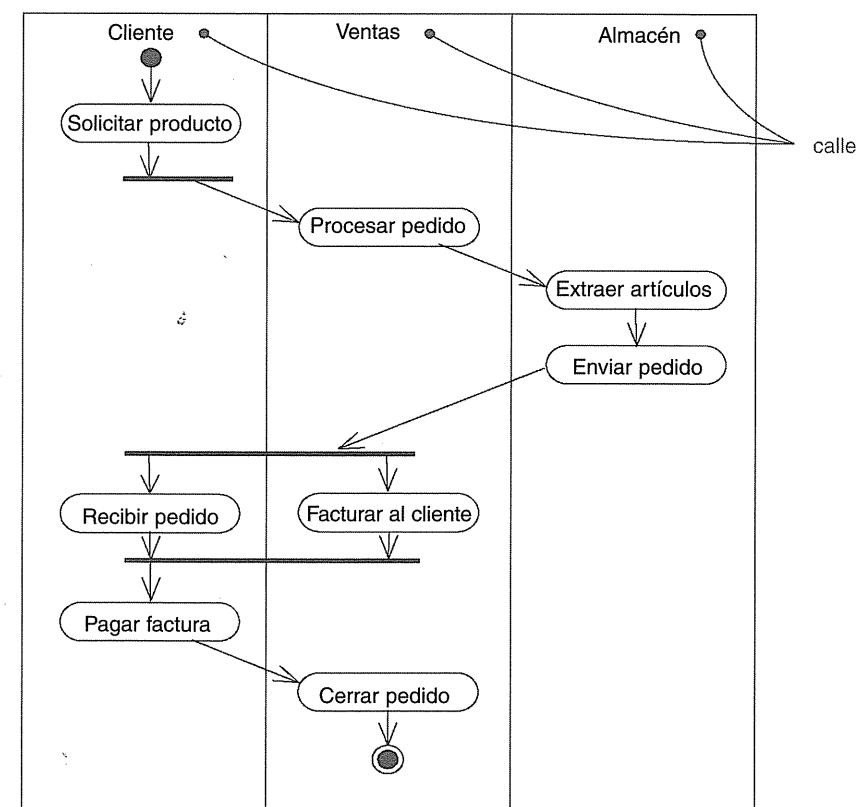


Figura 20.7: Calles.

Nota: Hay una débil conexión entre las calles y los flujos de control concurrentes. Conceptualmente, las actividades de cada calle se consideran, generalmente (aunque no siempre), independientes de las actividades de las calles vecinas. Esto tiene sentido porque, en el mundo real, las partes de las organizaciones que generalmente se corresponden con estas calles son independientes y concurrentes.

Flujo de objetos

Los objetos se discuten en el Capítulo 13; el modelo del vocabulario de un sistema se discute en el Capítulo 4.

Las relaciones de dependencia se discuten en los Capítulos 5 y 10.

Los valores y el estado de un objeto se discuten en el Capítulo 13; los atributos se discuten en los Capítulos 4 y 9.

En el flujo de control asociado a un diagrama de actividades pueden verse involucrados objetos. Por ejemplo, en el flujo de trabajo del procesamiento de un pedido, como el de la figura anterior, el vocabulario del espacio del problema también incluiría clases como Pedido y Factura. Ciertas actividades producirán instancias de esas dos clases (por ejemplo, Procesar pedido creará un objeto Pedido); otras actividades pueden usar o modificar estos objetos (por ejemplo, Enviar pedido cambiará el estado del objeto pedido a completado).

Como se muestra en la Figura 20.8, se pueden especificar los objetos implicados en un diagrama de actividades colocándolos en el diagrama, conectados con una flecha a las acciones que los crean o los consumen.

Esto se denomina flujo de objetos, porque representa el flujo de un valor objeto desde una acción hasta otra. Un flujo de objetos conlleva un flujo de control de forma implícita (¡no podemos ejecutar una acción que requiere un valor sin el valor!), así que no es necesario dibujar un flujo de control entre acciones conectadas por flujos de objetos.

Además de mostrar el flujo de un objeto a través de un diagrama de actividades, también es posible mostrar cómo cambia su estado. Como se muestra en la figura, el estado de un objeto se representa dando al estado un nombre entre corchetes bajo el nombre del objeto.

Regiones de expansión

A menudo, la misma operación debe ejecutarse sobre todos los elementos de un conjunto. Por ejemplo, si un pedido contiene un conjunto de líneas, el manejador del pedido debe ejecutar la misma operación para cada línea: comprobar la disponibilidad del artículo, consultar el precio, comprobar el impuesto aplicable, etcétera. Las operaciones sobre las listas a menudo se modelan como bucles, pero entonces el modelador debe iterar sobre los artículos, extraer uno en cada paso, ejecutar la operación, almacenar el resultado en un array de salida, incre-

mentar el índice y verificar si se ha terminado o no. La mecánica de la ejecución del bucle oscurece el significado real de la operación. Este patrón tan frecuente puede modelarse directamente con una *región de expansión*.

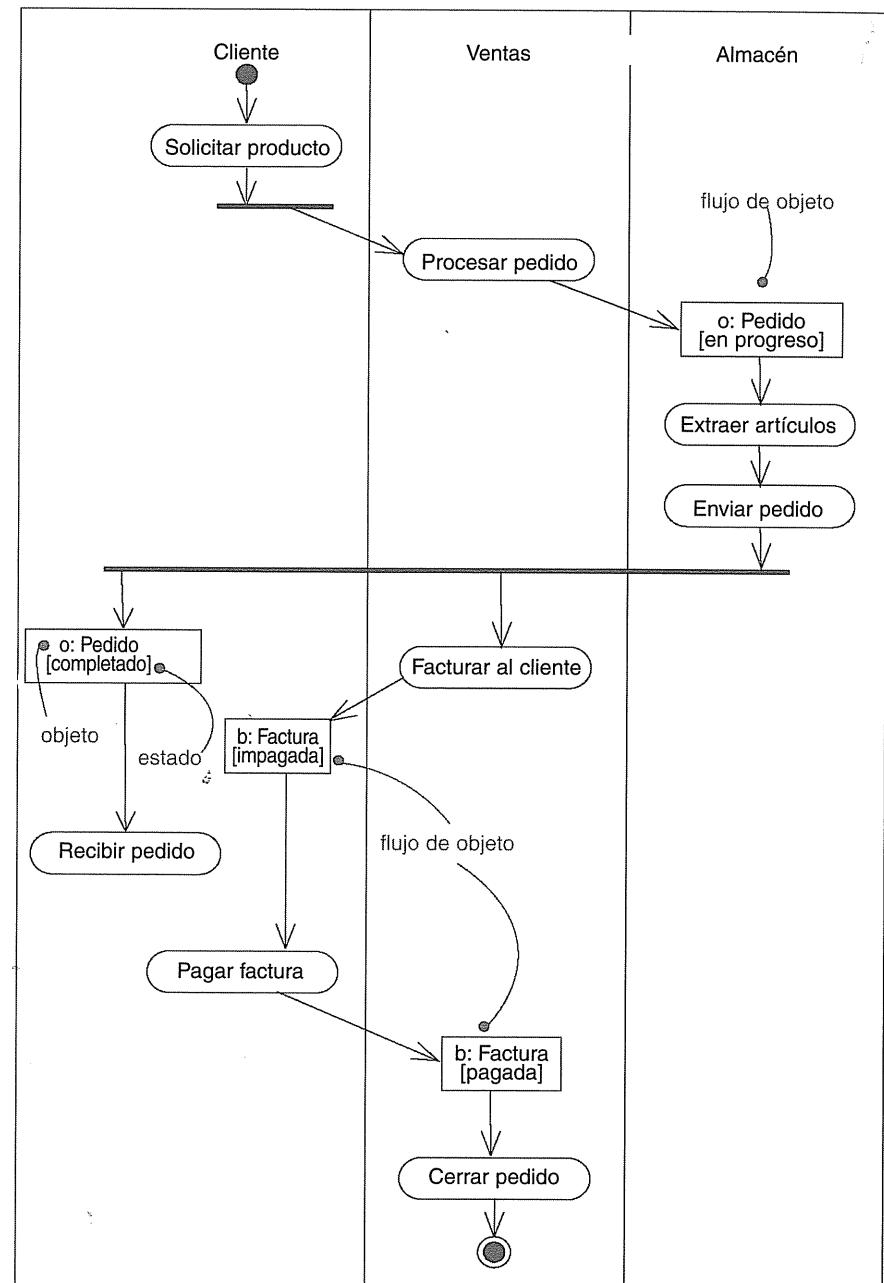


Figura 20.8: Flujo de objetos.

Una región de expansión representa un fragmento de un modelo de actividades que se ejecuta sobre los elementos de una lista o de un conjunto. La región se representa en un diagrama de actividades dibujando una línea discontinua alrededor de una región de un diagrama. Las entradas a la región y las salidas de ella son colecciones de valores, como las líneas de artículos de un pedido. Las colecciones de entrada y salida se representan como una fila de pequeños recuadros unidos (para sugerir un array de valores). Cuando un valor array llega a la colección de entrada de una región de expansión desde el resto del modelo de actividades, aquél se divide en valores individuales. La región de expansión se ejecuta una vez para cada uno de los elementos del array. No es necesario modelar la iteración, ya que está contenida de forma implícita en la región de expansión. Las diferentes ejecuciones pueden llevarse a cabo de manera concurrente, si esto es posible. Cuando cada ejecución de la región de expansión termina, su valor de salida (si lo hay) se introduce en un array en el mismo orden que el valor de entrada correspondiente. En otras palabras, la región de expansión ejecuta una operación “para todo” (*forall*) sobre los elementos del array para crear un nuevo array.

En el caso más simple, una región de expansión tiene un array de entrada y otro de salida, pero podría haber uno o más arrays de entrada y cero o más arrays de salida. Todos los arrays deben ser del mismo tamaño, pero no tienen por qué contener el mismo tipo de valores. Los valores de las mismas posiciones se ejecutan juntos para producir valores de salida en la misma posición. La región podría no tener valores de salida si todas las operaciones se ejecutaran directamente sobre los elementos del array por sus efectos colaterales.

Las regiones de expansión permiten que las operaciones sobre colecciones y las operaciones sobre elementos individuales se representen en el mismo diagrama, sin tener que representar toda la complejidad detallada de la iteración.

La Figura 20.9 muestra un ejemplo de una región de expansión. En el cuerpo principal del diagrama se recibe un pedido. Esto produce un valor de tipo Pedido, que consiste en un array de valores LíneaDeArtículo. El valor Pedido es la entrada de una región de expansión. Cada ejecución de la región de expansión trabaja sobre un elemento de la colección Pedido. Por tanto, dentro de la región, el tipo de valor de entrada se corresponde con un elemento del array del Pedido, es decir, una LíneaDeArtículo. La actividad de la región de expansión se divide en dos acciones: una acción busca el Producto y lo añade al envío, y la otra acción calcula el coste de ese elemento. No es necesario procesar cada LíneaDeArtículo en orden; las diferentes ejecuciones de la región de expansión pueden funcionar concurrentemente. Cuando

todas las ejecuciones de la región de expansión han terminado, los artículos se agrupan en un Envío (una colección de Productos) y los importes se agrupan en una Factura (una colección de valores Importe). El valor Envío es la entrada de la acción EnviarPedido y el valor Factura es la entrada de la acción EnviarFactura.

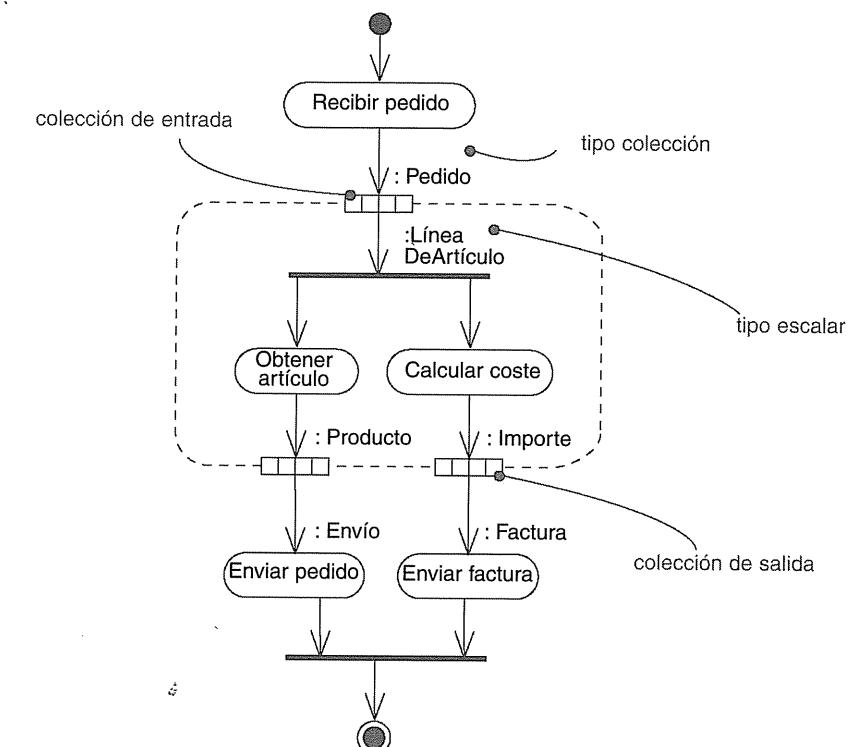


Figura 20.9: Región de expansión.

Usos comunes

Los diagramas de actividades se utilizan para modelar los aspectos dinámicos de un sistema. Estos aspectos dinámicos pueden involucrar la actividad de cualquier tipo de abstracción en cualquier vista de la arquitectura de un sistema, incluyendo clases (las cuales pueden ser clases activas), interfaces, componentes y nodos.

Cuando se utiliza un diagrama de actividades para modelar algún aspecto dinámico de un sistema, se puede hacer en el contexto de casi cualquier elemento de modelado. Sin embargo, normalmente se usan los diagramas de actividades en

el contexto del sistema global, un subsistema, una operación o una clase. También se pueden asociar diagramas de actividades a los casos de uso (para modelar un escenario) y a las colaboraciones (para modelar los aspectos dinámicos de una sociedad de objetos).

Cuando se modelan los aspectos dinámicos de un sistema, normalmente se utilizan los diagramas de actividades de dos formas.

1. Para modelar un flujo de trabajo.

Para ello se hace hincapié en las actividades, tal y como son vistas por los actores que colaboran con el sistema. A menudo, en el entorno de los sistemas con gran cantidad de software, existen flujos de trabajo y se utilizan para visualizar, especificar, construir y documentar procesos de negocio que implican al sistema en desarrollo. En este uso de los diagramas de actividades, es particularmente importante el modelado de los flujos de objetos.

2. Para modelar una operación.

Para ello se utilizan los diagramas de actividades como diagramas de flujo, para modelar los detalles de una computación. En este uso de los diagramas de actividades, es particularmente importante el modelado de la bifurcación, la división y la unión. El contexto de un diagrama de actividades utilizado con esta finalidad incluye los parámetros de la operación, así como sus objetos locales.

Técnicas comunes de modelado

Modelado de un flujo de trabajo (*Workflow*)

El modelado del contexto de un sistema se discute en el Capítulo 18.

Ningún sistema con gran cantidad de software se encuentra aislado; siempre hay un contexto en el que reside el sistema, y ese contexto siempre incluye actores que interactúan con el sistema. Es posible encontrar sistemas automáticos que trabajen en el contexto de procesos de negocio de más alto nivel, como es el caso del software de empresa para misiones críticas. Estos procesos de negocio son tipos de flujos de trabajo porque representan el flujo de trabajo y objetos a través del negocio. Por ejemplo, en un negocio de venta, habrá algunos sistemas automáticos (por ejemplo, sistemas de punto de venta que interactúan con los

sistemas de marketing y almacén), así como sistemas formados por personas (gente que trabaja en cada terminal de venta, además de los departamentos de televendas, marketing, compras y logística). Los diagramas de actividades pueden utilizarse para modelar los procesos de negocio en los que colaboran estos sistemas automáticos y humanos.

Para modelar un flujo de trabajo:

El modelado del vocabulario de un sistema se discute en el Capítulo 4; las precondiciones y las postcondiciones se discuten en el Capítulo 4.

- Hay que establecer un centro de interés para el flujo de trabajo. Para los sistemas no triviales es imposible mostrar todos los flujos de trabajo interesantes en un diagrama.
- Hay que seleccionar los objetos del negocio que tienen las responsabilidades de más alto nivel en cada parte del flujo de trabajo global. Éstos pueden ser cosas concretas del vocabulario del sistema, o pueden ser más abstractos. En cualquier caso, debe crearse una calle para cada objeto del negocio importante.
- Hay que identificar las precondiciones del estado inicial del flujo de trabajo y las postcondiciones del estado final. Esto es importante para ayudar a modelar los límites del flujo de trabajo.
- Comenzando por el estado inicial del flujo de trabajo, hay que especificar las actividades y acciones que tienen lugar a lo largo del tiempo, y deben representarse en el diagrama de actividades.
- Hay que modelar las acciones complicadas o los conjuntos de acciones que aparezcan muchas veces como llamadas a diagramas de actividades aparte.
- Hay que representar los flujos que conectan las acciones y los nodos de actividad. Debe comenzarse con los flujos secuenciales del flujo de trabajo, a continuación hay que considerar las bifurcaciones y, por último, hay que tener en cuenta las divisiones y las uniones.
- Si el flujo involucra objetos importantes, hay que representarlos también en el diagrama de actividades. Hay que mostrar sus valores y estado cuando cambien, si es necesario para comunicar el propósito del flujo de objetos.

Por ejemplo, la Figura 20.10 muestra un diagrama de actividades de un negocio de venta, que especifica el flujo de control que se desarrolla cuando un cliente devuelve un artículo de un pedido por correo. El proceso comienza con la acción *Solicitar devolución del Cliente* y después pasa a través de *Televentas* (*Obtener número de devolución*), vuelve al *Cliente*

(Enviar artículo), después pasa al Almacén (Recibir artículo y Recolocar artículo) y por último acaba en Contabilidad (Actualizar cuenta). Como indica el diagrama, un objeto significativo (una instancia de Artículo) también fluye en el proceso, cambiando del estado devuelto al estado disponible.

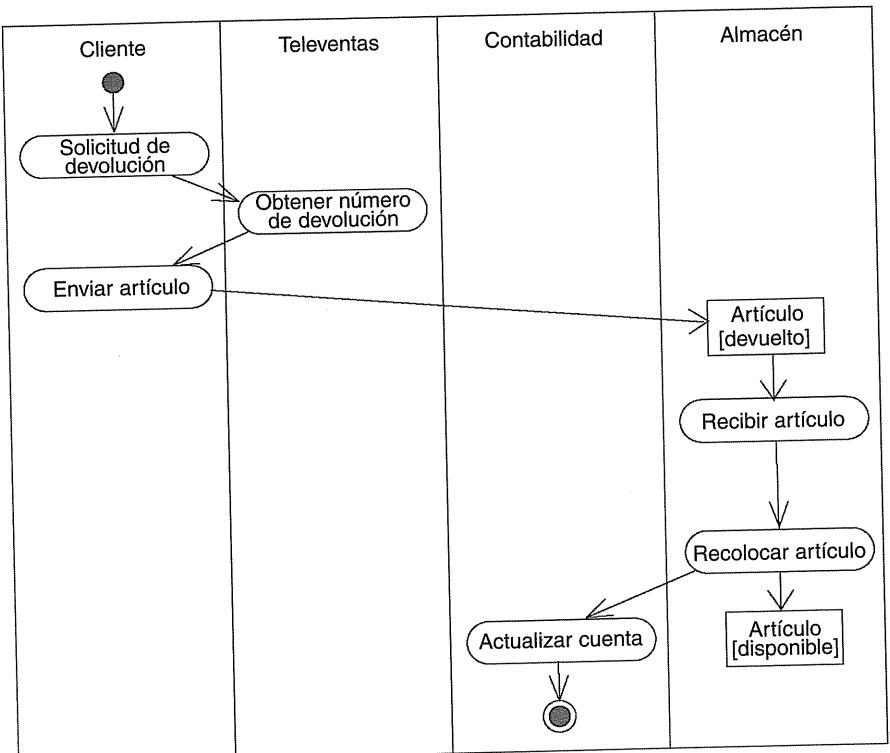


Figura 20.10: Modelado de un flujo de trabajo.

Nota: Los flujos de trabajo suelen ser procesos de negocio la mayoría de las veces, aunque no siempre. Por ejemplo, los diagramas de actividades se pueden utilizar para especificar procesos de desarrollo de software, como el proceso de gestión de configuraciones. Además, los diagramas de actividades se pueden utilizar para modelar sistemas no software, como el flujo de pacientes a través de un sistema de atención sanitaria.

En este ejemplo no existen bifurcaciones, divisiones ni uniones. Estas características aparecen en flujos de trabajo más complejos.

Modelado de una operación

Las clases y las operaciones se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28; las precondiciones, las postcondiciones y los invariantes se discuten en el Capítulo 9; las clases activas se discuten en el Capítulo 23.

Un diagrama de actividades puede asociarse a cualquier elemento de modelado para visualizar, especificar, construir y documentar el comportamiento de ese elemento. Los diagramas de actividades pueden asociarse a clases, interfaces, componentes, nodos, casos de uso y colaboraciones. Los elementos a los que más frecuentemente se asocia un diagrama de actividades son las operaciones.

Cuando se utiliza de esta forma, un diagrama de actividades es simplemente un diagrama de flujo de las acciones de una operación. La ventaja principal de un diagrama de actividades es que todos los elementos del diagrama están ligados semánticamente a un modelo subyacente con bastante riqueza expresiva. Por ejemplo, se puede comprobar si el tipo de cualquier operación o señal referida por una acción es compatible con la clase del objeto receptor.

Para modelar una operación:

- Hay que reunir las abstracciones implicadas en la operación. Esto incluye los parámetros de la operación (incluyendo el tipo de retorno, si lo hay), los atributos de la clase a la que pertenece y ciertas clases vecinas.
- Hay que identificar las precondiciones en el estado inicial de la operación y las postcondiciones en el estado final. También hay que identificar cuálquier invariante de la clase a la que pertenece que deba mantenerse durante la ejecución de la operación.
- Hay que especificar las actividades y acciones que tienen lugar a lo largo de la ejecución, comenzando por el estado inicial de la operación, y representarlas en el diagrama de actividades como nodos de actividad o acciones.
- Hay que usar bifurcaciones cuando sea necesario especificar caminos alternativos e iteraciones.
- Hay que usar divisiones y uniones cuando sea necesario especificar flujos paralelos de control, sólo si la operación se encuentra en una clase activa.

Si una operación conlleva la interacción de una sociedad de objetos, también se puede modelar la realización de esa operación utilizando colaboraciones, como se discute en el Capítulo 28.

Por ejemplo, en el contexto de la clase Línea, la Figura 20.11 muestra un diagrama de actividades que especifica el algoritmo de la operación intersección, cuya firma incluye un parámetro (línea, de la clase

Línea) y un valor de retorno (de la clase Punto). La clase Línea tiene dos atributos interesantes: pendiente (que contiene la pendiente de la línea) y delta (que contiene el desplazamiento de la línea respecto del origen).

El algoritmo de la operación es sencillo, como se muestra en el siguiente diagrama de actividades. En primer lugar, hay una guarda que comprueba si la pendiente de la línea actual es la misma que la pendiente del parámetro línea. Si es así, las líneas no se cortan, y se devuelve el Punto en $(0, 0)$. En otro caso, la operación calcula el valor de x para el punto de intersección y, a continuación, el valor de y ; x e y son objetos locales a la operación. Por último, se devuelve un Punto en (x, y) .

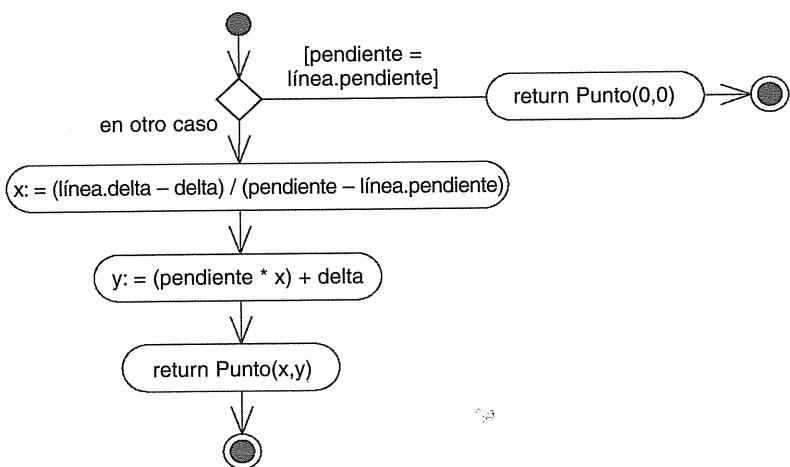


Figura 20.11: Modelado de una operación.

Nota: La utilización de los diagramas de actividades para hacer los diagramas de flujo de una operación es casi hacer de UML un lenguaje de programación visual. Se pueden hacer los diagramas de flujo de todas las operaciones, pero en la práctica no será así. Normalmente es más directo escribir el cuerpo de una operación en un lenguaje de programación. Los diagramas de actividades se utilizarán para modelar una operación cuando el comportamiento sea complejo y, por tanto, difícil de comprender mediante la simple observación del código. La observación de un diagrama de flujo revela cosas sobre el algoritmo que no se podrían haber visto simplemente mirando el código.

Ingeniería directa e inversa

Se puede hacer *ingeniería directa* (creación de código a partir de un modelo) con los diagramas de actividades, especialmente si el contexto del diagrama es una operación. Por ejemplo, con el diagrama de actividades anterior, una herramienta de ingeniería directa podría generar el siguiente código C++ para la operación intersección.

```

Punto Linea::intersección(Linea linea) {
    if (pendiente == linea.pendiente) return
        Punto(0,0);
    int x = (linea.delta - delta) /
        (pendiente - linea.pendiente);
    int y = (pendiente * x) + delta;
    return Punto(x, y);
}
  
```

La declaración de las dos variables locales denota un poco de ingenio. Una herramienta menos sofisticada podría declarar primero las dos variables y luego asignarles valores.

La *ingeniería inversa* (creación de un modelo a partir de código) también es posible con los diagramas de actividades, especialmente si el contexto del código es el cuerpo de una operación. En particular, el diagrama anterior podría haber sido generado a partir de la implementación de la clase Línea.

Más interesante que la ingeniería inversa de un modelo a partir de código es la animación de un modelo durante la ejecución del sistema desarrollado. Por ejemplo, dado el diagrama anterior, una herramienta podría animar las acciones del diagrama mientras se van sucediendo en un sistema en ejecución. Mejor aun, teniendo esta herramienta bajo el control de un depurador, se podría controlar la velocidad de ejecución, introduciendo posiblemente puntos de ruptura para detener la acción en puntos de interés, con el fin de examinar los valores de los atributos de los objetos individuales.

Sugerencias y consejos

Cuando se crean diagramas de actividades en UML, hay que recordar que los diagramas de actividades son proyecciones del modelo de los aspectos dinámicos de un sistema. Un único diagrama de actividades no puede capturar todo

acerca de la dinámica de un sistema. En lugar de ello, se utilizarán varios diagramas de actividades para modelar la dinámica de un flujo de trabajo o una operación.

Un diagrama de actividades bien estructurado:

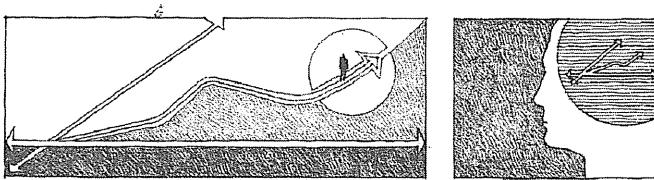
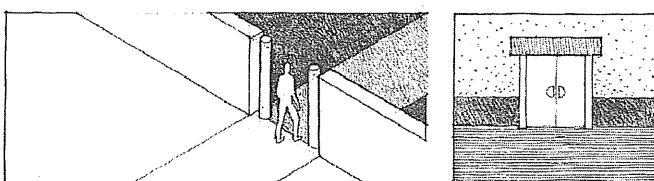
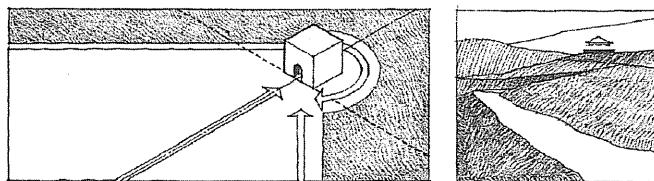
- Se ocupa de modelar un aspecto de la dinámica de un sistema.
- Contiene sólo aquellos elementos esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con su nivel de abstracción; sólo se muestran los adornos que son esenciales para su comprensión.
- No es tan minimalista que no ofrezca información al lector sobre los aspectos importantes de la semántica.

Cuando se dibuje un diagrama de actividades:

- Hay que darle un nombre que comunique su propósito.
- Hay que comenzar por modelar el flujo principal. Las bifurcaciones, la concurrencia y los flujos de objetos se deben considerar como secundarios, posiblemente en diagramas separados.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre las características importantes del diagrama.



Parte 5 MODELADO AVANZADO DEL COMPORTAMIENTO



En este capítulo

- Eventos de señal, eventos de llamada, eventos de tiempo y eventos de cambio.
- Modelado de una familia de señales.
- Modelado de excepciones.
- Gestión de eventos en objetos activos y pasivos.

En el mundo real suceden cosas. No sólo suceden cosas, sino que muchas de ellas suceden a la vez, y en los momentos más inesperados. Las “cosas que suceden” se llaman eventos, y cada uno representa la especificación de un acontecimiento significativo ubicado en el tiempo y en el espacio.

En el contexto de las máquinas de estados, los eventos se utilizan para modelar la aparición de un estímulo que puede disparar una transición de estado. Los eventos pueden incluir señales, llamadas, el paso del tiempo o un cambio en el estado.

Los eventos pueden ser síncronos o asíncronos, así que el modelado de eventos se incluye en el modelado de procesos y hilos.

Introducción

Un sistema totalmente estático no es nada interesante, porque no sucede nada. Todos los sistemas reales tienen una dimensión dinámica, y esta dinámica se activa por las cosas que ocurren externa o internamente. En un cajero automático, la acción se inicia cuando el usuario pulsa un botón para iniciar una transacción. En un robot autónomo, la acción se inicia cuando el robot tropieza con un objeto. En un dispositivo enrutador, la acción se inicia con la detección de una

sobrecarga de los buffers de mensajes. En una planta química, la acción comienza cuando ha pasado el tiempo necesario para una reacción química.

En UML, cada cosa que sucede se modela como un evento. Un evento es la especificación de un acontecimiento significativo ubicado en el tiempo y en el espacio. Una señal, el paso del tiempo y el cambio de estado son eventos asíncronos, que representan sucesos que pueden acaecer en cualquier instante. Las llamadas son normalmente eventos síncronos, que representan la invocación de una operación.

UML proporciona una representación gráfica para los eventos, como se muestra en la Figura 21.1. Esta notación permite visualizar la declaración de eventos (como la señal Colgar), así como el uso de eventos para disparar una transición de estado (como la señal Colgar, que causa una transición del estado Activo al estado Inactivo de un teléfono, así como la ejecución de la acción cortarConexión).

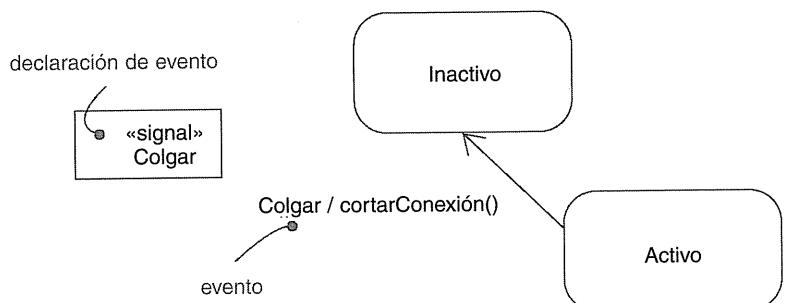


Figura 21.1: Eventos.

Términos y conceptos

Un *evento* es la especificación de un acontecimiento significativo ubicado en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede disparar una transición de estado. Una *señal* es un tipo de evento que representa la especificación de un estímulo asíncrono que se transmite entre instancias.

Los actores se discuten en el Capítulo 17; los sistemas se discuten en el Capítulo 32.

Tipos de eventos

Los eventos pueden ser externos o internos. Los eventos externos son aquellos que fluyen entre el sistema y sus actores. Por ejemplo, la pulsación de un

botón y una interrupción de un sensor de colisiones son ejemplos de eventos externos. Los eventos internos son aquellos que fluyen entre los objetos del sistema. Una excepción de desbordamiento (*overflow*) es un ejemplo de un evento interno.

La creación y destrucción de objetos también son tipos de señales, como se discute en el Capítulo 16.

En UML se pueden modelar cuatro tipos de eventos: señales, llamadas, el paso del tiempo y el cambio de estado.

Señales

Un mensaje es un objeto con nombre que es enviado asíncronamente por un objeto y recibido por otro. Una señal es un clasificador para algunos mensajes; la señal representa un tipo de mensaje:

Las clases se discuten en los Capítulos 4 y 9; la generalización se discute en los Capítulos 5 y 10.

Las señales tienen mucho en común con las clases normales. Por ejemplo, las señales pueden tener instancias, aunque normalmente no es necesario modelarlas explícitamente. También pueden existir relaciones de generalización entre señales, lo que permite modelar jerarquías de eventos, algunos de los cuales son genéricos (por ejemplo, la señal FalloDeRed) y otros son específicos (por ejemplo, una especialización de FalloDeRed llamada FalloServidorAlmacén). Al igual que las clases, las señales también pueden tener atributos y operaciones. Antes de ser enviada por un objeto o después de ser recibida por otro, una señal es simplemente un objeto normal.

Las máquinas de estados se discuten en el Capítulo 22; las interacciones se discuten en el Capítulo 16; las interfaces se discuten en el Capítulo 11; las dependencias se discuten en el Capítulo 5; los estereotipos se discuten en el Capítulo 6.

Nota: Los atributos de una señal sirven como sus parámetros. Por ejemplo, al enviar una señal como Colisión, también se puede especificar un valor para sus atributos como parámetros, como Colisión (5.3).

Una señal puede enviarse como la acción de una transición en una máquina de estados. También puede modelarse como un mensaje entre dos roles en una interacción. La ejecución de un método también puede enviar señales. De hecho, cuando se modela una clase o una interfaz, una parte importante de la especificación del comportamiento de ese elemento consiste en especificar las señales que pueden enviar sus operaciones.

En UML, como se muestra en la Figura 21.2, las señales se modelan como clases estereotipadas. Se puede utilizar una dependencia, estereotipada como send, para indicar que una operación envía una señal particular.

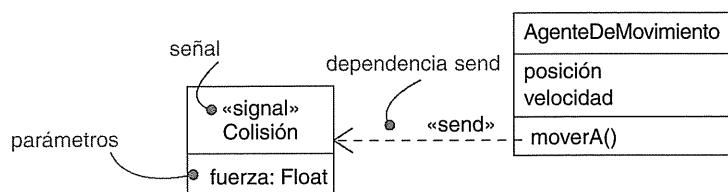


Figura 21.2: Señales.

Eventos de llamada

Las máquinas de estados se discuten en el Capítulo 22.

Así como un evento de señal representa la ocurrencia de una señal, un evento de llamada representa la recepción, por parte de un objeto, de una solicitud de invocación de una de sus operaciones. Un evento de llamada puede disparar una transición en una máquina de estados o puede invocar un método en el objeto destinatario. La elección se especifica en la definición de la operación en la clase.

Mientras que una señal es un evento asíncrono, un evento de llamada es, en general, síncrono. Esto significa que cuando un objeto invoca una operación sobre otro objeto que tiene una máquina de estados, el control pasa del emisor al receptor, el evento dispara la transición, la operación acaba, el receptor pasa a un nuevo estado y el control regresa al emisor. En aquellos casos en los que el emisor no necesita esperar una respuesta, la llamada puede especificarse como asíncrona.

Como se muestra en la Figura 21.3, el modelado de un evento de llamada es indistinguible del modelado de un evento de señal. En ambos casos se muestra el evento, junto con sus parámetros, como el disparador de una transición.

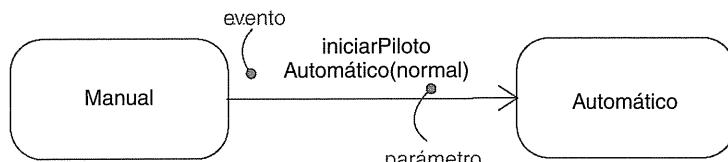


Figura 21.3: Eventos de llamada.

Nota: Aunque no hay señales visuales para distinguir un evento de señal de un evento de llamada, la diferencia está clara en la especificación del modelo. Por supuesto, el receptor de un evento conoce-

rá la diferencia (declarando la operación en su lista de operaciones). Normalmente, una señal será manejada por su máquina de estados, y un evento de llamada será manejado por un método. Se pueden utilizar herramientas para navegar desde el evento hasta la señal o la operación.

Eventos de tiempo y cambio

Un evento de tiempo es un evento que representa el paso del tiempo. Como muestra la Figura 21.4, en UML se modela un evento de tiempo con la palabra clave *after* seguida de alguna expresión que se evalúa para producir algún período de tiempo. Tales expresiones pueden ser sencillas (por ejemplo, *after 2 segundos*) o complejas (por ejemplo, *after 1 ms desde la salida de Inactivo*). A menos que se especifique explícitamente, el período de tiempo asociado a una expresión de este tipo comienza en el instante a partir del cual se entra en el estado actual. Para indicar un evento de tiempo que ocurre en un tiempo absoluto, se utiliza la palabra clave *at*. Por ejemplo, el evento de tiempo *at (1 Enero 2005, 1200 UT)* especifica un evento que ocurre al mediodía en Tiempo Universal el día de año nuevo de 2005.

Un evento de cambio es un evento que representa un cambio de estado o el cumplimiento de alguna condición. Como se muestra en la Figura 21.4, en UML se modela un evento de cambio con la palabra clave *when* seguida de alguna expresión booleana. Se pueden emplear esas expresiones para la evaluación continua de una expresión (por ejemplo, *when altitud < 1000*).

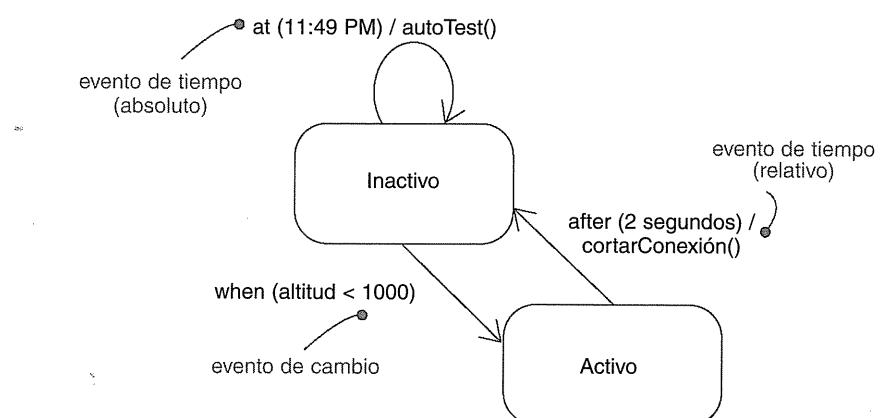


Figura 21.4: Eventos de tiempo y de cambio.

Un evento de cambio ocurre una vez cuando el valor de la condición cambia de falso a verdadero. El evento no ocurre cuando el valor de la condición cambia de verdadero a falso. El evento no se repite mientras la condición sea cierta.

Nota: Aunque un evento de cambio modela una condición que se comprueba continuamente, normalmente se puede analizar la situación para detectar cuándo comprobar la condición en puntos discretos del tiempo.

Envío y recepción de eventos

Los procesos e hilos se discuten en el Capítulo 23.

Las instancias se discuten en el Capítulo 13.

Las máquinas de estados se discuten en el Capítulo 22; los objetos activos se discuten en el Capítulo 23.

Los eventos de señal y los eventos de llamada implican al menos a dos objetos: el objeto que envía la señal o invoca la operación, y el objeto al que se dirige el evento. Como las señales son asíncronas, y las llamadas asíncronas son también señales, la semántica de los eventos se mezcla con la semántica de los objetos activos y pasivos.

Cualquier instancia de cualquier clase puede enviar una señal o invocar una operación de un objeto receptor. Cuando se envía una señal, el objeto emisor continúa con su flujo de control después de haberla enviado, sin esperar ningún retorno del receptor. Por ejemplo, si un actor al interactuar con un cajero automático envía la señal botónPulsado, el actor puede continuar por su cuenta independientemente del sistema al que se envió la señal. Por el contrario, cuando un objeto invoca una operación, el emisor llama a la operación y espera al receptor. Por ejemplo, en un sistema de ventas, una instancia de la clase Comerciante podría invocar la operación confirmarTransacción sobre alguna instancia de la clase Venta, lo cual afectará al estado del objeto Venta. Si esta llamada es síncrona, el objeto Comerciante esperará hasta que termine la operación.

Nota: En algunas situaciones, es posible que se desee mostrar el envío de una señal por parte de un objeto a un conjunto de objetos (*multicasting*) o a cualquier objeto del sistema que pueda estar esuchando (*broadcasting*). Para modelar el *multicasting*, se representará un objeto que envía una señal a una colección que contendrá un conjunto de receptores. Para modelar el *broadcasting*, se mostrará un objeto que envía una señal a otro objeto que representa el sistema global.

Cualquier instancia de cualquier clase puede recibir un evento de llamada o una señal. Si éste es un evento de llamada síncrono, entonces el emisor y el

receptor están sincronizados por *rendezvous* durante la duración de la operación. Esto significa que el flujo de control del emisor se bloquea hasta que termina la ejecución de la operación. Si es una señal, el emisor y el receptor no se sincronizan: el emisor envía la señal pero no espera una respuesta del receptor. En cualquier caso, este evento puede perderse (si no se especifica una respuesta al evento), o puede activar la máquina de estados del receptor (si dispone de una), o simplemente puede invocar una llamada normal a un método.

Nota: Una llamada puede ser asíncrona. En ese caso, el invocador continúa inmediatamente después de haber emitido la llamada. La transmisión del mensaje al receptor y su ejecución ocurren de manera concurrente con la siguiente ejecución del invocador. Cuando la ejecución del método se completa, ésta simplemente termina. Si el método intenta devolver algún valor, éste no se tiene en cuenta.

Las operaciones se discuten en el Capítulo 4; los compartimentos extra de las clases se discuten en el Capítulo 4.

Las interfaces se discuten en el Capítulo 11; las operaciones asíncronas se discuten en el Capítulo 23.

En UML, los eventos de llamada que puede recibir un objeto se modelan como operaciones sobre la clase del objeto. En UML, las señales con nombre que puede recibir un objeto se modelan designándolas en un compartimento extra de la clase, como se muestra en la Figura 21.5.

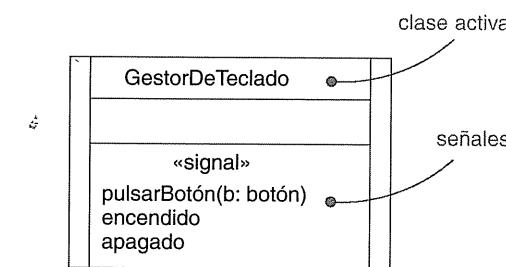


Figura 21.5: Señales y clases activas.

Nota: También se pueden asociar señales con nombre a una interfaz de la misma manera. En cualquier caso, las señales que se listan en un compartimento extra no son la declaración de la señal, sino sólo el uso de ella.

Técnicas comunes de modelado

Modelado de una familia de señales

La generalización se discute en los Capítulo 5 y 10.

En la mayoría de los sistemas dirigidos por eventos, los eventos de señal son jerárquicos. Por ejemplo, un robot autónomo podría distinguir entre señales externas, como Colisión, e internas, como FalloHardware. No obstante, las señales internas y externas no tienen por qué ser disjuntas. Incluso dentro de estas dos grandes clasificaciones se podrían encontrar especializaciones. Por ejemplo, las señales FalloHardware podrían especializarse en FalloBatería y FalloMecánico. Incluso éstas podrían especializarse más aún, como AtascoDelMotor, un tipo de FalloMecánico.

Las máquinas de estados se discuten en el Capítulo 22.

Cuando se modelan jerarquías de señales de esta forma, se pueden especificar eventos polimórficos. Por ejemplo, considérese una máquina de estados con una transición activada sólo por la recepción de un AtascoDelMotor. Al ser una señal hoja en esta jerarquía, la transición sólo puede ser disparada por esta señal, así que no es polimórfica. Por el contrario, supongamos que se ha modelado la máquina de estados con una transición disparada por la recepción de un FalloHardware. En este caso, la transición es polimórfica, y puede ser disparada por un FalloHardware o por cualquiera de sus especializaciones, incluyendo FalloBatería, FalloMecánico y AtascoDelMotor.

Para modelar una familia de señales:

- Hay que considerar todos los tipos diferentes de señales a las que puede responder un conjunto de objetos activos.
- Hay que buscar los tipos frecuentes de señales y colocarlos en una jerarquía de generalización/especialización por medio de la herencia. Hay que colocar arriba las más generales y debajo las más especializadas.
- Hay que buscar, en las máquinas de estado de los objetos activos, donde es posible utilizar el polimorfismo. Donde se encuentre polimorfismo, hay que ajustar la jerarquía si es necesario, introduciendo señales abstractas intermedias.

Las clases abstractas se discuten en los Capítulos 5 y 9.

La Figura 21.6 modela una familia de señales que pueden ser manejadas por un robot autónomo. Nótese que la señal raíz (SeñalDelRobot) es abstrac-

ta, es decir, no puede tener instancias directas. Esta señal tiene dos especializaciones concretas inmediatas (Colisión y FalloHardware), una de las cuales (FalloHardware) aún está más especializada. Nótese que la señal Colisión tiene un parámetro.

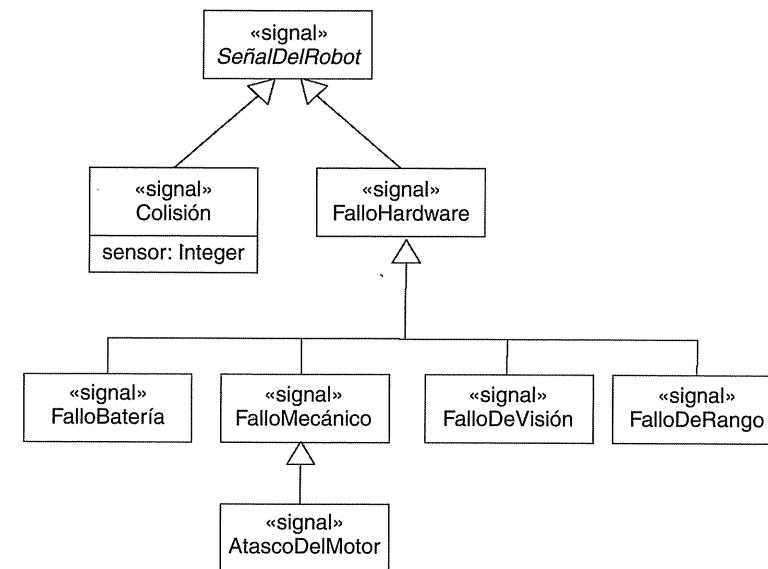


Figura 21.6: Modelado de una familia de señales.

Modelado de situaciones anormales

Las clases se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11; los estereotipos se discuten en el Capítulo 6.

Una cuestión importante cuando se visualiza, especifica y documenta el comportamiento de una clase o una interfaz es especificar las situaciones anormales que pueden producir sus operaciones. Si manejamos una clase o una interfaz, las operaciones que se pueden invocar estarán claras, pero las situaciones anormales que puede producir cada operación no estarán tan claras a menos que se modelen explícitamente.

En UML, las situaciones anormales son un tipo adicional de eventos, que pueden modelarse como señales. Se pueden asociar eventos de error a las especificaciones de las operaciones. El modelado de las excepciones es, en cierto modo, lo contrario de modelar una familia genérica de señales. Una familia de señales se modela principalmente para especificar los tipos de señales que puede recibir un objeto activo; las situaciones anormales se modelan principal-

mente para especificar los tipos de situaciones anormales que puede producir un objeto.

Para modelar situaciones anormales:

- Hay que considerar las situaciones normales que ocurren en cada clase e interfaz, y con cada operación de estos elementos. Después hay que pensar en las cosas que pueden ir mal y hay que modelarlas como señales entre objetos.
- Hay que organizar esas señales en una jerarquía. Deben colocarse arriba las generales y abajo las especializadas, y hay que introducir excepciones intermedias donde sea necesario.
- Hay que especificar las señales de situaciones anormales que puede producir cada operación. Esto se puede hacer explícitamente (mostrando dependencias `send` desde una operación hacia sus señales) o se pueden utilizar diagramas de secuencia que ilustren diferentes escenarios.

Las clases plantilla se discuten en el Capítulo 9.

La Figura 21.7 modela una jerarquía de situaciones anormales que pueden producirse en una biblioteca estándar de clases contenedoras, como la clase plantilla `Conjunto`. Esta jerarquía está encabezada por la señal abstracta `ErrorDeConjunto`. La jerarquía incluye tres tipos de errores especializados: `Duplicado`, `Overflow` y `Underflow`. Como se puede ver, la operación `añadir()` puede producir las señales `Duplicado` y `Overflow`, y la operación `eliminar()` sólo produce la señal `Underflow`. De forma alternativa, se podrían haber incluido estas dependencias indicándolas en la especificación de cada operación. En cualquier caso, al conocer las señales que puede enviar cada operación se pueden crear clientes que utilicen la clase `Conjunto` correctamente.

Nota: Las señales, incluyendo las de situaciones anormales, son eventos síncronos entre objetos. UML también incluye excepciones como las de Ada o C++. Las excepciones son condiciones que hacen que se abandone el flujo de ejecución principal y se pase a un flujo de ejecución secundario en su lugar. Las excepciones no son señales. En vez de ello, son un mecanismo conveniente para especificar un flujo de control alternativo dentro de un hilo de ejecución síncrono.

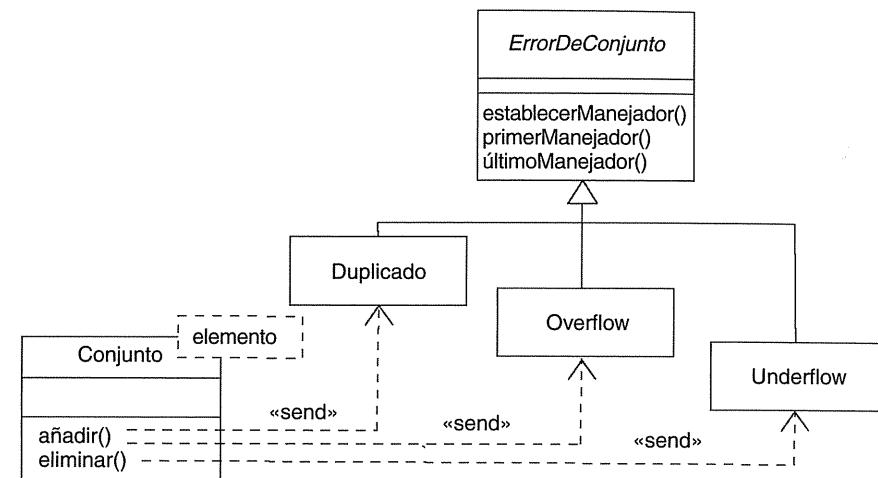


Figura 21.7: Modelado de condiciones de error.

Sugerencias y consejos

Cuando se modela un evento:

- Hay que construir jerarquías de señales para explotar las propiedades comunes de las señales que estén relacionadas.
- Hay que asegurarse de que se dispone de una máquina de estados apropiada para cada elemento que puede recibir el evento.
- Hay que asegurarse de modelar no sólo aquellos elementos que pueden recibir eventos, sino también aquellos que pueden enviarlos.

Cuando se dibuja un evento en UML:

- En general, hay que modelar las jerarquías de eventos explícitamente, pero su uso debe modelarse en la especificación de cada clase u operación que envíe o reciba ese evento.



Capítulo 22

MÁQUINAS DE ESTADOS

En este capítulo

- Estados, transiciones y actividades.
- Modelado de la vida de un objeto.
- Creación de algoritmos bien estructurados.

Las interacciones se discuten en el Capítulo 16; los objetos se discuten en el Capítulo 13.

Las clases se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; los sistemas se discuten en el Capítulo 32; los diagramas de actividades se discuten en el Capítulo 20; los diagramas de estados se discuten en el Capítulo 25.

El comportamiento de una sociedad de objetos que colaboran puede ser modelado mediante una interacción. El comportamiento de un objeto individual puede ser modelado mediante una máquina de estados. Una máquina de estados es un comportamiento que especifica las secuencias de estados por los que pasa un objeto durante su vida, en respuesta a eventos, junto con sus respuestas a esos eventos.

Las máquinas de estados se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar la vida de las instancias de una clase, un caso de uso o un sistema completo. Estas instancias pueden responder a eventos tales como señales, operaciones o el paso del tiempo. Al ocurrir un evento, tendrá lugar un cierto efecto, según el estado actual del objeto. Un *efecto* es la especificación de la ejecución de un comportamiento dentro de una máquina de estados. Los efectos al final conllevan la ejecución de acciones que cambian el estado de un objeto o devuelven valores. El *estado* de un objeto es un período de tiempo durante el cual satisface alguna condición, realiza alguna actividad o espera algún evento.

La dinámica de la ejecución se puede ver de dos formas: destacando el flujo de control entre actividades (con diagramas de actividades), o destacando los estados potenciales de los objetos y las transiciones entre esos estados (con diagramas de estados).

Las máquinas de estados bien estructuradas son como los algoritmos bien estructurados: eficientes, sencillas, adaptables y fáciles de comprender.

Introducción

Considérese la vida del termostato de una casa en un desapacible día otoñal.

A primeras horas de la madrugada, las cosas están tranquilas para el humilde termostato. La temperatura de la casa es estable y, a menos que haya alguna racha fuerte de viento o una tormenta pasajera, la temperatura externa también es estable. Sin embargo, cuando empieza a amanecer, las cosas se ponen más interesantes. El sol comienza a elevarse sobre el horizonte, y sube ligeramente la temperatura ambiente. Los miembros de la familia empiezan a despertarse; alguno puede dejar la cama y girar el marcador del termostato. Ambos eventos son significativos para el sistema de calefacción y refrigeración de la casa. El termostato empieza a comportarse como deben hacerlo todos los buenos termostatos, enviando órdenes al calentador (para elevar la temperatura interior) o al aire acondicionado (para disminuir la temperatura interior).

Una vez que toda la familia se ha ido al trabajo o al colegio, las cosas se tranquilizan, y la temperatura se estabiliza de nuevo. Sin embargo, un programa automático podría intervenir, ordenando al termostato que disminuya la temperatura para ahorrar electricidad y gas. El termostato vuelve a trabajar. Más tarde, el programa se activa de nuevo, y esta vez ordena al termostato que eleve la temperatura para que la familia pueda regresar a un hogar acogedor.

Por la noche, con la casa llena de personas y con el calor de la cocina, el termostato tiene bastante trabajo que hacer para mantener la temperatura, incluso manejando el calentador y el sistema de refrigeración eficientemente.

Por último, cuando la familia se acuesta, las cosas vuelven a un estado tranquilo.

Muchos sistemas con gran cantidad de software se comportan como el termostato. Un marcapasos funciona continuamente, pero se adapta a los cambios en la presión sanguínea o en el ejercicio físico. Un enrutador también funciona de manera continua, encaminando silenciosamente flujos asíncronos de bits, a veces adaptando su comportamiento en respuesta a las órdenes del administrador de la red. Un teléfono móvil trabaja bajo demanda, respondiendo a las entradas del usuario y a los mensajes de la operadora.

El modelado de los aspectos estructurales de un sistema se discute en las Partes 2 y 3.

En UML, los aspectos estáticos de un sistema se modelan utilizando elementos tales como diagramas de clases y diagramas de objetos. Estos diagramas permiten visualizar, especificar, construir y documentar los elementos del sistema, incluyendo clases, interfaces, componentes, nodos, casos de uso y sus instancias, junto a la forma en que estos elementos se relacionan entre sí.

También se pueden modelar los aspectos dinámicos de un sistema utilizando interacciones, como se discute en el Capítulo 16; los eventos se discuten en el Capítulo 21.

En UML, los aspectos dinámicos de un sistema se modelan utilizando máquinas de estados. Mientras que una interacción modela una sociedad de objetos que colaboran para llevar a cabo alguna acción, una máquina de estados modela la vida de un único objeto, bien sea una instancia de una clase, un caso de uso o un sistema completo. Durante su vida, un objeto puede estar expuesto a varios tipos de eventos, como una señal, la invocación de una operación, la creación o destrucción del objeto, el paso del tiempo o el cambio en alguna condición. Como respuesta a estos eventos, el objeto reacciona con alguna acción, que representa una computación, y después cambia su estado a un nuevo valor. Por lo tanto, el comportamiento del objeto se ve afectado por el pasado, al menos tal y como el estado actual refleja el pasado. Un objeto puede recibir un evento, responder con una acción y cambiar su estado. Ese mismo objeto puede recibir otro evento, y su respuesta al evento puede ser diferente, dependiendo de su estado actual, como resultado de su respuesta al primer evento.

Los diagramas de actividades se discuten en el Capítulo 20; los diagramas de estados se discuten en el Capítulo 25.

Las máquinas de estados se utilizan para modelar el comportamiento de cualquier elemento de modelado, aunque, la mayoría de las veces, éste será una clase, un caso de uso o un sistema completo. Las máquinas de estados pueden visualizarse utilizando diagramas de estados. Podemos centrarnos en el comportamiento dirigido por eventos de un objeto, lo que es especialmente útil cuando se modelan sistemas reactivos.

UML proporciona una representación gráfica para los estados, las transiciones, los eventos y las acciones, como se muestra en la Figura 22.1. Esta notación permite visualizar el comportamiento de un objeto de forma que permite destacar los elementos más importantes en su vida.

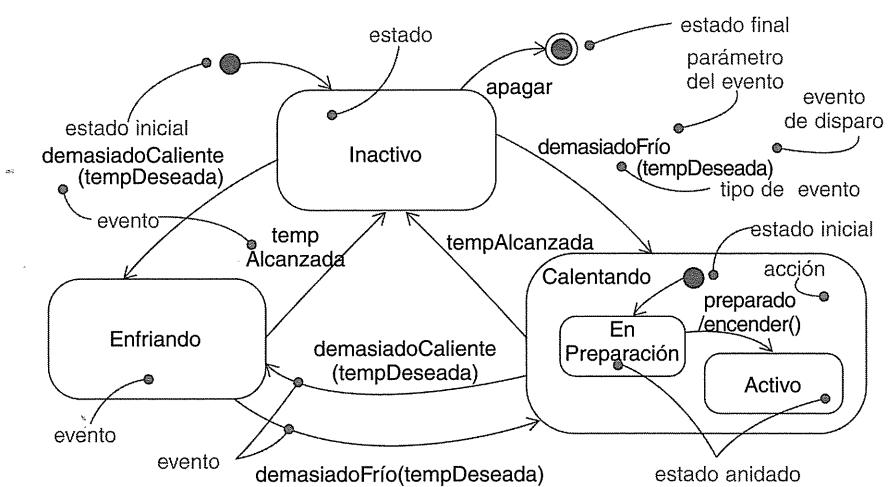


Figura 21.1: Máquinas de estados.

Términos y conceptos

Una *máquina de estados* es un comportamiento que especifica las secuencias de estados por los que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a estos eventos. Un *estado* es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un *evento* es la especificación de un acontecimiento significativo situado en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede disparar una transición de estados. Una *transición* es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones específicas. Una *actividad* es una ejecución no atómica en curso, dentro de una máquina de estados. Una *acción* es una computación atómica ejecutable que produce un cambio en el estado del modelo o devuelve un valor. Gráficamente, un estado se representa como un rectángulo con las esquinas redondeadas. Una transición se representa como una línea continua dirigida o camino desde el estado origen al nuevo estado.

Contexto

Los objetos se discuten en el Capítulo 13; los mensajes se discuten en el Capítulo 16.

Cada objeto tiene una vida. Cuando se crea, un objeto nace; cuando se destruye, un objeto deja de existir. Entretanto, un objeto puede actuar sobre otros objetos (enviando mensajes), y los demás pueden actuar sobre él (siendo el destinatario de un mensaje). En muchos casos, los mensajes serán simples llamadas de operaciones síncronas. Por ejemplo, una instancia de la clase `Cliente` podría invocar la operación `obtenerSaldoDeCuenta` sobre una instancia de la clase `CuentaBancaria`. Los objetos de esta naturaleza no necesitan una máquina de estados para especificar su comportamiento, porque su comportamiento actual no depende de su pasado.

Las señales se discuten en el Capítulo 21.

En otros tipos de sistemas, aparecerán objetos que deben responder a señales, las cuales son mensajes asíncronos entre instancias. Por ejemplo, un teléfono móvil debe responder a llamadas que pueden ocurrir en cualquier momento (desde otros teléfonos), a eventos producidos al pulsar las teclas (por el usuario que inicia una llamada) y a eventos de la red (al pasar de una centralita a otra). Análogamente, habrá objetos cuyo comportamiento actual dependa de su comportamiento pasado. Por ejemplo, el comportamiento de un sistema de guía de un misil aire-aire dependerá de su estado actual, por ejemplo

Los objetos activos se discuten en el Capítulo 23; el modelado de sistemas reactivos se discute en el Capítulo 25; los casos de uso y los actores se discuten en el Capítulo 17; las interacciones se discuten en el Capítulo 16; las interfaces se discuten en el Capítulo 11.

NoVolando (no es una buena idea lanzar un misil mientras está enganchado a un avión que aún está en la pista) o Buscando (no se debería activar el misil hasta tener una buena idea de dónde va a impactar).

La mejor forma de especificar el comportamiento de los objetos que deben responder a estímulos asíncronos o cuyo comportamiento actual depende de su pasado es utilizar una máquina de estados. Esto engloba a las instancias de clases que pueden recibir señales, incluyendo muchos objetos activos. De hecho, un objeto que recibe una señal pero no tiene una transición para esa señal en el estado actual y no la difiere, simplemente ignorará la señal. En otras palabras, la ausencia de una transición para la señal no es un error; sólo significa que la señal no interesa en ese punto. Las máquinas de estados también se utilizan para modelar el comportamiento de sistemas completos, especialmente sistemas reactivos, que deben responder a las señales de actores externos al sistema.

Nota: La mayoría de las veces, el comportamiento de un caso de uso se modela mediante interacciones, pero también se pueden emplear las máquinas de estados para el mismo propósito. Igualmente, las máquinas de estados se pueden emplear para modelar el comportamiento de una interfaz. Aunque una interfaz no puede tener instancias directas, una clase que realice a dicha interfaz sí puede. Esa clase debe conformar con el comportamiento especificado por la máquina de estados de la interfaz.

El estado de un objeto se puede visualizar en una interacción, como se discute en el Capítulo 13; las cuatro últimas partes de un estado se discuten en secciones posteriores de este mismo capítulo.

Estados

Un estado es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un objeto permanece en un estado durante una cantidad de tiempo finita. Por ejemplo, un `Calentador` en una casa puede estar en cualquiera de los cuatro estados siguientes: `Inactivo` (esperando una orden para calentar la casa), `EnPreparacion` (el gas está abierto, pero está esperando a que se eleve la temperatura), `Activo` (el gas está abierto y el ventilador está encendido) y `Apagando` (el gas se ha cerrado, pero el ventilador continúa activado, arrojando el calor residual del sistema).

Cuando la máquina de estados de un objeto se encuentra en un estado dado, se dice que el objeto está en ese estado. Por ejemplo, una instancia de `Calentador` podría estar `Inactivo` o quizás `Apagando`.

Un estado tiene varias partes:

1. Nombre Una cadena de texto que distingue al estado de otros estados; un estado puede ser anónimo, es decir, no tiene nombre.
2. Efectos de entrada/salida Acciones ejecutadas al entrar y salir del estado, respectivamente.
3. Transiciones internas Transiciones que se manejan sin causar un cambio en el estado.
4. Subestados Estructura anidada de un estado, que engloba subestados disjuntos (activos secuencialmente) o concurrentes (activos concurrentemente).
5. Eventos diferidos Una lista de eventos que no se manejan en este estado sino que se posponen y se añaden a una cola para ser manejados por el objeto en otro estado.

Nota: El nombre de un estado puede ser texto formado por cualquier número de letras, dígitos y ciertos signos de puntuación (excepto signos como los dos puntos) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de estados son nombres cortos o expresiones nominales extraídos del vocabulario del sistema que se está modelando. Normalmente, en el nombre de un estado se pone en mayúsculas la primera letra de cada palabra, como en Inactivo o Apagando.

Como se muestra en la Figura 22.2, un estado se representa con un rectángulo con las esquinas redondeadas.

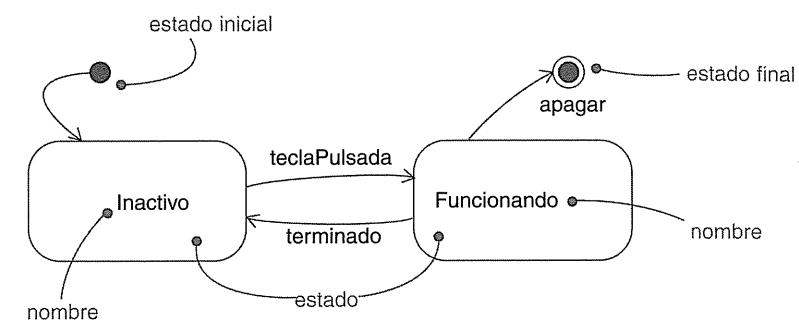


Figura 22.2: Estados.

Estados inicial y final. Como muestra la figura, en la máquina de estados de un objeto se pueden definir dos estados especiales. En primer lugar, el estado inicial, que indica el punto de comienzo por defecto para la máquina de estados o del subestado. Un estado inicial se representa por un círculo negro. En segundo lugar, el estado final, que indica que la ejecución de la máquina de estados o del estado que lo contiene ha finalizado. Un estado final se representa como un círculo negro dentro de un círculo blanco (una diana).

Nota: Los estados inicial y final son en realidad pseudoestados. No pueden tener las partes habituales de un estado normal, excepto un nombre. Una transición desde un estado inicial hasta un estado final puede ir acompañada de cualquiera de las características, incluyendo una condición de guarda y una acción (pero no un evento de disparo).

Transiciones

Una *transición* es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones especificadas. Cuando se produce ese cambio de estado, se dice que la transición se ha disparado. Hasta que se dispara la transición, se dice que el objeto está en el estado origen; después de dispararse, se dice que está en el estado destino. Por ejemplo, un Calentador podría pasar del estado Inactivo al estado EnPreparación cuando ocurriera un evento como demasiadoFrío (con el parámetro tempDeseada).

Los eventos se discuten en el Capítulo 21.

Una transición tiene cinco partes:

1. Estado origen
 2. Evento de disparo
 3. Condición de guarda
 4. Efecto
 5. Estado destino
- El estado afectado por la transición; si un objeto está en el estado origen, una transición de salida puede dispararse cuando el objeto reciba el evento de disparo de la transición, y si la condición de guarda, si la hay, se satisface.
- El evento cuya recepción por el objeto que está en el estado origen provoca el disparo de la transición si se satisface su condición de guarda.
- Una expresión booleana que se evalúa cuando la transición se activa por la recepción del evento de disparo; si la expresión toma el valor *verdadero*, la transición se puede disparar; si la expresión toma el valor *falso*, la transición no se dispara, y si no hay otra transición que pueda ser disparada por el mismo evento, éste se pierde.
- Un comportamiento ejecutable, como una acción, que puede actuar directamente sobre el objeto asociado a la máquina de estados, e indirectamente sobre otros objetos visibles al objeto.
- El estado activo tras completarse la transición.

Como se muestra en la Figura 22.3, una transición se representa con una línea continua dirigida desde el estado origen hacia el destino. Una autotransición es una transición cuyos estados origen y destino son el mismo.

Nota: Una transición puede tener múltiples orígenes, en cuyo caso representa una unión (*join*) de muchos estados concurrentes, así como múltiples destinos, en cuyo caso representa una división (*fork*) a múltiples estados concurrentes. Véase la descripción de los subestados ortogonales más adelante.

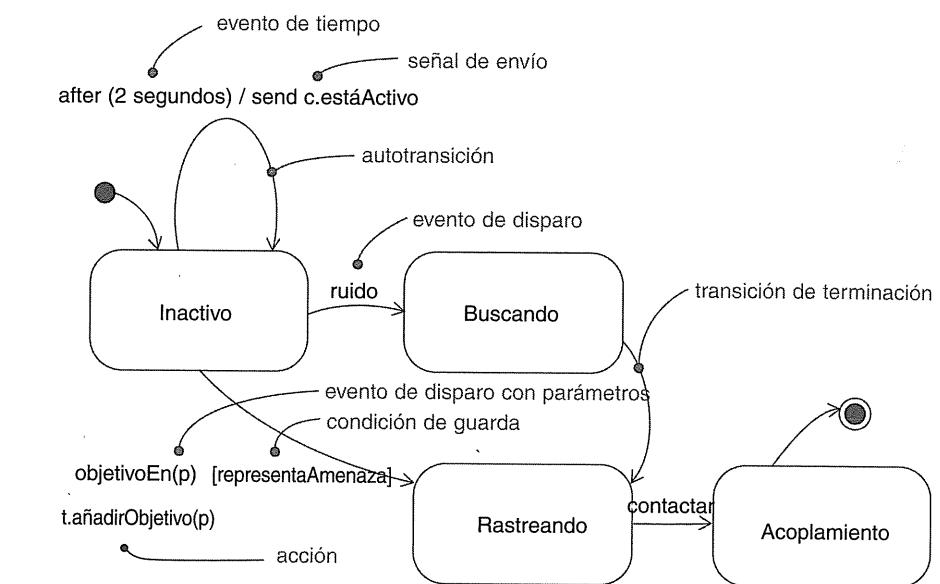


Figura 22.3: Transiciones.

Los eventos se discuten en el Capítulo 21.

La especificación de una familia de señales se discute en el Capítulo 21; las condiciones de guarda múltiples no solapadas forman una bifurcación, como se discute en el Capítulo 20.

Evento de disparo. Un evento es la especificación de un acontecimiento significativo situado en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede disparar una transición de estado. Como se representa en la figura anterior, los eventos pueden incluir señales, llamadas, el paso del tiempo, o un cambio en el estado. Una señal o una llamada pueden tener parámetros cuyos valores estén disponibles para la transición, incluyendo expresiones para la condición de guarda y la acción.

También es posible tener una transición de terminación, representada por una transición sin evento de disparo. Una transición de terminación se dispara implícitamente cuando su estado origen ha completado su comportamiento, si lo tiene.

Nota: Un evento de disparo puede ser polimórfico. Por ejemplo, si se ha especificado una familia de señales, entonces una transición cuyo evento de disparo sea S puede ser activada por S, así como por cualquier evento hijo de S.

Condición de guarda. Como se muestra en la figura anterior, una condición de guarda se representa como una expresión booleana entre corchetes y se coloca tras el evento de disparo. Una condición de guarda sólo se evalúa después

de ocurrir el evento de disparo de la transición. Por lo tanto, es posible tener muchas transiciones desde el mismo estado origen y con el mismo evento de disparo, siempre y cuando sus condiciones no se solapen.

Una condición de guarda se evalúa sólo una vez por cada transición, cuando ocurre el evento, pero puede ser evaluada de nuevo si la transición se vuelve a disparar. Dentro de la expresión booleana se pueden incluir condiciones sobre el estado de un objeto (por ejemplo, la expresión `unCalentador in Inactivo`, que toma el valor verdadero si el objeto `un Calentador` está actualmente en el estado `Inactivo`). Si el valor de la condición es falso cuando se evalúa, el evento no ocurre más tarde cuando la condición sea cierta. Para modelar este tipo de comportamiento debe utilizarse un evento de cambio.

Los eventos de cambio se discuten en el Capítulo 21.

Nota: Aunque una condición de guarda sólo se evalúa una vez cada vez que se activa su transición, un evento de cambio se evalúa potencialmente de forma continua.

Efecto. Un efecto es un comportamiento que se evalúa cuando se dispara una transición. Los efectos pueden incluir una computación en línea, llamadas a operaciones (sobre el objeto que contiene la máquina de estados, así como sobre otros objetos visibles), la creación o destrucción de otro objeto, o el envío de una señal a un objeto. Para indicar el envío de una señal, el nombre de la señal se puede preceder de la palabra clave `send` como una indicación visual.

Las actividades se discuten en una sección posterior de este capítulo; las dependencias se discuten en los Capítulos 5 y 10.

Las transiciones sólo ocurren cuando la máquina de estados está inactiva, es decir, cuando no está ejecutando el efecto de una transición previa. La ejecución del efecto de una transición y cualquier efecto de entrada y salida se ejecuta hasta su terminación antes de que se permita que nuevos eventos causen transiciones adicionales. Esto contrasta con una actividad (descrita más adelante), que puede ser interrumpida por eventos.

Nota: Se puede representar explícitamente el objeto al que se envía una señal con una dependencia estereotipada como `send`, cuyo origen sea el estado y cuyo destino sea el objeto.

Aspectos avanzados de los estados y las transiciones

Con UML, se puede modelar una amplia variedad de comportamientos utilizando únicamente las características básicas de los estados y las transiciones. Con estas características, se consiguen máquinas de estados planas, lo que significa que los modelos de comportamiento sólo consistirán en arcos (transiciones) y nodos (estados).

Sin embargo, las máquinas de estados de UML tienen varias características avanzadas que ayudan a crear modelos de comportamiento complejos. A menudo, estas características reducen el número de estados y transiciones necesarios, y permiten simplificar construcciones frecuentes y algo complejas, que de otra forma aparecerían con las máquinas de estados planas. Algunas de estas características avanzadas incluyen los efectos de entrada y salida, transiciones internas, actividades y eventos diferidos. Estas características se representan como cadenas de texto dentro de un compartimento en el símbolo del estado, como se puede ver en la Figura 22.4.

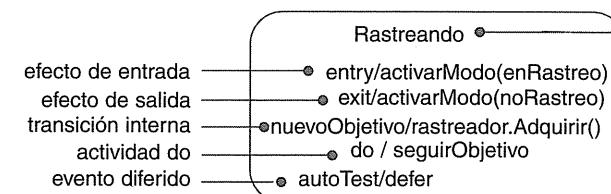


Figura 22.4: Estados y transiciones avanzados.

Efectos de entrada y salida. En varias situaciones de modelado, a veces se desea realizar alguna acción de configuración cada vez que se entra en un estado, sin importar qué transición ha conducido a ese estado. Igualmente, al abandonar un estado, a veces se desea realizar la misma acción de limpieza, sin importar qué transición provoca la salida. Por ejemplo, en un sistema de guía de misiles, quizás se quiera indicar explícitamente que el sistema está en `Rastreo` cada vez que está en el estado `Rastreando`, y en `noRastreo` cada vez que sale del estado. Con las máquinas de estados planas, se puede lograr este efecto poniendo estas acciones sobre cada transición de entrada y salida, según sea apropiado. Sin embargo, esto tiende a producir errores; cada vez que se añade una nueva transición hay que recordar añadir estas acciones. Además, modificar esta acción implica que hay que tocar cada transición asociada.

Como se muestra en la Figura 22.4, UML proporciona una forma simplificada para expresar esta construcción. En el símbolo del estado se puede incluir un efecto de entrada (etiquetado con la palabra `entry`) y un efecto de salida (etiquetado con la palabra `exit`), junto a una acción apropiada. Cada vez que se entre al estado, se ejecuta la acción de entrada; cada vez que se abandone el estado, se ejecuta la acción de salida.

Los efectos de entrada y salida no pueden tener argumentos ni condiciones de guarda. Sin embargo, el efecto de entrada del nivel superior de una máquina de estados de una clase puede tener parámetros, para los argumentos que recibe la máquina cuando se crea el objeto.

Transiciones internas. Una vez dentro de un estado, es posible identificar eventos que deban ser manejados sin abandonar el estado. Surgen así las denominadas transiciones internas, y se diferencian ligeramente de las autotransiciones. En una autotransición, como la que se muestra en la Figura 22.3, un evento dispara la transición, se abandona el estado, se ejecuta una acción (si la hay), y se vuelve a entrar en el mismo estado. Como la transición sale y entra en el estado, una autotransición ejecuta la acción de salida, después ejecuta la acción de la autotransición y, por último, ejecuta la acción de entrada del estado. Sin embargo, puede suceder que se desee manejar el evento pero sin activar las acciones de entrada y salida del estado. UML proporciona una abreviatura para esta construcción utilizando una transición interna. Una transición interna es una transición que responde a un evento ejecutando un efecto, pero no cambia de estado. En la Figura 22.4, el evento `nuevoObjetivo` etiqueta una transición interna; si este evento ocurre mientras el objeto está en el estado `Rastreando`, se ejecuta la acción `rastreador . adquirir`, pero el estado sigue siendo el mismo, y no se ejecutan las acciones de entrada ni de salida. Una *transición interna* se identifica incluyendo su texto (con el nombre del evento, la condición de guarda opcional, y el efecto) dentro del símbolo del estado, en vez de hacerlo en una línea de transición. Las palabras `entry`, `exit` y `do` son palabras reservadas que no pueden utilizarse como nombres de eventos. Cuando se está en el estado y ocurre una transición interna, el efecto correspondiente se ejecuta sin abandonar y volver a entrar en el estado. Por lo tanto, el evento se maneja sin invocar las acciones de salida ni de entrada del estado.

- **Nota:** Las transiciones internas pueden tener eventos con parámetros y condiciones de guarda.

Los eventos se discuten en el Capítulo 21.

Actividades-Do. Cuando un objeto está en un estado, normalmente se encuentra ocioso, esperando a que ocurra un evento. A veces, no obstante, puede que se quiera modelar una actividad en curso. Mientras está en el estado, el objeto realiza alguna tarea que continuará hasta que sea interrumpido por un evento. Por ejemplo, si un objeto está en el estado `Rastreando`, podría seguir `Objetivo` mientras estuviera en ese estado. Como se muestra en la Figura 22.4, en UML se utiliza la transición especial `do` para especificar el trabajo que se realizará en ese estado después de ejecutar la acción de entrada. También se puede especificar un comportamiento, como una secuencia de operaciones; por ejemplo, `do / op1(a); op2(b); op3(c)`. Si la aparición del evento causa una transición que fuerza una salida del estado, cualquier actividad-`do` que esté actualmente en marcha se ve interrumpida inmediatamente.

Nota: Una actividad-`do` es equivalente a un efecto de entrada que inicia la actividad cuando se entra en el estado y a un efecto de salida que detiene la actividad cuando se sale del estado.

Eventos diferidos. Considérese un estado como `Rastreando`. Supóngase que sólo hay una transición que lleva fuera del estado, como se muestra en la Figura 22.3, disparada por el evento `contactar`. Mientras se esté en el estado `Rastreando`, se perderá cualquier evento distinto a `contactar` y a los manejados por sus subestados. Esto significa que el evento puede ocurrir, pero será ignorado y no se producirá ninguna acción por la presencia de ese evento.

En cualquier situación de modelado, es preciso reconocer algunos eventos e ignorar otros. Los que se quiere reconocer se incluyen como eventos disparadores de las transiciones; los que se quiere ignorar simplemente se dejan fuera. Sin embargo, en algunas situaciones de modelado se deseará reconocer algunos eventos pero posponer una respuesta a ellos para más tarde. Por ejemplo, mientras se está en el estado `Rastreando`, se retrasará la respuesta a señales como `autoTest`, quizás enviada por algún agente de mantenimiento del sistema.

En UML, este comportamiento se puede especificar mediante eventos diferidos. Un evento diferido es una lista de eventos cuya aparición en el estado se pospone hasta que se activa otro estado; si los eventos no han sido diferidos en ese estado, se maneja el evento y se pueden disparar transiciones como si realmente acabaran de producirse. Si la máquina de estados pasa por una secuencia de estados en los cuales el evento está diferido, éste se mantiene hasta que se encuentre finalmente un estado donde el evento no esté diferido. Durante este intervalo, pueden ocurrir otros eventos no diferidos. Como se muestra en la Figura 21.4,

un evento diferido se puede especificar listando el evento con la acción especial `defer`. En este ejemplo, los eventos `autoTest` pueden ocurrir mientras se está en el estado `Rastreando`, pero se retienen hasta que el objeto pasa al estado `Acoplamiento`, momento en el que aparecen como si acabaran de ocurrir.

Nota: La implementación de eventos diferidos requiere la presencia de una cola interna de eventos. Si aparece un evento pero está listado como diferido, se añade a la cola. Los eventos se extraen de la cola tan pronto como el objeto entra en un estado que no difiere estos eventos.

Submáquinas. Una máquina de estados puede ser referenciada desde dentro de otra máquina. Una máquina de estados así referenciada se denomina *submáquina*. Éstas son útiles para construir grandes modelos de estados de manera estructurada. Para más detalle, véase el *Manual de Referencia de UML*.

Subestados

Estas características avanzadas de los estados y las transiciones solucionan varios problemas de modelado frecuentes en las máquinas de estados. No obstante, aún hay otra característica de las máquinas de estados de UML (los subestados) que ayuda aún más a simplificar el modelado de comportamientos complejos. Un subestado es un estado anidado dentro de otro. Por ejemplo, un Calentador podría estar en el estado `Calentando`, pero también mientras se está en el estado `Calentando`, podría haber un estado anidado llamado `EnPreparación`. En este caso, es correcto decir que el objeto está `Calentando` y `EnPreparación`.

Los estados compuestos tienen una estructura de anidamiento similar a la de la composición, como se discute en los Capítulos 5 y 10.

Un estado simple es el que no tiene una subestructura. Un estado con subestados (o sea, estados anidados) se llama estado compuesto. Un estado compuesto puede contener subestados concurrentes (ortogonales) o secuenciales (no ortogonales). En UML, un estado compuesto se representa igual que un estado simple, pero con un compartimento gráfico opcional que muestra una máquina de estados anidada. Los subestados se pueden anidar a cualquier nivel.

Subestados no ortogonales. Considérese el problema de modelar el comportamiento de un cajero automático. Hay tres estados básicos en los que puede encontrarse el sistema: `Inactivo` (esperando la interacción de un cliente), `Activo` (manejando una transacción de un cliente) y `Mantenimiento` (quizá recargando el depósito del dinero). Mientras está `Activo`, el comportamiento del cajero sigue un camino muy simple: validar el cliente, seleccionar

una transacción, procesar la transacción e imprimir el recibo. Después de imprimir, el cajero vuelve al estado `Inactivo`. Estas etapas de comportamiento se podrían representar con los estados `Validación`, `Selección`, `Procesamiento` e `Impresión`. Incluso sería deseable permitir que el cliente seleccionara y procesara varias transacciones después de hacer la `Validación` de la cuenta y antes de la `Impresión` del recibo final.

El problema aquí es que, en cualquier etapa de este comportamiento, el cliente podría decidir cancelar la transacción, y el cajero volvería a su estado `Inactivo`. Con las máquinas de estados planas se puede lograr este efecto, pero es bastante complicado. Como el cliente puede cancelar la transacción en cualquier momento, hay que incluir una transición adecuada desde cualquier estado de la secuencia `Activo`. Esto es complicado porque es fácil olvidar la inclusión de estas transiciones en todos los sitios adecuados, y tener muchos de esos eventos de interrupción significaría que muchas transiciones llegarían al mismo estado destino desde distintos orígenes, pero con el mismo evento de disparo, la misma condición de guarda y la misma acción.

El modelado del vocabulario de un sistema se discute en el Capítulo 4; las precondiciones y las postcondiciones se discuten en el Capítulo 4.

Si se utilizan estados anidados, hay una forma más sencilla de modelar este problema, como se muestra en la Figura 22.5. Aquí, el estado `Activo` tiene una subestructura, que contiene los estados `Validación`, `Selección`, `Procesamiento` e `Impresión`. El estado del cajero cambia de `Inactivo` a `Activo` cuando se introduce una tarjeta de crédito en la máquina. Al entrar en el estado `Activo`, se ejecuta la acción de entrada `leerTarjeta`. Entonces el control pasa al estado `Validación`, que es el estado inicial de la subestructura; a continuación pasa al estado `Selección` y después pasa al estado `Procesamiento`. Cuando se sale del estado `Procesamiento`, el control puede volver a `Selección` (si el cliente selecciona otra transacción) o puede pasar a `Impresión`. Después de `Impresión`, hay una transición sin evento de disparo de vuelta al estado `Inactivo`. Nótese que el estado `Activo` tiene una acción de salida, que expulsa la tarjeta de crédito del cliente.

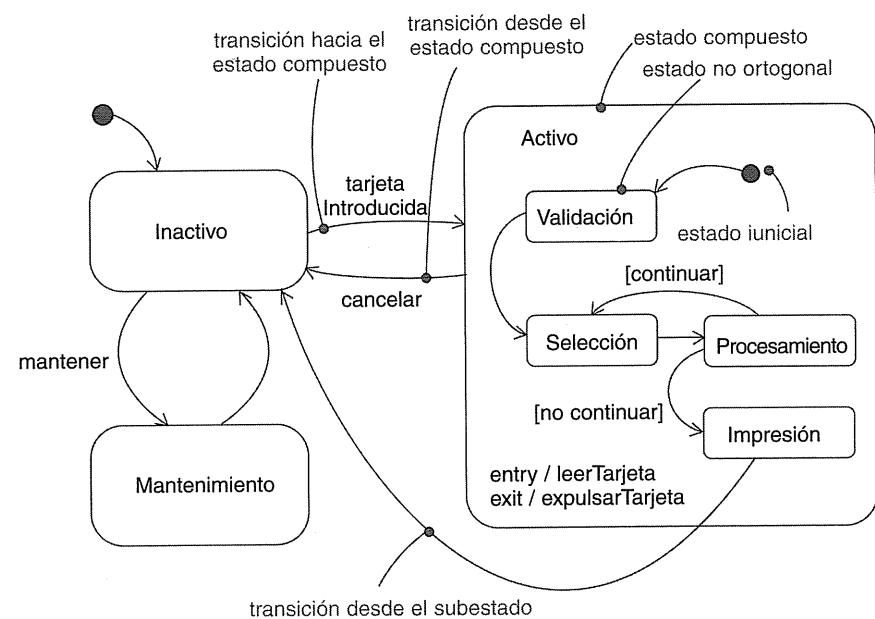


Figura 22.5: Subestados secuenciales.

También debemos fijarnos en la transición desde el estado **Activo** hasta el estado **Inactivo**, disparada por el evento **cancelar**. En cualquier subestado de **Activo**, el cliente podría cancelar la transacción, lo que haría pasar al cajero al estado **Inactivo** (pero sólo después de haber expulsado la tarjeta del cliente, que es la acción de salida que se ejecuta al salir del estado **Activo**, independientemente de lo que cause la transición de salida). Sin los subestados, sería necesaria una transición disparada por **cancelar** en cada estado de la subestructura.

Los subestados como **Validación** y **Procesamiento** se llaman subestados no ortogonales o disjuntos. Dado un conjunto de estados no ortogonales en el contexto de un estado compuesto, se dice que el objeto está en el estado compuesto, y sólo en uno de sus subestados (o en el estado final) en un momento dado. Por lo tanto, los subestados no ortogonales partitionan el espacio de estados del estado compuesto en estados disjuntos.

Dado un estado compuesto, una transición desde un origen externo puede apuntar al estado compuesto o a un subestado. Si su destino es el estado compuesto, la máquina de estados anidada debe incluir un estado inicial, al cual pasa el control después de haber entrado en el estado compuesto y haber ejecutado su acción de entrada (si la hay). Si el destino es el estado anidado, el control pasa a éste, después

de ejecutar la acción de entrada del estado compuesto (si la hay) y a continuación la acción de entrada del subestado (si la hay).

Una transición que salga de un estado compuesto puede tener como origen el propio estado compuesto o un subestado. En cualquier caso, el control sale primero del estado anidado (y se ejecuta su acción de salida, si la hay), después deja el estado compuesto (y se ejecuta su acción de salida, si la hay). Una transición cuyo origen sea el estado compuesto, básicamente intercepta (interrumpe) la actividad de la máquina de estados anidada. La transición de terminación de un estado compuesto se ejecuta cuando el control llega al subestado final dentro del estado compuesto.

Nota: Una máquina de estados no ortogonal anidada puede tener como máximo un subestado inicial y un subestado final.

Estados de historia. Una máquina de estados describe los aspectos dinámicos de un objeto cuyo comportamiento actual depende de su pasado. En efecto, una máquina de estados especifica la secuencia válida de estados por la que puede pasar un objeto a lo largo de su vida.

A menos que se especifique lo contrario, cuando una transición entra en un estado compuesto, la acción de la máquina de estados anidada comienza de nuevo en su estado inicial (a menos, por supuesto, que la transición se dirija directamente a un subestado). No obstante, a veces se desea modelar un objeto de forma que recuerde el último subestado que estaba activo antes de abandonar el estado compuesto. Por ejemplo, en el modelado del estado de un agente que se encarga de hacer copias de seguridad de forma no interactiva a través de una red, es deseable que el agente recuerde dónde estaba en el proceso si, por ejemplo, alguna vez es interrumpido por una consulta del operador.

Esto se puede modelar con las máquinas de estados planas, pero es complicado. En cada subestado secuencial habría que hacer que la acción de salida guardase un valor en alguna variable local al estado compuesto. Entonces, el estado inicial de este estado compuesto necesitaría una transición hacia cada subestado con una condición de guarda, consultando la variable. De esta forma, el abandono de un estado compuesto haría que se recordase el último subestado; y entrar en el estado compuesto conduciría al subestado correcto. Esto es complicado, porque obliga a recordar la necesidad de modificar cada subestado y configurar una acción de salida apropiada. Esto conlleva una gran cantidad de transiciones que parten del mismo estado inicial hacia diferentes subestados destino con condiciones de guarda muy similares (pero diferentes).

En UML, una forma más sencilla de modelar esta construcción es utilizar los estados de historia. Un estado de historia permite que un estado compuesto que contiene subestados secuenciales recuerde el último subestado activo antes de la transición que provocó la salida del estado compuesto. Como se muestra en la Figura 22.6, un estado de historia superficial se representa como un pequeño círculo que contiene el símbolo H.

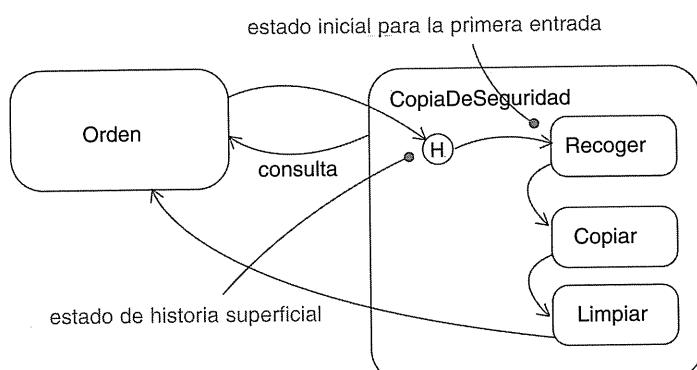


Figura 22.6: Estado de historia.

Cuando se desea que una transición active el último subestado, se representa una transición desde un origen externo al estado compuesto dirigida directamente hacia el estado de historia. La primera vez que se entre a un estado compuesto no hay historia. Éste es el significado de la única transición que va del estado de historia hacia un estado secuencial, en este caso Recoger. El estado destino de esta transición especifica el estado inicial de la máquina de estados anidada para la primera vez que se entre en ella. A continuación, supóngase que mientras se está en el estado CopiaDeSeguridad y en el estado Copiar, se produce el evento consulta. El control abandona Copiar y CopiaDeSeguridad (ejecutando sus acciones de salida según sea necesario) y vuelve al estado Orden. Cuando se completa la acción de Orden, se dispara la transición sin evento de disparo, regresando al estado de historia del estado compuesto CopiaDeSeguridad. Esta vez, como ya hay una historia en la máquina de estados anidada, el control pasa al estado Copiar (así no se pasa por el estado Recoger) porque el último subestado activo previo a la transición desde CopiaDeSeguridad era Copiar.

Nota: El símbolo H designa una historia superficial, que recuerda sólo la historia de la máquina de estados anidada inmediata. También se puede especificar una historia profunda, representada con un pequeño círculo con el símbolo H*. La historia profunda recuerda el estado anidado más interno a cualquier profundidad. Si sólo hay un nivel de anidamiento, los estados de historia superficial y profunda son semánticamente equivalentes. Si hay más de un nivel de anidamiento, la historia superficial recuerda sólo el estado anidado más externo; la historia profunda recuerda el estado anidado más interno a cualquier profundidad.

En cualquier caso, si una máquina de estados anidada alcanza un estado final, pierde su historia almacenada y se comporta como si aún no se hubiera entrado en ella por primera vez.

Subestados ortogonales. Los subestados no ortogonales son el tipo de máquinas de estados anidadas que aparecen más frecuentemente cuando se modela. Sin embargo, en ciertas situaciones de modelado, es preciso especificar regiones ortogonales. Estas regiones permiten especificar dos o más máquinas de estados que se ejecutan en paralelo en el contexto del objeto que las contiene.

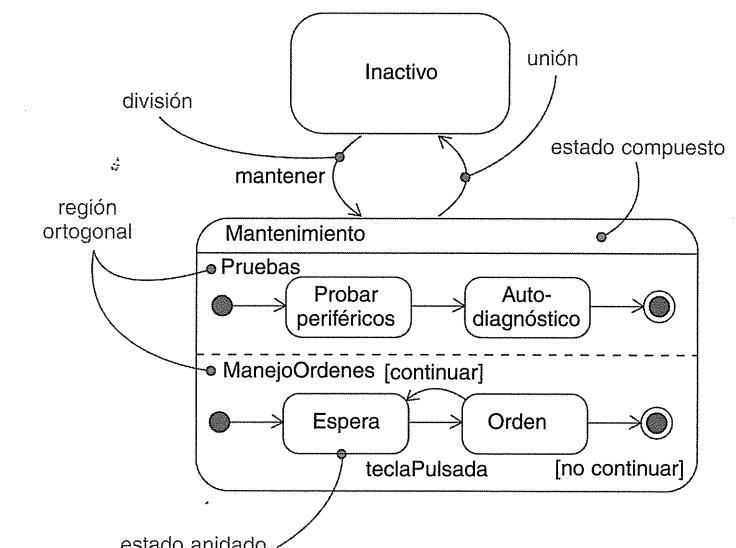


Figura 22.7: Subestados concurrentes.

Por ejemplo, la Figura 22.7 muestra una expansión del estado Mantenimiento de la Figura 22.5. Mantenimiento se ha descompuesto en dos regiones ortogonales, Pruebas y ManejoOrdenes, que se han representado anidadas en el estado Mantenimiento pero separadas entre sí por una línea discontinua. Cada una de estas regiones ortogonales se ha descompuesto a su vez en subestados. Cuando el control pasa del estado Inactivo al estado Mantenimiento, se divide en dos flujos concurrentes (el objeto que los contiene estará tanto en la región Pruebas como en la región ManejoOrdenes). Además, mientras esté en la región ManejoOrdenes, el objeto que lo contiene estará en el estado Espera o en el estado Orden.

Nota: Esto es lo que distingue a los subestados no ortogonales de los ortogonales. Dados dos o más subestados no ortogonales al mismo nivel, un objeto estará en uno y sólo uno de esos subestados. Dadas dos o más regiones ortogonales al mismo nivel, un objeto estará en un estado de cada una de las regiones ortogonales.

La ejecución de estas dos regiones ortogonales continúa en paralelo. Cada máquina de estados anidada alcanza su estado final antes o después. Si una región orthogonal alcanza su estado final antes que la otra, el control en esa región espera en su estado final. Cuando ambas máquinas de estados anidadas alcanzan su estado final, el control de las dos regiones ortogonales se vuelve a unir en un único flujo.

Cada vez que hay una transición hacia un estado compuesto que se descompone en regiones ortogonales, el control se divide en tantos flujos como regiones ortogonales hay. Análogamente, cuando hay una transición desde un estado compuesto dividido en regiones ortogonales, el control se une en un único flujo. Esto es cierto en todos los casos. Si todas las regiones ortogonales alcanzan su estado final, o si hay una transición explícita hacia fuera del estado compuesto, el control se une en un único flujo.

Nota: Cada región orthogonal puede tener un estado inicial, un estado final y un estado de historia.

División y unión. Normalmente, la entrada a un estado compuesto con regiones ortogonales va al estado inicial de cada región orthogonal. También es posible pasar de un estado externo directamente a uno o más estados ortogonales. Esto se denomina una división (*fork*), porque el control pasa de un estado simple a varios estados ortogonales. La división se representa como una línea negra gruesa a la que llega una flecha y de la que salen varias flechas, cada una de ellas a uno de los estados ortogonales. Debe haber como máximo un estado

destino en cada región orthogonal. Si una o más regiones ortogonales no tienen estado destino, se elige de manera implícita el estado inicial de cada región. Una transición a un estado simple orthogonal, dentro de un estado compuesto, también es una división implícita: los estados iniciales de cada una de las otras regiones ortogonales forman parte de la división de manera implícita.

Igualmente, una transición desde cualquier estado de un subestado compuesto con regiones ortogonales fuerza una salida de todas las regiones ortogonales. Una transición así a menudo representa una condición de error que fuerza la terminación de las computaciones paralelas.

Una unión (*join*) es una transición con dos o más flechas entrantes y una flecha saliente. Cada flecha entrante debe venir desde un estado en regiones ortogonales diferentes del mismo estado compuesto. La unión puede tener un evento disparador. La transición de unión sólo tiene efecto si todos los estados de origen están activos; el estado de las otras regiones ortogonales en el estado compuesto no importa. Si ocurre el evento, el control sale de todas las regiones ortogonales en el estado compuesto, no sólo de aquellas que tienen flechas salientes.

La Figura 22.8 muestra una variación del ejemplo anterior con transiciones de división y de unión explícitas. La transición mantener hacia el estado compuesto Mantenimiento es una división implícita hacia los estados iniciales por defecto de las regiones ortogonales. Sin embargo, en este ejemplo, también hay una división explícita desde Inactivo hacia los dos estados anidados Autodiagnóstico y el estado final de la región ManejoOrdenes. (Un estado final es un estado real, y puede ser el destino de una transición). Si ocurre un evento de error mientras está activo el estado Autodiagnóstico, se dispara la transición de unión implícita Reparación: se abandonan tanto el estado Autodiagnóstico como el estado que esté activo en la región ManejoOrdenes. También hay una transición de unión explícita hacia el estado Desconectado. Esta transición se dispara sólo si ocurre el evento desconectar mientras están activos el estado ProbandoDispositivos y el estado final de la región ManejoOrdenes. Si ambos estados no están activos, el evento no tiene efecto.

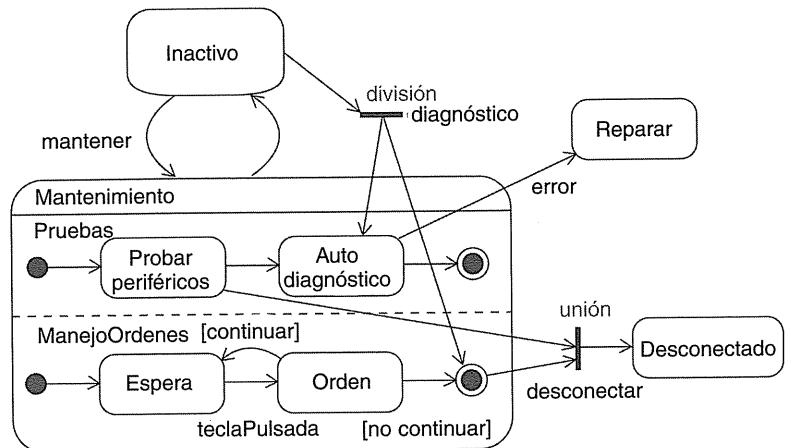


Figura 22.8: Transiciones de división y de unión.

Los objetos activos se discuten en el Capítulo 23.

Objetos activos. Otra forma de modelar la concurrencia es utilizar objetos activos. Así, en vez de particionar la máquina de estados de un objeto en dos (o más) regiones concurrentes, se pueden definir dos objetos activos, cada uno de los cuales es responsable del comportamiento de una de las regiones concurrentes. Si el comportamiento de uno de estos flujos concurrentes se ve afectado por el estado del otro, se puede modelar esto utilizando regiones ortogonales. Si el comportamiento de uno de estos flujos concurrentes se ve afectado por los mensajes enviados a y desde el otro, se puede modelar esto utilizando objetos activos. Si hay poca o ninguna comunicación entre los flujos concurrentes, entonces el método a utilizar es cuestión de gustos, aunque, la mayoría de las veces, el uso de objetos activos hace más evidente la decisión de diseño.

Técnicas comunes de modelado

Modelado de la vida de un objeto

La mayoría de las veces, las máquinas de estados se utilizan para modelar la vida de un objeto, especialmente instancias de clases, casos de uso y el sistema global. Mientras que las interacciones modelan el comportamiento de una sociedad de objetos que colaboran entre sí, una máquina de estados modela el comportamiento de un único objeto a lo largo de su vida, del tipo de los que aparecen cuando se modelan interfaces de usuario, controladores y dispositivos.

Cuando se modela la vida de un objeto, básicamente se especifican tres cosas: los eventos a los que puede responder el objeto, la respuesta a estos eventos y la influencia del pasado en el comportamiento actual. El modelado de la vida de un objeto también implica decidir en qué orden puede responder el objeto a los eventos, de forma significativa, desde el momento de la creación del objeto hasta su destrucción.

Para modelar la vida de un objeto:

- Hay que establecer el contexto de la máquina de estados, bien sea una clase, un caso de uso o el sistema global.
 - Si el contexto es una clase o un caso de uso, hay que considerar las clases vecinas, incluyendo las clases padre y cualquier clase alcanzable por asociaciones o dependencias. Estas clases vecinas son candidatas a ser destinos de las acciones y a ser incluidas en las condiciones de guarda.
 - Si el contexto es el sistema global, sólo se debe considerar uno de los comportamientos del sistema. Teóricamente, cada objeto del sistema puede participar en un modelo de la vida del sistema, y, excepto en los casos más triviales, un modelo completo sería intratatable.
- Hay que establecer los estados inicial y final del objeto. Para orientar el resto del modelo, es posible establecer las pre y poscondiciones de los estados inicial y final, respectivamente.
- Hay que decidir a qué eventos puede responder el objeto. Si ya han sido especificados, se encontrarán en las interfaces de los objetos; si no lo han sido aún, habrá que considerar qué objetos pueden interactuar con el objeto en el contexto que se trata, y después qué eventos se pueden generar.
- Hay que diseñar los estados de alto nivel en los que puede estar el objeto, comenzando con el estado inicial y acabando en el estado final. Estos estados se conectarán con transiciones disparadas por los eventos apropiados. A continuación, se añaden las acciones a estas transiciones.
- Hay que identificar cualquier acción de entrada o de salida (especialmente si se observa que la construcción que abarca se utiliza en la máquina de estados).
- Hay que expandir los estados con subestados si es necesario.
- Hay que comprobar que todos los eventos mencionados en la máquina de estados están incluidos en el conjunto de posibles eventos proporcionados.

Los objetos se discuten en el Capítulo 13; las clases se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; los sistemas se discuten en el Capítulo 32; las interacciones se discuten en el Capítulo 16; las colaboraciones se discuten en el Capítulo 28; las precondiciones y las postcondiciones se discuten en el Capítulo 10; las interfaces se discuten en el Capítulo 11.

nado por la interfaz del objeto. Análogamente, hay que comprobar que la máquina de estados maneja todos los posibles eventos según la interfaz del objeto. Por último, se deben considerar los lugares donde explícitamente se quiera prescindir de los eventos.

- Hay que comprobar que todas las acciones mencionadas en la máquina de estados están soportadas por las relaciones, métodos y operaciones del objeto que la contiene.
- Hay que recorrer la máquina de estados, ya sea manualmente o con herramientas, para comprobar las secuencias esperadas de eventos y sus respuestas. Estar especialmente atento a los estados inalcanzables y los estados en los que la máquina se puede quedar indefinidamente.
- Después de reorganizar la máquina de estados, hay que contrastarla con las secuencias esperadas de nuevo, para asegurarse de que no se ha cambiado la semántica del objeto.

Por ejemplo, la Figura 22.9 muestra la máquina de estados del controlador de un sistema de seguridad doméstico, responsable de controlar varios sensores distribuidos por el exterior de la casa.

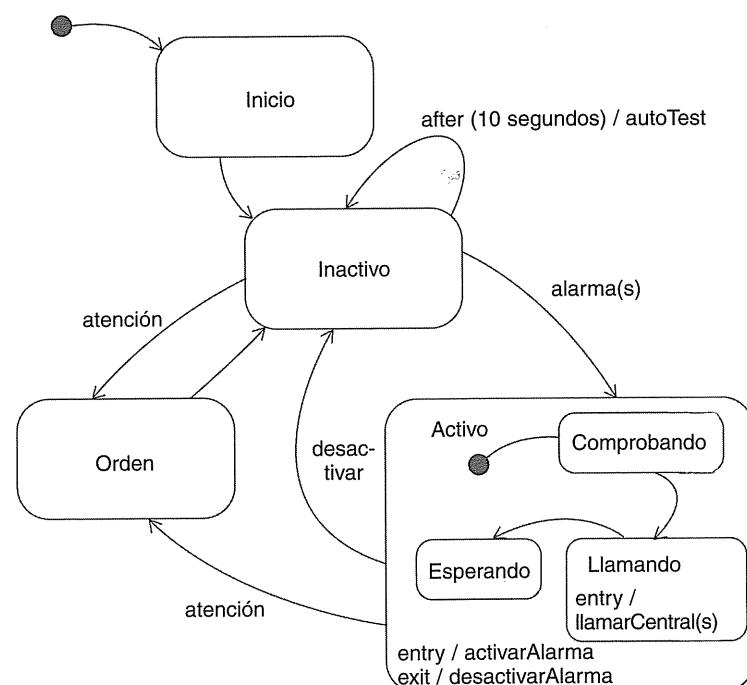


Figura 22.9: Modelado de la vida de un objeto.

En la vida de esta clase controladora hay cuatro estados principales: **Inicio** (el controlador está poniéndose en marcha), **Inactivo** (el controlador está preparado y esperando una señal de alarma u órdenes del usuario), **Orden** (el controlador está procesando órdenes del usuario) y **Activo** (el controlador está procesando una alarma). Cuando se crea por primera vez un objeto controlador, pasa al estado **Inicio** y luego de forma incondicional al estado **Inactivo**. No se muestran los detalles de estos dos estados, excepto la autotransición con el evento de tiempo en el estado **Inactivo**. Este tipo de evento de tiempo es frecuente en los sistemas embebidos, que a menudo suelen tener un temporizador que provoca un chequeo periódico del estado del sistema.

El control pasa del estado **Inactivo** al estado **Activo** cuando se recibe un evento **alarma** (que incluye el parámetro **s**, el cual identifica al sensor que ha disparado la alarma). Al entrar en el estado **Activo**, se ejecuta **activarAlarma** como acción de entrada, y el control pasa primero al estado **Comprobando** (que valida la alarma), luego al estado **Llamando** (que llama a la empresa de seguridad para registrar la alarma) y, por último, al estado **Esperando**. Sólo se puede salir de los estados **Activo** y **Esperando** al **desactivar** la alarma, o cuando el usuario envíe una señal de atención, presumiblemente para dar una orden.

Nótese que no hay estado final. Esto también es frecuente en los sistemas embebidos, concebidos para funcionar continuamente.

Sugerencias y consejos

Cuando se modelan máquinas de estados en UML, debe recordarse que cada máquina de estados representa los aspectos dinámicos de un objeto individual, que normalmente representa una instancia de una clase, un caso de uso o el sistema global. Una máquina de estados bien estructurada:

- Es sencilla y, por lo tanto, no contiene ningún estado o transición superfluo.
- Tiene un contexto claro y, por lo tanto, debe tener acceso a todos los objetos visibles dentro del objeto que la contiene (estos vecinos sólo deben utilizarse si son necesarios para desarrollar el comportamiento especificado por la máquina de estados).
- Es eficiente y debería llevar a cabo su comportamiento con un equilibrio óptimo de tiempo y recursos según sean requeridos por las acciones que se ejecuten.

El modelado del vocabulario de un sistema se discute en el Capítulo 4.

- Es comprensible y por ello debe dar nombres a los estados y a las transiciones a partir del vocabulario del sistema.
- No contiene muchos niveles de anidamiento (uno o dos niveles de subestados suelen ser suficientes para manejar la mayoría de los comportamientos complejos).
- Utiliza con moderación las regiones ortogonales, ya que una mejor alternativa a menudo es utilizar clases activas.

Cuando se dibuje una máquina de estados en UML:

- Hay que evitar los cruces de transiciones.
- Hay que expandir los estados compuestos sólo donde sea necesario para hacer más comprensible el diagrama.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 23

PROCESOS E HILOS

En este capítulo

- Objetos activos, procesos e hilos.
- Modelado de múltiples flujos de control.
- Modelado de la comunicación entre procesos.
- Construcción de abstracciones con hilos seguros (*thread-safe*).

Las vistas de interacción en el contexto de la arquitectura del software se discuten en el Capítulo 2.

El mundo real no es sólo un lugar cruel y despiadado, sino también un lugar con mucho ajetreo. Pueden ocurrir eventos y pueden pasar otras cosas, todo ello al mismo tiempo. Por lo tanto, cuando se modela un sistema del mundo real, hay que tener en cuenta su vista de procesos, la cual incluye los hilos y procesos que forman los mecanismos de sincronización y concurrencia del sistema.

En UML, cada flujo de control independiente se modela como un objeto activo que representa un proceso o hilo que puede iniciar actividad de control. Un proceso es un flujo *pesado* que puede ejecutarse concurrentemente con otros procesos; un hilo es un flujo *ligero* que puede ejecutarse concurrentemente con otros hilos dentro de un mismo proceso.

La construcción de abstracciones que funcionen fiablemente en presencia de múltiples flujos de control es difícil. En particular, hay que considerar métodos para la comunicación y la sincronización más complejos que para los sistemas secuenciales. También hay que tener cuidado de no sobrecargar la vista de procesos (demasiados flujos concurrentes darán al traste con el sistema) ni quedarse corto al diseñar (una concurrencia insuficiente no optimizará el rendimiento del sistema).

Introducción

El modelado de casetas de perro y rascacielos se discute en el Capítulo 1.

Para un perro que vive en su caseta, el mundo es un lugar muy simple y secuencial. Comer. Dormir. Perseguir a un gato. Comer algo más. Soñar con perseguir a gatos. Nunca es un problema usar la caseta para dormir o para protegerse de la lluvia, ya que el perro y sólo el perro necesita cruzar la puerta de la caseta para entrar y salir. Nunca hay una competición por los recursos.

En la vida de una familia y su casa el mundo no es tan sencillo. No nos pondremos metafísicos, pero cada miembro de la familia vive su propia vida, aunque interactúa con los otros (para comer, ver la televisión, jugar, limpiar). Los miembros de la familia comparten ciertos recursos. Los niños pueden compartir una habitación. La familia entera puede compartir un teléfono o un computador. También se comparten las tareas. El padre hace la colada y la compra; la madre lleva las cuentas y se ocupa del jardín; los niños ayudan en la limpieza y la cocina. La competición por estos recursos compartidos y la coordinación entre las tareas independientes puede ser todo un reto. Compartir el cuarto de baño cuando todo el mundo está preparado para irse al colegio o al trabajo puede ser problemático; la cena no se servirá si el padre no ha hecho antes la compra.

En la vida de los inquilinos de un gran edificio, el mundo es realmente complejo. Cientos, si no miles, de personas pueden trabajar en el mismo edificio, cada uno llevando su propio horario. Todos deben pasar a través de un número limitado de entradas. Todos deben utilizar el mismo conjunto de ascensores. Todos deben compartir los mismos servicios de calefacción, refrigeración, agua, electricidad, sanitarios y aparcamientos. Si quieren trabajar juntos de forma óptima, deben comunicarse y sincronizar sus interacciones adecuadamente.

Los objetos se discuten en el Capítulo 13.

En UML cada flujo de control independiente se modela como un objeto activo. Un objeto activo es un proceso o un hilo que puede iniciar actividad de control. Al igual que los demás objetos, un objeto activo es una instancia de una clase. En este caso, un objeto activo es una instancia de una clase activa. También al igual que los demás objetos, los objetos activos pueden comunicarse entre sí mediante el paso de mensajes, aunque aquí el paso de mensajes debe extenderse con cierta semántica de concurrencia, para ayudar a sincronizar las interacciones entre flujos independientes.

En el terreno del software, muchos lenguajes de programación soportan directamente el concepto de objeto activo. Java, Smalltalk y Ada incorporan concurrencia. C++ soporta la concurrencia a través de varias bibliotecas que se basan en los mecanismos de concurrencia del sistema operativo. Es importante utilizar UML para visualizar, especificar, construir y documentar estas abstracciones.

nes, porque si no se hace así, es casi imposible razonar sobre cuestiones de concurrencia, comunicación y sincronización.

Las clases se discuten en los Capítulos 4 y 9; las señales se discuten en el Capítulo 21.

UML proporciona una representación gráfica para una clase activa, como se muestra en la Figura 23.1. Las clases activas son un tipo de clases, de forma que tienen los comportamientos habituales de las clases para el nombre, los atributos y las operaciones. Las clases activas a menudo reciben señales, que normalmente se enumerarán en un compartimento extra.

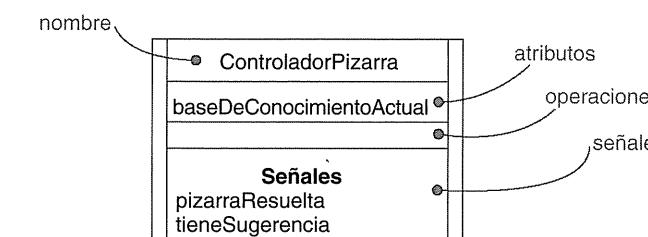


Figura 23.1: Clase activa.

Términos y conceptos

Los diagramas de interacción se discuten en el Capítulo 19.

Un *objeto activo* es un objeto que tiene un proceso o hilo y puede iniciar actividad de control. Una *clase activa* es una clase cuyas instancias son objetos activos. Un *proceso* es un flujo pesado que se puede ejecutar concurrentemente con otros procesos. Un *hilillo* es un flujo ligero que se puede ejecutar concurrentemente con otros hilos dentro del mismo proceso. Gráficamente, una clase activa se representa con un rectángulo con líneas dobles a la derecha e izquierda. Los procesos e hilos se representan como clases activas estereotipadas (y también aparecen como secuencias en los diagramas de interacción).

Flujo de control

En un sistema totalmente secuencial hay un único flujo de control. Esto significa que puede tener lugar una y sólo una única cosa en un momento dado. Cuando un programa secuencial comienza, el control parte del principio del programa y las operaciones se ejecutan una tras otra. Incluso si hay elementos concurrentes entre los actores externos al sistema, un programa secuencial sólo procesará un evento en cada momento, añadiendo a una cola o descartando cualquier evento externo concurrente.

Los actores se discuten en el Capítulo 17.

Las acciones se discuten en el Capítulo 16.

Por eso se llama flujo de control. Si se sigue la ejecución de un programa secuencial, se puede ver el lugar por el que pasa el flujo de ejecución de una instrucción a otra, en orden secuencial. Se podrán ver bifurcaciones, ciclos y saltos y, si hay recursión o iteración, se verá que el flujo vuelve a empezar sobre sí mismo. No obstante, en un sistema secuencial, habrá un único flujo de ejecución.

En un sistema concurrente, hay más de un flujo de control (es decir, puede ocurrir más de una cosa a la vez). En un sistema concurrente hay varios flujos de control simultáneos, cada uno originado en un proceso o hilo independiente. Si se toma una instantánea de un sistema concurrente mientras se está ejecutando, se observarán varios lugares donde se encuentra localizada la ejecución.

Los nodos se discuten en el Capítulo 27.

En UML se utilizan las clases activas para representar un proceso o hilo que constituye la raíz de un flujo de control independiente, que será concurrente con otros flujos de control.

Nota: Se puede lograr una concurrencia verdadera de tres formas: primera, distribuyendo objetos activos entre varios nodos; segunda, colocando objetos activos en nodos con varios procesadores; y tercera, con una combinación de ambos métodos.

Clases y eventos

Las clases se discuten en los Capítulos 4 y 9.

Las clases activas son clases, aunque con una propiedad muy especial. Una clase activa representa un flujo de control independiente, mientras que una clase normal no incluye ese flujo. En contraste con las clases activas, las clases normales se llaman pasivas de forma implícita, porque no pueden iniciar actividad de control.

Los objetos se discuten en el Capítulo 13; los atributos y las operaciones se discuten en el Capítulo 4; las relaciones se discuten en los Capítulos 4 y 10; los mecanismos de extensibilidad se discuten en el Capítulo 6; las interfaces se discuten en el Capítulo 11.

Las clases activas se utilizan para modelar familias frecuentes de procesos e hilos. En términos técnicos, esto significa que un objeto activo (una instancia de una clase activa) materializa (es una manifestación de) un proceso o hilo. Cuando se modelan sistemas concurrentes con objetos activos, se le da un nombre a cada flujo de control independiente. Cuando se crea un objeto activo, se inicia el flujo de control asociado; cuando se destruye el objeto activo, se destruye el flujo de control asociado.

Las clases activas comparten las mismas características que las otras clases. Las clases activas pueden tener instancias. Las clases activas pueden tener atributos y operaciones. Las clases activas pueden participar en relaciones de dependencia, generalización y asociación (incluyendo agregación). Las clases activas pueden utilizar cualquier mecanismo de extensibilidad de UML, incluyendo los

Las máquinas de estados se discuten en el Capítulo 22; los eventos se discuten en el Capítulo 21.

estereotipos, los valores etiquetados y las restricciones. Las clases activas pueden proporcionar la realización de interfaces. Las clases activas pueden ser realizadas por colaboraciones, y el comportamiento de una clase activa puede especificarse con máquinas de estados. Las clases activas pueden participar en colaboraciones.

En los diagramas, los objetos activos pueden aparecer siempre que pueden aparecer los objetos pasivos. La colaboración de objetos activos y pasivos se puede de modelar con diagramas de interacción (incluyendo los diagramas de secuencia y comunicación). Un objeto activo puede aparecer como destinatario de un evento en una máquina de estados.

Hablando de máquinas de estados, tanto los objetos activos como los pasivos pueden enviar y recibir eventos de señal y eventos de llamada.

Nota: El uso de clases activas es opcional. En realidad, no añaden mucho a la semántica.

Comunicación

Las interacciones se discuten en el Capítulo 16.

Cuando los objetos colaboran entre sí, interactúan pasándose mensajes. En un sistema con objetos activos y pasivos, hay cuatro posibles combinaciones de interacción que se deben considerar.

Los eventos de señal y los eventos de llamada se discuten en el Capítulo 21.

En primer lugar, un mensaje puede ir de un objeto pasivo a otro. Suponiendo que en cada momento sólo hay un flujo de control entre estos objetos, esta interacción no es otra cosa que la simple invocación de una operación.

En segundo lugar, un mensaje puede pasar de un objeto activo a otro. Cuando esto ocurre, se da una comunicación entre procesos, y hay dos estilos de comunicación posibles. En el primer estilo, un objeto activo puede invocar una operación de otro de forma síncrona. Este tipo de comunicación tiene una semántica de *rendezvous*, lo que significa que el objeto emisor llama a la operación, espera a que el receptor acepte la llamada, se invoca la operación, se devuelve un objeto de retorno (si lo hay) al objeto emisor, y entonces los dos continúan sus caminos independientes. Durante la duración de la llamada, los dos flujos de control están bloqueados. En el segundo estilo, un objeto activo puede enviar una señal o invocar una operación de otro objeto de forma asíncrona. Este tipo de comunicación tiene una semántica de buzón, lo que significa que el objeto emisor envía la señal o llama a la operación y luego continúa por su camino independiente. Mientras tanto, el receptor acepta la señal o la llamada cuando está pre-

parado (teniendo en cola los eventos o las llamadas implicados) y continúa por su lado después de atenderla. Esto se conoce como *buzón* porque los dos objetos no están sincronizados; en vez de ello, un objeto deja un mensaje para el otro.

En UML, un mensaje síncrono se representa con una flecha con la punta rellena, y un mensaje asíncrono con una flecha con la punta sin cerrar, como se muestra en la Figura 23.2.

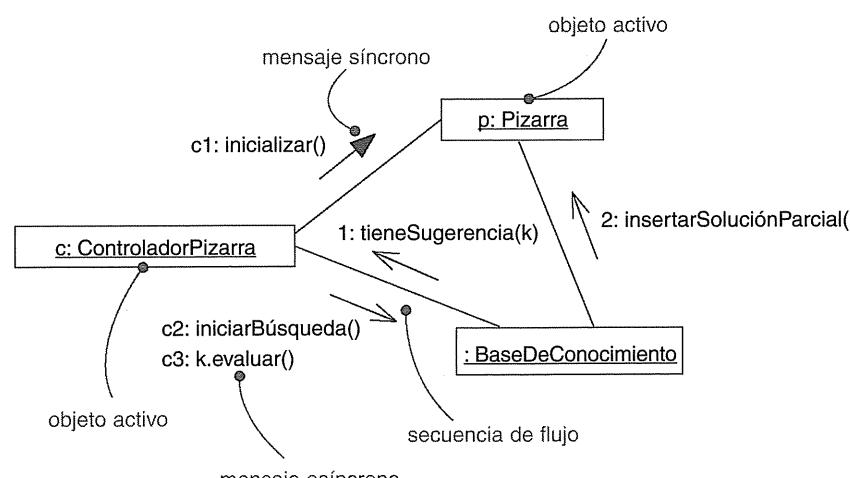


Figura 23.2: Comunicación.

En tercer lugar, un mensaje puede pasar de un objeto activo a un objeto pasivo. Puede aparecer un conflicto potencial si varios objetos activos hacen pasar a la vez su flujo de control a través de un objeto pasivo. Es un conflicto real si más de un objeto lee o escribe los mismos atributos. En esa situación, hay que modelar la sincronización de esos dos flujos muy cuidadosamente, como se discute en la siguiente sección.

Las restricciones se discuten en el Capítulo 6.

En cuarto lugar, un mensaje puede pasar de un objeto pasivo a un objeto activo. A primera vista esto puede parecer ilegal, pero si se recuerda que cada flujo de control parte de algún objeto activo, se entenderá que un objeto pasivo que pase un mensaje a un objeto activo tiene la misma semántica que un objeto activo que pase un mensaje a otro objeto activo.

Nota: Es posible modelar variaciones de paso de mensajes síncronos y asíncronos mediante el uso de restricciones. Por ejemplo, para modelar una sincronización sin espera (*balking*) como las que existen en Ada, se puede utilizar un mensaje síncrono con una restric-

ción, tal como `{wait = 0}`, que expresa que el emisor no esperará al receptor. Análogamente, se puede modelar una sincronización con límite de tiempo (*timeout*) con una restricción como `{wait = 1 ms}`, que indica que el emisor no esperará más de un milisegundo para que el receptor acepte el mensaje.

Sincronización

Imaginemos por un momento los múltiples flujos de control de un sistema concurrente. Cuando el flujo pasa a través de una operación, se dice que en un momento dado el lugar donde se encuentra el control es la operación. Si esa operación está definida en alguna clase, también se dice que en un momento dado el lugar donde se encuentra el control es una instancia específica de esa clase. Puede haber varios flujos de control en una misma operación (y por lo tanto en un objeto), y puede haber diferentes flujos de control en diferentes operaciones (pero que todavía producen varios flujos de control en el objeto).

El problema aparece cuando en un instante dado existe más de un flujo de control en un objeto. Si no se tiene cuidado, los flujos pueden interferir entre sí, corrompiendo el estado del objeto. Éste es el problema clásico de la exclusión mutua. Si no se trata correctamente, se producen toda clase de condiciones de competencia e interferencias, que hacen que los sistemas concurrentes fallen de formas misteriosas e irrepetibles.

La clave para resolver este problema es serializar el acceso al objeto crítico. Hay tres alternativas a este enfoque, cada una de las cuales implica asociar ciertas propiedades de sincronización a las operaciones definidas en la clase. En UML se pueden modelar los tres enfoques.

1. Secuencial Los invocadores deben coordinarse fuera del objeto para que en un instante dado sólo exista un flujo de control en el objeto. En presencia de varios flujos de control, no se puede garantizar la semántica ni la integridad del objeto.
2. Con guardas La semántica y la integridad del objeto se garantizan en presencia de múltiples flujos de control, tratando secuencialmente todas las llamadas a las operaciones con guardas del objeto. En efecto, en un momento dado sólo puede ser invocada una operación sobre el objeto, lo que

lleva a una semántica secuencial. Si no se tiene cuidado, se corre el riesgo de producir un bloqueo mortal (*dead-lock*).

3. Concurrente La semántica y la integridad del objeto se garantizan en presencia de múltiples flujos de control porque cada flujo accede a conjuntos disjuntos de datos o sólo leen datos. Esta situación puede conseguirse con unas reglas de diseño cuidadosas.

Las restricciones se discuten en el Capítulo 6.

Algunos lenguajes de programación soportan estas construcciones directamente. Java, por ejemplo, tiene la propiedad `synchronized`, equivalente a la propiedad `concurrent` de UML. En cualquier lenguaje con mecanismos de concurrencia, se pueden utilizar semáforos para construir un software que dé soporte a todas estas propiedades.

Como se muestra en la Figura 23.3, se pueden asociar estas propiedades a una operación, utilizando la notación de restricciones de UML para su representación. Hay que tener en cuenta que la concurrencia debe especificarse por separado para cada operación y para el objeto completo. El especificar la concurrencia para una operación significa que varias invocaciones de esa operación pueden ejecutarse concurrentemente sin peligro. Especificar la concurrencia para un objeto significa que las invocaciones de diferentes operaciones pueden ejecutarse concurrentemente sin peligro. Esta última es una condición más fuerte.

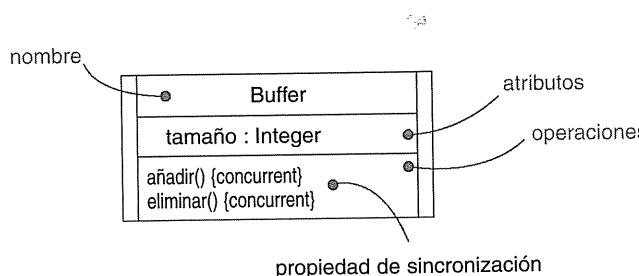


Figura 23.3: Sincronización.

Nota: Se pueden modelar variaciones de estas primitivas de sincronización con restricciones. Por ejemplo, se podría modificar la propiedad `concurrent` permitiendo varios lectores simultáneos pero sólo un escritor.

Técnicas comunes de modelado

Modelado de múltiples flujos de control

Los mecanismos se discuten en el Capítulo 29; los diagramas de clases se discuten en el Capítulo 8; los diagramas de interacción se discuten en el Capítulo 19.

La construcción de un sistema que incluya varios flujos de control es difícil. No sólo hay que decidir la mejor forma de dividir el trabajo entre los objetos activos concurrentes, sino que, una vez hecho esto, también hay que idear los mecanismos apropiados para la comunicación y la sincronización entre los objetos activos y pasivos del sistema, para asegurarse de que se comportarán adecuadamente en presencia de esos diferentes flujos. Por esa razón, visualizar la forma en que interactúan unos flujos con otros es de gran ayuda. Esto se puede hacer en UML empleando diagramas de clases (para capturar la semántica estática) y diagramas de interacción (para capturar la semántica dinámica) con clases y objetos activos.

Para modelar múltiples flujos de control:

Las vistas de procesos se discuten en el Capítulo 19; las clases se discuten en los Capítulos 4 y 9; las relaciones se discuten en los Capítulos 5 y 10.

- Hay que identificar qué acciones concurrentes son posibles y materializar cada flujo como una clase activa. Un conjunto de objetos activos con propiedades comunes se debe generalizar en una clase activa. Hay que tener cuidado para no sobrecargar el diseño de la vista de procesos del sistema introduciendo demasiada concurrencia.
- Hay que considerar una distribución de responsabilidades equilibrada entre esas clases activas, y luego examinar las otras clases activas y pasivas con las que cada una colabora estáticamente. Hay que asegurarse de que cada clase activa es bastante cohesiva y está poco acoplada con relación a las clases vecinas, y que cada una tiene el conjunto apropiado de atributos, operaciones y señales.
- Hay que capturar estas decisiones estáticas en diagramas de clases, destacando explícitamente cada clase activa.
- Hay que considerar cómo colabora dinámicamente cada grupo de clases con las demás. Estas decisiones se capturarán en diagramas de interacción. Los objetos activos deben mostrarse explícitamente como las raíces de tales flujos. Hay que identificar cada secuencia relacionada, asignándole el nombre del objeto activo.
- Hay que prestar una atención especial a la comunicación entre objetos activos. Se debe aplicar el paso de mensajes de forma síncrona o asíncrona, según convenga en cada caso.

- Hay que prestar una atención especial a la sincronización entre esos objetos activos y aquellos objetos pasivos con los que colaboran. Se debe aplicar la semántica de operaciones secuencial, con guardas o concurrente, según convenga en cada caso.

Por ejemplo, la Figura 23.4 muestra parte de la vista de procesos de un sistema financiero. Hay tres objetos que introducen información en el sistema de forma concurrente: un **ObservadorBolsa**, un **ControladorIndices** y un objeto **InformaciónCNN** (llamados **o**, **c** e **i**, respectivamente). Dos de estos objetos (**o** y **c**) se comunican con sus propias instancias de **Analista** (**a1** y **a2**). Al menos hasta donde llega este modelo, el **Analista** se puede diseñar bajo la suposición simplificadora de que en sus instancias sólo estará activo un flujo de control en un momento dado. Sin embargo, ambas instancias de **Analista** se comunican simultáneamente con un **GestorAlertas** (llamado **g**). Por lo tanto, **g** debe ser diseñado de forma que se preserve su semántica en presencia de múltiples flujos. Tanto **g** como **i** se comunican simultáneamente con **t**, un **AgenteComercial**. A cada flujo se le asigna un número de secuencia, que se distingue precediéndolo del nombre del flujo de control al que pertenece.

Nota: Los diagramas de interacción de esta naturaleza son útiles para ayudar a visualizar dónde pueden cruzarse dos flujos de control y, por lo tanto, dónde hay que prestar particular atención a los problemas de comunicación y sincronización. Se permite que las herramientas ofrezcan incluso señales visuales distintas, como colorear cada flujo de una forma diferente.

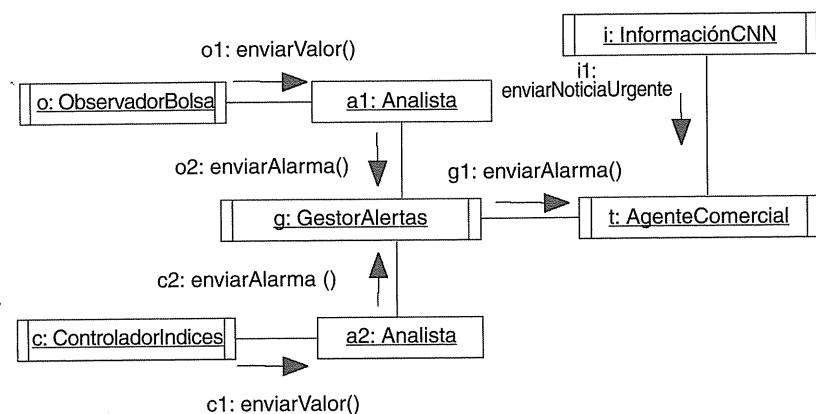


Figura 23.4: Modelado de flujos de control.

Las máquinas de estados se discuten en el Capítulo 22.

También es frecuente asociar las correspondientes máquinas de estados a diagramas como éste, en las que los estados ortogonales muestran el comportamiento detallado de cada objeto activo.

Modelado de la comunicación entre procesos

Los eventos de señal y los eventos de llamada se discuten en el Capítulo 21.

La incorporación de múltiples flujos de control en un sistema también conlleva el tener que considerar los mecanismos que permiten a objetos de diferentes flujos comunicarse entre sí. Los objetos de hilos diferentes (los cuales se encuentran en el mismo espacio de direcciones) pueden comunicarse por medio de señales o eventos de llamada, donde estos últimos pueden exhibir una semántica tanto síncrona como asíncrona. La comunicación entre objetos de diferentes procesos (que se encuentran en espacios de direcciones separados) requiere normalmente la utilización de otros mecanismos.

El modelado de la localización se discute en el Capítulo 24.

En los sistemas distribuidos, el problema de la comunicación entre procesos se complica con el hecho de que los procesos pueden encontrarse en nodos separados. Existen dos enfoques clásicos a la comunicación entre procesos: paso de mensajes y llamadas de procedimientos remotos. En UML, estos enfoques se modelan como eventos asíncronos o síncronos, respectivamente. Pero al no tratarse ya de simples llamadas dentro de un proceso, es necesario adornar los diseños con información adicional.

Para modelar la comunicación entre procesos:

- Hay que modelar los múltiples flujos de control.
- Hay que modelar el paso de mensajes con comunicación asíncrona; hay que modelar las llamadas a procedimientos remotos con comunicación síncrona.
- Hay que especificar informalmente mediante notas el mecanismo subyacente a la comunicación, o más formalmente con colaboraciones.

Los estereotipos se discuten en el Capítulo 6; las notas se discuten en el Capítulo 6; las colaboraciones se discuten en el Capítulo 28; los nodos se discuten en el Capítulo 27.

La Figura 23.5 representa un sistema distribuido de reservas que funciona repartido entre cuatro nodos. Cada objeto está marcado con el estereotipo **process**. Cada objeto también está marcado con un valor etiquetado **location**, que especifica su localización física. La comunicación entre el **AgenteReservas**, el **GestorBilletes** y el **AgenteHotel** es asíncrona. Modelado mediante una nota, se expresa que la comunicación se basa en un servicio de mensajes de los Java Beans. La comunicación entre el **PlanificadorDeViajes** y el **AgenteReservas** es síncrona. La semántica de su interacción se encuentra

en la colaboración llamada CORBA ORB. El PlanificadorDeViajes actúa como cliente, y el AgenteReservas actúa como servidor. Al mirar dentro de la colaboración, se pueden ver los detalles de la colaboración entre este servidor y el cliente.

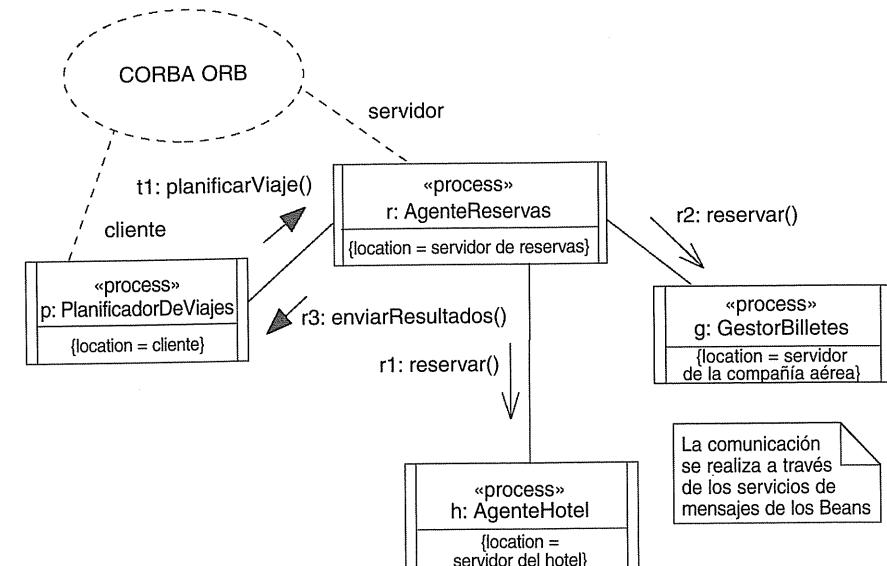


Figura 23.5: Modelado de la comunicación entre procesos.

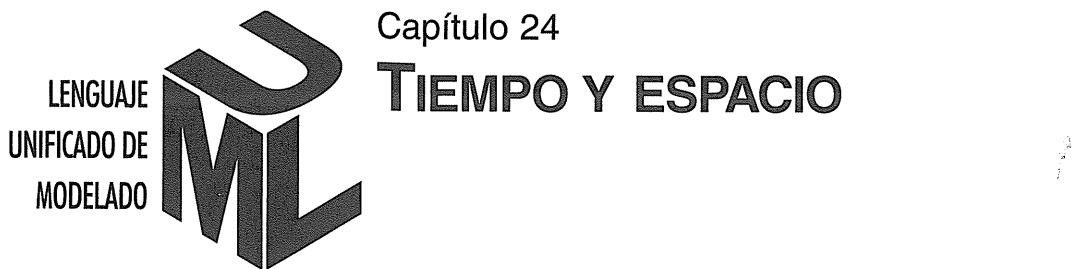
Sugerencias y consejos

Una clase activa y un objeto activo bien estructurados:

- Representan un flujo de control independiente que maximiza el potencial de concurrencia real del sistema.
- No deben ser de granularidad tan fina que requieran muchos otros elementos activos, lo que podría producir una arquitectura de procesos frágil y diseñada en exceso.
- Manejan adecuadamente la comunicación entre elementos activos del mismo nivel, eligiendo entre el paso de mensajes síncrono o asíncrono.
- Tratan adecuadamente a cada objeto como una región crítica, utilizando las propiedades de sincronización adecuadas para preservar su semántica en presencia de múltiples flujos de control.

Cuando se dibuje una clase activa o un objeto activo en UML:

- Hay que mostrar sólo aquellos atributos, operaciones y señales que sean importantes para comprender la abstracción en su contexto, y hay que ocultar los no importantes usando filtros, si la herramienta de modelado lo permite.
- Hay que mostrar explícitamente todas las propiedades de sincronización de las operaciones.



Capítulo 24

TIEMPO Y ESPACIO

En este capítulo

- Tiempo, duración y localización.
- Modelado de restricciones de tiempo.
- Modelado de la distribución de objetos.
- Modelado de objetos que migran.
- Sistemas de tiempo real y sistemas distribuidos.

El mundo real es un lugar cruel y despiadado. En cualquier instante impredecible, puede suceder un evento que exija una respuesta específica en un lapso de tiempo concreto. Los recursos de un sistema podrían estar distribuidos por todo el mundo (puede que incluso algunos de ellos se muevan de un sitio a otro), dando lugar a cuestiones de latencia, sincronización, seguridad y calidad de servicio.

El modelado del tiempo y el espacio es un elemento fundamental de cualquier sistema de tiempo real y/o distribuido. En este capítulo, se utilizarán varias características de UML, entre las que se incluyen marcas de tiempo, expresiones de tiempo, restricciones y valores etiquetados, para visualizar, especificar, construir y documentar estos sistemas.

Es difícil modelar sistemas de tiempo real y distribuidos. Los modelos buenos revelan las propiedades relacionadas con las características temporales y espaciales de un sistema.

Introducción

En la mayoría de los sistemas software, cuando se empieza a modelar, normalmente se puede suponer un entorno sin conflictos (los mensajes se envían en un tiempo cero, las redes nunca se caen, las estaciones de trabajo nunca fallan, la

carga a través de la red siempre está equilibrada). Desgraciadamente, el mundo real no funciona así; los mensajes tardan tiempo en enviarse (a veces nunca llegan), las redes se caen, las estaciones de trabajo fallan, y la carga de una red suele estar desequilibrada. Por lo tanto, cuando se está ante sistemas que deben funcionar en el mundo real, hay que tener en cuenta las cuestiones de tiempo y espacio.

Un sistema de tiempo real es aquel que debe llevar a cabo cierto comportamiento en un lapso de tiempo absoluto o relativo preciso, y con una duración predecible y, a menudo, limitada. En un extremo, están los sistemas de tiempo real muy exigentes, que pueden requerir un comportamiento repetible y completo con un tiempo de respuesta del orden de nanosegundos o milisegundos. En el otro extremo, los modelos pueden ser menos estrictos, y requerir un comportamiento también predecible, pero con un tiempo de respuesta del orden de segundos o por encima.

Los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27.

Un sistema distribuido es aquel cuyos componentes pueden encontrarse distribuidos físicamente entre varios nodos. Estos nodos pueden representar diferentes procesadores, físicamente ubicados en el mismo equipo, o pueden representar computadores que se encuentren separados por miles de kilómetros.

Para representar las necesidades de modelado de los sistemas de tiempo real y distribuidos, UML proporciona una representación gráfica para las marcas de tiempo, las expresiones de tiempo y las restricciones de tiempo y localización, como se muestra en la Figura 24.1.

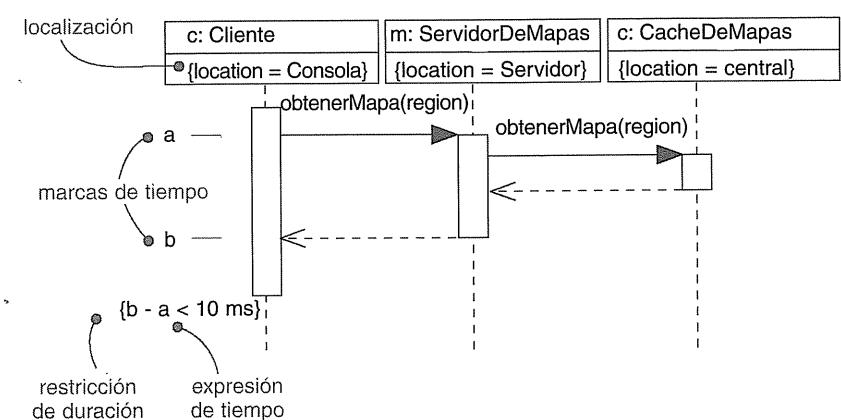


Figura 24.1: Restricciones de tiempo y localización.

Términos y conceptos

Una *marca de tiempo* denota el instante en el que ocurre un evento. Gráficamente, una marca de tiempo se representa como un pequeño guion (marca horizontal) en el borde de un diagrama de secuencia. Una *expresión de tiempo* es una expresión que al evaluarse genera un valor de tiempo absoluto o relativo. Una expresión de tiempo también puede formarse utilizando el nombre de un mensaje, y una indicación de una etapa en su procesamiento, por ejemplo, `solicitar.horaEnvio` o `solicitar.horaRecepcion`. Una *restricción de tiempo* es un enunciado semántico sobre el valor absoluto o relativo del tiempo. Gráficamente, una restricción de tiempo se representa como cualquier otra restricción, es decir, una cadena de texto entre llaves y normalmente conectada a un elemento mediante una relación de dependencia. La *localización* es la asignación de un componente a un nodo. La localización es un atributo de un objeto.

Tiempo

Los eventos, incluyendo los eventos de tiempo, se discuten en el Capítulo 21; los mensajes y las interacciones se discuten en el Capítulo 16; las restricciones se discuten en el Capítulo 6.

Los sistemas de tiempo real son, como su propio nombre indica, sistemas en los que el tiempo desempeña un papel crítico. Los eventos pueden ocurrir a intervalos de tiempo regulares o irregulares; la respuesta a un evento debe ocurrir en un lapso de tiempo predecible absoluto, o en un lapso de tiempo predecible, pero relativo al propio evento.

El paso de mensajes representa el aspecto dinámico de cualquier sistema; así, que cuando se modelan con UML los aspectos de un sistema en los que el tiempo de respuesta es un factor crítico, se puede dar un nombre a cada mensaje de una interacción, para poder utilizarlo en expresiones temporales. Los mensajes en una interacción no suelen tener nombres. Normalmente se representan con el nombre de un evento, tal como una señal o una llamada. Sin embargo, se les puede dar nombres para utilizarlos en expresiones de tiempo, ya que el mismo evento puede desencadenar diferentes mensajes. Si el mensaje designado es ambiguo, se puede utilizar el nombre explícito del mensaje para indicar qué mensaje se quiere incluir en una expresión de tiempo. Dado el nombre de un mensaje, se puede hacer referencia a cualquiera de sus tres funciones, es decir, `sendTime` (tiempo de envío), `receiveTime` (tiempo de recepción) y `transmissionTime` (tiempo de transmisión). (Estos nombres son sugerencias nuestras, no los nombres de funciones oficiales de UML. Un sistema de tiempo real podría incluso tener más funciones). Para las invocaciones síncronas, también se puede hacer referencia al tiempo de ida y vuelta como `executionTime` (tiempo de ejecución), lo que de nuevo es una sugerencia nuestra.

Estas tres funciones se pueden utilizar para especificar expresiones de tiempo arbitrariamente complejas, quizás incluso usando pesos o desplazamientos constantes o variables (siempre y cuando estas variables puedan ser ligadas en tiempo de ejecución). Por último, como se muestra en la Figura 24.2, esas expresiones de tiempo se pueden insertar en una restricción de tiempo para especificar el comportamiento temporal del sistema. Estas restricciones se pueden representar colocándolas junto al mensaje apropiado, o conectándolas explícitamente mediante una relación de dependencia.

Nota: Es una buena idea, especialmente en los sistemas complejos, escribir expresiones con constantes en vez de tiempos explícitos. Estas constantes se pueden definir en una parte del modelo y después se puede hacer referencia a ellas desde varios sitios. De esa forma, es más fácil actualizar el modelo si cambian los requisitos del sistema relacionados con el tiempo.

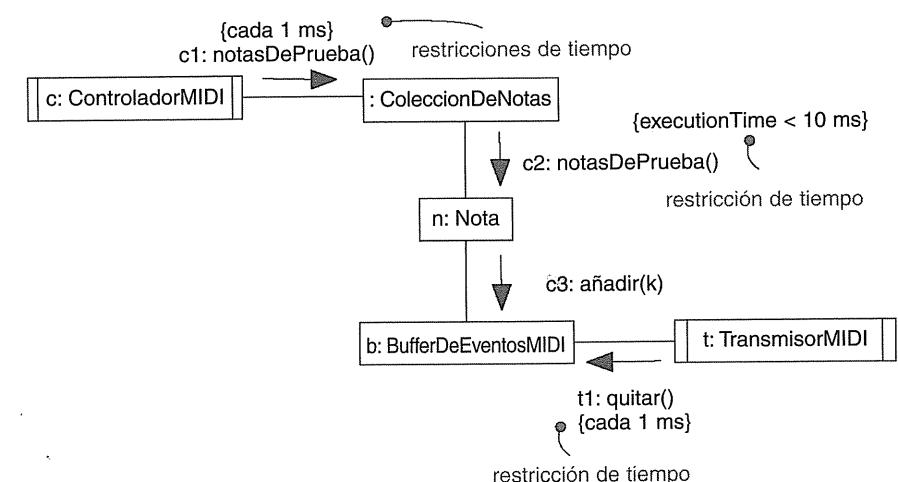


Figura 24.2: Tiempo.

Localización

Los sistemas distribuidos, por su naturaleza, incluyen componentes que se encuentran físicamente dispersos entre los nodos de un sistema. En muchos sistemas, los componentes se fijan a un nodo cuando son instalados en el sistema; en otros sistemas, los componentes pueden migrar de un nodo a otro.

En UML, la vista de despliegue de un sistema se modela mediante diagramas de despliegue, que representan la topología de los procesadores y dispositivos sobre los que se ejecuta el sistema. Los artefactos como los programas ejecutables, las bibliotecas y las tablas residen en estos nodos. Cada instancia de un nodo contendrá instancias de ciertos artefactos, y cada instancia de un artefacto estará contenida en una sola instancia de un nodo (aunque instancias del mismo tipo de artefacto pueden distribuirse por diferentes nodos). Por ejemplo, como se muestra en la Figura 24.3, la clase AgenteDeCarga se manifiesta en el artefacto inicializador.exe que reside en el nodo de tipo Enrutador.

Como se muestra en la figura, la localización de un elemento se puede modelar de dos formas en UML. En la primera, como se muestra en el Enrutador, el elemento se puede incluir físicamente (textual o gráficamente) en un compartimento extra del nodo que lo contiene. En la segunda, se puede utilizar una dependencia con la palabra clave «deploy» que une el artefacto con el nodo que lo contiene.

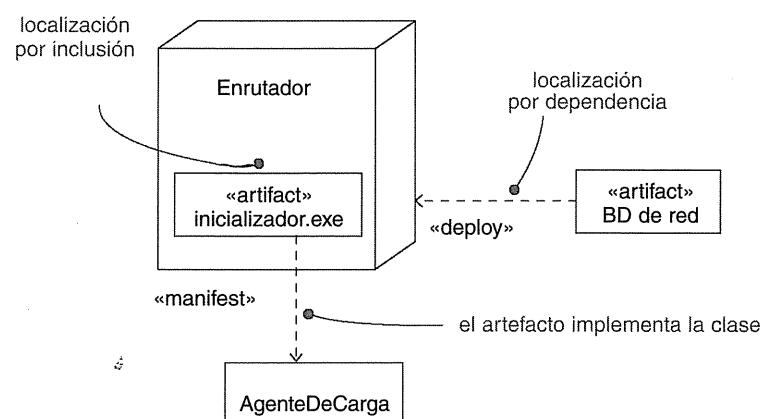


Figura 24.3: Localización.

Técnicas comunes de modelado

Modelado de restricciones de tiempo

Las restricciones, uno de los mecanismos de extensibilidad de UML, se discuten en el Capítulo 6.

Las principales propiedades relacionadas con el papel decisivo del tiempo en los sistemas de tiempo real para las que se utilizarán restricciones de tiempo son el modelado del tiempo absoluto de un evento y el modelado del tiempo relativo entre eventos.

Para modelar restricciones de tiempo:

- Hay que considerar si cada evento incluido en una interacción debe comenzar en un tiempo absoluto. Esta propiedad de tiempo real se puede modelar como una restricción de tiempo sobre el mensaje.
- Para cada secuencia interesante de mensajes en una interacción, hay que considerar si hay un tiempo máximo relativo asociado a ella. Hay que modelar esta propiedad de tiempo real como una restricción temporal sobre la secuencia.

Por ejemplo, como se muestra en la Figura 24.4, la restricción más a la izquierda especifica cada cuánto tiempo se produce, de forma periódica, el evento de llamada `refrescar`. Del mismo nodo, la restricción a la derecha especifica la duración máxima de las llamadas a `obtenerImagen`.

A menudo se elegirán nombres cortos para los mensajes, a fin de no confundirlos con nombres de operaciones.

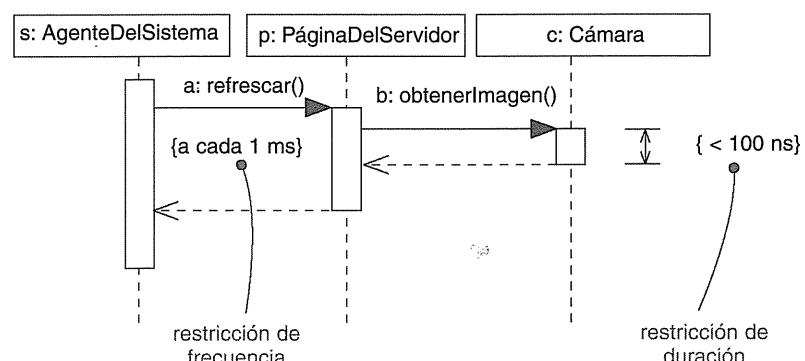


Figura 24.4: Modelado de restricciones de tiempo.

Modelado de la distribución de objetos

El modelado de la distribución de un componente se discute en el Capítulo 15.

Cuando se modela la topología de un sistema distribuido, hay que considerar la localización física, tanto de los nodos como de los artefactos. Si el objetivo es la gestión de la configuración del sistema desplegado, es importante el modelado de la distribución de nodos, para visualizar, especificar, construir y documentar la localización de elementos físicos tales como programas ejecutables, bibliotecas y tablas. Si el objetivo es la funcionalidad, escalabilidad y rendimiento del sistema, lo importante es el modelado de la distribución de objetos.

El modelado de los procesos e hilos se discute en el Capítulo 23.

Decidir cómo distribuir los objetos en un sistema es un problema difícil, y no sólo porque los problemas de la distribución interactúen con los problemas de la concurrencia. Las soluciones simplistas tienden a conseguir un rendimiento muy pobre, y las soluciones de diseños excesivamente complejos no son mucho mejores. De hecho, probablemente son peores, porque acaban siendo frágiles.

Para modelar la distribución de objetos:

- Para cada clase de objetos del sistema que desempeñe un papel importante, hay que considerar su localidad de referencia. En otras palabras, hay que considerar todos los objetos vecinos y su localización. Una localidad fuertemente acoplada colocará los objetos vecinos muy cerca; una poco acoplada colocará los objetos vecinos alejados (y así, habrá latencia al comunicarse con ellos). Hay que colocar los objetos, de forma tentativa, lo más cerca posible de los actores que los manipulan.
- Seguidamente, hay que considerar los patrones de interacción entre conjuntos relacionados de objetos. Colocar juntos los conjuntos de objetos con un alto grado de interacción, para reducir el coste de la comunicación. Hay que separar los conjuntos de objetos con un bajo grado de interacción.
- A continuación, hay que considerar la distribución de responsabilidades en el sistema. Hay que redistribuir los objetos para equilibrar la carga de cada nodo.
- Hay que considerar también cuestiones de seguridad, volatilidad y calidad de servicio, y hay que redistribuir los objetos según sea apropiado.
- Hay que asignar los objetos a los artefactos, de forma que los objetos altamente acoplados estén en el mismo artefacto.
- Hay que asignar los artefactos a los nodos, de forma que las necesidades de computación de cada nodo estén de acuerdo con su capacidad. Si hace falta, hay que añadir nodos adicionales.
- Hay que equilibrar los costes de rendimiento y de comunicación asignando los artefactos que estén estrechamente acoplados al mismo nodo.

Los diagramas de objetos se discuten en el Capítulo 14.

La Figura 24.5 proporciona un diagrama de objetos que modela la distribución de ciertos objetos en un sistema de ventas. La utilidad de este diagrama reside en que permite visualizar la distribución física de ciertos objetos clave. Como se aprecia en el diagrama, dos objetos se encuentran en una `EstaciónDeTrabajo` (los objetos `Pedido` y `Ventas`), otros dos objetos residen en un `Servidor` (los objetos `AgenteObservador` y `Producto`), y otro objeto se encuentra en un `AlmacénDeDatos` (el objeto `TablaDeProductos`).

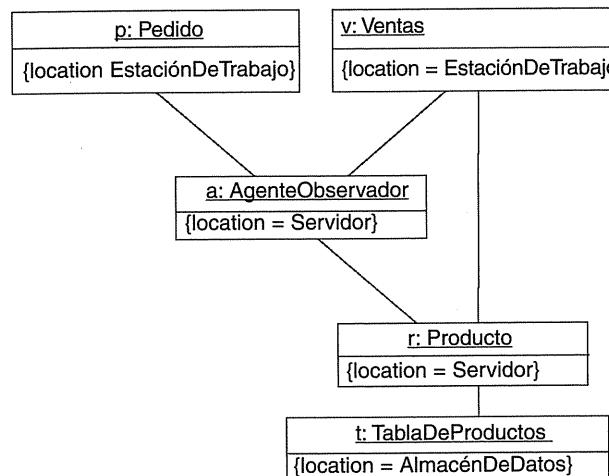


Figura 24.5: Modelado de la distribución de objetos.

Sugerencias y consejos

Un modelo bien estructurado con propiedades relacionadas con el tiempo y el espacio:

- Sólo muestra aquellas propiedades temporales y espaciales que sean necesarias y suficientes para capturar el comportamiento deseado del sistema.
- Centraliza el uso de esas propiedades para que sea fácil encontrarlas y modificarlas.

Cuando se dibuje una propiedad de tiempo o de espacio en UML:

- Hay que dar nombres significativos a las marcas de tiempo (los nombres de los mensajes).
- Hay que distinguir claramente entre expresiones de tiempo relativas y absolutas.
- Hay que mostrar las propiedades espaciales sólo cuando sea importante visualizar la localización de elementos a través del sistema desplegado.
- Para las necesidades más avanzadas, debe considerarse el uso del *Perfil UML para Planificación, Rendimiento y Tiempo*. Esta especificación del OMG cubre las necesidades de los sistemas de tiempo real y de alto rendimiento.



Capítulo 25 DIAGRAMAS DE ESTADOS

En este capítulo

- Modelado de objetos reactivos.
- Ingeniería directa e inversa.

Los diagramas de estados son uno de los cinco tipos de diagramas de UML que se utilizan para modelar los aspectos dinámicos de un sistema. Un diagrama de estados muestra una máquina de estados. Tanto los diagramas de actividades como los diagramas de estados son útiles para modelar la vida de un objeto. Sin embargo, mientras que un diagrama de actividades muestra el flujo de control entre actividades a través de varios objetos, un diagrama de estados muestra el flujo de control entre estados dentro de un único objeto.

Los diagramas de secuencia, de comunicación, de actividades y de casos de uso también modelan los aspectos dinámicos de los sistemas. Los diagramas de secuencia y los diagramas de comunicación se discuten en el Capítulo 19; los diagramas de actividades se discuten en el Capítulo 20; los diagramas de casos de uso se discuten en el Capítulo 18.

Los diagramas de estados se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto supone el modelado del comportamiento de objetos reactivos. Un objeto reactivo es aquel para el que la mejor forma de caracterizar su comportamiento es señalar cuál es su respuesta a los eventos lanzados desde fuera de su contexto. Un objeto reactivo tiene un ciclo de vida bien definido, cuyo comportamiento se ve afectado por su pasado. Los diagramas de estados pueden asociarse a las clases, los casos de uso, o a sistemas completos para visualizar, especificar, construir y documentar la dinámica de un objeto individual.

Los diagramas de estados no sólo son importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables a través de ingeniería directa e inversa.

Introducción

La diferencia entre construir una caseta para un perro y construir un rascacielos se discute en el Capítulo 1.

Considérese al inversor que financia la construcción de un rascacielos. Es poco probable que esté interesado en los detalles del proceso de construcción. La selección de materiales, la planificación de los trabajos y las reuniones sobre detalles de ingeniería son actividades importantes para el constructor, pero no son tan importantes para la persona que financia el proyecto.

El inversor está interesado en obtener unos importantes beneficios de la inversión, y esto significa proteger la inversión frente al riesgo. Un inversor realmente confiado entregará al constructor una gran cantidad de dinero, se marchará durante un tiempo, y regresará sólo cuando el constructor esté en condiciones de entregarle las llaves del edificio. Un inversor como éste está interesado únicamente en el estado final del edificio.

Un inversor más práctico también confiará en el constructor, pero al mismo tiempo se preocupará de verificar que el proyecto está en marcha antes de entregar el dinero. Así, en vez de darle al constructor una gran cantidad de dinero y despreocuparse, el inversor prudente establecerá unos hitos claros en el proyecto, cada uno de los cuales irá asociado a la terminación de ciertas actividades, y tras los cuales irá entregando dinero al constructor para la siguiente fase del proyecto. Por ejemplo, al comenzar el proyecto se entregaría una modesta cantidad de fondos, para financiar el trabajo de arquitectura. Después de haber sido aprobada la visión arquitectónica, podrían aportarse más fondos para pagar los trabajos de ingeniería. Tras completarse este trabajo de manera satisfactoria, puede entregarse una gran cantidad de dinero para que el constructor proceda a remover tierras.

Los diagramas de Gantt y los diagramas Pert se discuten en el Capítulo 20.

Los diagramas de actividades utilizados como diagramas de flujo se discuten en el Capítulo 20; las máquinas de estados se discuten en el Capítulo 22.

A lo largo del camino, desde el movimiento de tierras hasta la emisión del certificado de habitabilidad, habrá otros hitos. Cada uno de estos hitos representa un estado estable del proyecto: arquitectura terminada, ingeniería hecha, tierras removidas, infraestructura terminada, edificio cerrado, etcétera. Para el inversor, es más importante seguir el cambio de estado del edificio que seguir el flujo de actividades, que es lo que el constructor podría estar haciendo utilizando diagramas Pert para modelar el flujo de trabajo del proyecto.

Cuando se modelan sistemas con gran cantidad de software, la forma más natural de visualizar, especificar, construir y documentar el comportamiento de ciertos tipos de objetos es centrarse en el flujo de control entre estados en vez del flujo de actividades. Esto último se hace utilizando un diagrama de flujo (y en UML, con un diagrama de actividades). Imagínese, por un momento, el modelado del comportamiento de un sistema de seguridad de una vivienda. Un

sistema de esa naturaleza funciona de forma continua, reaccionando a ciertos eventos externos, tales como la rotura de una ventana. Además, el orden de los eventos cambia la forma en que se comporta el sistema. Por ejemplo, la detección de la rotura de una ventana sólo disparará una alarma si el sistema antes está activado. La mejor forma de especificar el comportamiento de un sistema de estas características es modelar sus estados estables (por ejemplo, Inactivo, Montado, Activo, Comprobando, etcétera), los eventos que producen un cambio de estado y las acciones que ocurren en cada cambio de estado.

En UML, el comportamiento dirigido por eventos de un objeto se modela utilizando diagramas de estados. Como se muestra en la Figura 25.1, un diagrama de estados es simplemente la representación de una máquina de estados, que destaca el flujo de control entre estados.

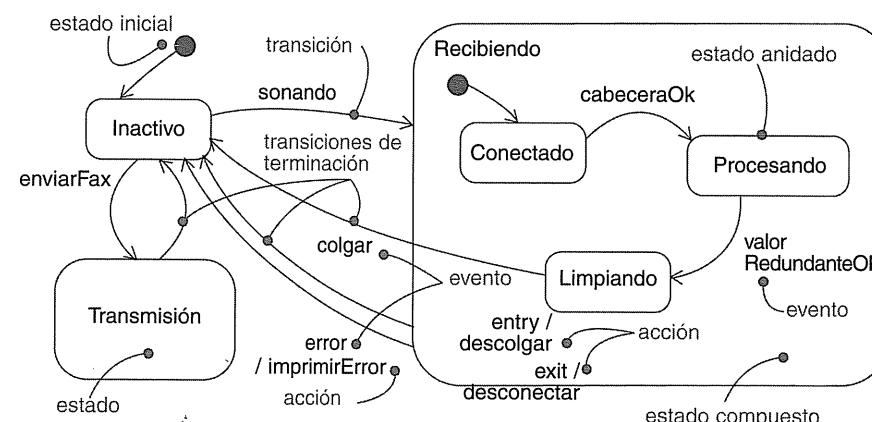


Figura 25.1: Diagrama de estados.

Términos y conceptos

Un *diagrama de estados* muestra una máquina de estados, destacando el flujo de control entre estados. Una *máquina de estados* es un comportamiento que especifica las secuencias de estados por los que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a esos eventos. Un *estado* es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un *evento* es la especificación de un acontecimiento significativo que ocupa un lugar en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede activar una transición de estado. Una *transición*

es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones especificadas. Una *actividad* es una ejecución en curso, dentro de una máquina de estados. Una *acción* es una computación primitiva ejecutable que produce un cambio en el estado del modelo o la devolución de un valor. Gráficamente, un diagrama de estados es una colección de nodos y arcos.

Nota: Los diagramas de estados de UML se basan en la notación de los *statecharts* inventados por David Harel. En particular, Harel desarrolló los conceptos de estados anidados y estados ortogonales en un sistema formal preciso. Los conceptos de UML son algo menos formales que la notación de Harel, y difieren en algunos detalles; en particular, están dirigidos a los sistemas orientados a objetos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de estados es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (es decir, un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de estados de los otros tipos de diagramas es su contenido particular.

Contenido

Normalmente, los diagramas de estados contienen:

- Estados simples y compuestos.
- Transiciones, incluyendo eventos y acciones.

Al igual que los otros diagramas, los diagramas de estados pueden contener notas y restricciones.

Los estados simples y compuestos, las transiciones, los eventos y las acciones se discuten en el Capítulo 22; los diagramas de actividades se discuten en el Capítulo 20; las notas y restricciones se discuten en el Capítulo 6.

Nota: Un diagrama de estados es básicamente una proyección de los elementos de una máquina de estados. Esto significa que los diagramas de estados pueden contener bifurcaciones, divisiones, uniones, estados de acción, estados de actividad, objetos, estados iniciales y finales, estados de historia, etcétera. En realidad, un diagrama de estados puede contener todas y cada una de las características de una máquina de estados.

Usos comunes

Las cinco vistas de una arquitectura se discuten en el Capítulo 2; las instancias se discuten en el Capítulo 13; las clases se discuten en los Capítulos 4 y 9.

Las clases activas se discuten en el Capítulo 23; las interfaces se discuten en el Capítulo 11; los componentes se discuten en el Capítulo 15; los nodos se discuten en el Capítulo 27; los casos de uso se discuten en el Capítulo 17; los sistemas se discuten en el Capítulo 32.

Los diagramas de estados se utilizan para modelar los aspectos dinámicos de un sistema. Estos aspectos dinámicos pueden involucrar el comportamiento dirigido por eventos de cualquier tipo de objeto en cualquier vista de la arquitectura de un sistema, incluyendo las clases (que incluyen a las clases activas), interfaces, componentes y nodos.

Cuando se modela algún aspecto dinámico de un sistema con un diagrama de estados, se puede hacer en el contexto de casi cualquier elemento de modelado. Sin embargo, lo habitual es utilizar los diagramas de estados en el contexto del sistema global, de un subsistema o de una clase. También se pueden asociar diagramas de estados a los casos de uso (para modelar un escenario).

En el modelado de los aspectos dinámicos de un sistema, una clase o un caso de uso, los diagramas de objetos se utilizan normalmente para modelar objetos reactivos.

Un objeto reactivo (o dirigido por eventos) es aquel para el que la mejor forma de caracterizar su comportamiento es señalar cuál es su respuesta a los eventos lanzados desde fuera de su contexto. Normalmente, un objeto reactivo está ocioso hasta que recibe un evento. Cuando recibe un evento, lo habitual es que su respuesta dependa de los eventos anteriores. Después de que el objeto responde a un evento, de nuevo vuelve a estar ocioso, esperando al siguiente evento. Con este tipo de objetos el interés radica en los estados estables, los eventos que disparan una transición de un estado a otro y las acciones que ocurren en cada cambio de estado.

Nota: En cambio, los diagramas de actividades se utilizan para modelar el flujo de trabajo o una operación. Los diagramas de actividades se adaptan mejor al modelado del flujo de actividades a lo largo del tiempo, como cuando se representa un diagrama de flujo.

Técnicas comunes de modelado

Modelado de objetos reactivos

Las interacciones se discuten en el Capítulo 16; los diagramas de actividades se discuten en el Capítulo 20.

El modelado de la vida de un objeto se discute en el Capítulo 22.

El tiempo y el espacio se discuten en el Capítulo 24.

La mayoría de las veces, los diagramas de estados se utilizan para modelar el comportamiento de objetos reactivos, especialmente instancias de clases, casos de uso y el sistema global. Así como las interacciones modelan el comportamiento de una sociedad de objetos que colaboran entre sí, un diagrama de estados modela el comportamiento de un único objeto a lo largo de su vida. Así como un diagrama de actividades modela el flujo de control entre actividades, un diagrama de estados modela el flujo de control entre estados.

Cuando se modela el comportamiento de un objeto reactivo, normalmente se especifican tres cosas: los estados estables en los que puede encontrarse el objeto, los eventos que disparan una transición entre estados y las acciones que tienen lugar durante cada cambio de estado. El modelado del comportamiento de un objeto reactivo implica también modelar la vida del objeto, comenzando en la creación del objeto y continuando hasta su destrucción, resaltando los estados estables en los que puede encontrarse el objeto.

Un estado estable representa una condición que puede satisfacer un objeto durante un período de tiempo identificable. Al ocurrir un evento, el objeto puede pasar de un estado a otro. Estos eventos también pueden disparar autotransiciones y transiciones internas, en las cuales el estado origen y el estado destino de la transición son el mismo. Como reacción a un evento o a un cambio de estado, el objeto puede responder ejecutando una acción.

Nota: Cuando se modela el comportamiento de un objeto reactivo, su acción se puede especificar ligándola a una transición o a un cambio de estado. En términos técnicos, una máquina de estado cuyas acciones están todas asociadas a transiciones se llama máquina de Mealy; una máquina de estados cuyas acciones están todas ligadas a estados se llama máquina de Moore. Matemáticamente, ambas formas tienen una potencia expresiva equivalente. En la práctica, normalmente se desarrollan diagramas de estados basados en una combinación de máquinas de Mealy y de Moore.

Las precondiciones y las postcondiciones se discuten en el Capítulo 10; las interfaces se discuten en el Capítulo 11.

Para modelar un objeto reactivo:

- Hay que elegir el contexto para la máquina de estados, ya sea una clase, un caso de uso o el sistema global.
- Hay que elegir los estados inicial y final del objeto. Para que sirva de guía al resto del modelo, deben enunciarse si es posible las pre y postcondiciones de los estados inicial y final, respectivamente.
- Hay que elegir los estados estables del objeto considerando las condiciones que puede satisfacer el objeto durante algún período identificable de tiempo. Hay que comenzar con los estados de más alto nivel y después considerar sus posibles subestados.
- Hay que elegir un orden parcial significativo de los estados estables a lo largo de la vida del objeto.
- Hay que elegir los eventos que pueden disparar una transición de un estado a otro. Hay que modelar esos eventos como disparadores de las transiciones que hacen pasar de una ordenación legal de estados a otra.
- Hay que asociar acciones a estas transiciones (como en una máquina de Mealy) y/o a los estados (como en una máquina de Moore).
- Hay que considerar diferentes formas de simplificar la máquina con subestados, bifurcaciones, divisiones, uniones y estados de historia.
- Hay que comprobar que todos los estados son alcanzables mediante alguna combinación de eventos.
- Hay que comprobar que ningún estado es un punto muerto del cual no se puede salir con ninguna combinación de eventos.
- Hay que hacer trazas a través de la máquina de estados, ya sea manualmente o con herramientas, para verificar las secuencias esperadas de eventos y sus respuestas.

Por ejemplo, la Figura 25.2 muestra el diagrama de estados para analizar un lenguaje libre de contexto muy sencillo, como el que puede haber en un sistema que genera o interpreta mensajes en XML. En este caso, la máquina se ha diseñado para analizar una secuencia de caracteres que coincide con la siguiente sintaxis:

mensaje : '<' cadena '>' cadena ';'

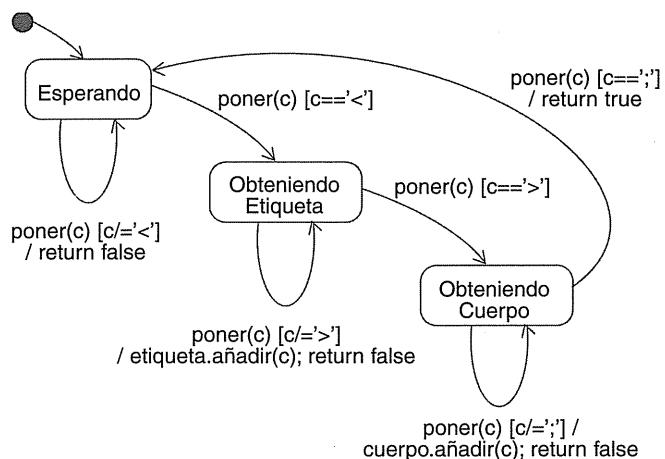


Figura 25.2: Modelado de objetos reactivos.

La primera cadena de caracteres representa una etiqueta; la segunda representa el cuerpo del mensaje. Dada una secuencia de caracteres, sólo se pueden aceptar los mensajes bien formados que sigan esta sintaxis.

Los eventos se discuten en el Capítulo 21.

Como se muestra en la figura, sólo hay tres estados estables en esta máquina de estados: Esperando, ObteniendoEtiqueta y ObteniendoCuerpo. Esta máquina de estados se ha diseñado como una máquina de Mealy, con las acciones asociadas a las transiciones. De hecho, sólo hay un evento de interés en esta máquina de estados, la invocación de poner con el parámetro actual c (un carácter). Mientras está Esperando, esta máquina rechaza cualquier carácter que no suponga el comienzo de una etiqueta (como se especifica en la condición de guarda). Cuando se recibe el carácter de inicio de una palabra, el estado del objeto cambia a ObteniendoEtiqueta. Mientras está en este estado, la máquina guarda cualquier carácter que no suponga el final de la etiqueta (como se especifica en la condición de guarda). Cuando se recibe el final de la etiqueta, el estado del objeto cambia a ObteniendoCuerpo. Mientras está en este estado, la máquina guarda cualquier carácter que no suponga el final del cuerpo de un mensaje (como se especifica en la condición de guarda). Cuando se recibe el final del mensaje, el estado del objeto cambia a Esperando, y se devuelve un valor indicando que el mensaje ha sido analizado (y la máquina está preparada para recibir otro mensaje).

Nótese que este diagrama especifica una máquina que funciona continuamente; no hay estado final.

Ingeniería directa e inversa

La *ingeniería directa* (creación de código a partir de un modelo) es posible con los diagramas de estados, especialmente si el contexto del diagrama es una clase. Por ejemplo, con el anterior diagrama de estados, una herramienta de ingeniería directa podría generar el siguiente código Java para la clase AnalizadorDeMensajes.

```

class AnalizadorDeMensajes {
    public
        boolean poner(char c) {
            switch (estado) {
                case Esperando:
                    if (c == '<') {
                        estado = ObteniendoEtiqueta;
                        etiqueta = new StringBuffer();
                        cuerpo = new StringBuffer();
                    }
                    break;
                case ObteniendoEtiqueta:
                    if (c == '>')
                        estado = ObteniendoCuerpo;
                    else
                        etiqueta.append(c);
                    break;
                case ObteniendoCuerpo:
                    if (c == ';')
                        estado = Esperando;
                    else
                        cuerpo.append(c);
                    return true;
            }
            return false;
        }
        StringBuffer obtenerEtiqueta() {
            return etiqueta;
        }
        StringBuffer obtenerCuerpo() {
            return cuerpo;
        }
    private

```

```

final static int Esperando = 0;
final static int ObteniendoEtiqueta = 1;
final static int ObteniendoCuerpo = 2;
int estado = Esperando;
StringBuffer etiqueta, cuerpo;
}

```

Esto requiere algo de ingenio. La herramienta de ingeniería directa debe generar los atributos privados necesarios y las constantes estáticas finales.

La *ingeniería inversa* (creación de un modelo a partir del código) es teóricamente posible, pero no es muy útil en la práctica. La elección de lo que constituye un estado significativo depende del diseñador. Las herramientas de ingeniería inversa no tienen capacidad de abstracción y, por lo tanto, no pueden producir diagramas de estados significativos. Algo más interesante que la ingeniería inversa de un modelo a partir de código es la animación de un modelo durante la ejecución del sistema desarrollado. Por ejemplo, dado el diagrama anterior, una herramienta podría animar los estados del diagrama mientras están siendo alcanzados en el sistema en ejecución. Análogamente, el disparo de las transiciones también podría ser animado, mostrando la recepción de eventos y la ejecución de acciones resultante. Bajo el control de un depurador, se podría controlar la velocidad de ejecución, colocando puntos de ruptura para detener la acción en los estados interesantes para examinar los valores de los atributos de los objetos individuales.

Sugerencias y consejos

Cuando se crean diagramas de estados en UML, hay que recordar que cada diagrama de estados es una proyección sobre el mismo modelo de los aspectos dinámicos de un sistema. Un único diagrama de estados puede capturar la semántica de un único objeto reactivo, pero un único diagrama de estados no puede capturar la semántica de un sistema completo que no sea trivial.

Un diagrama de estados bien estructurado:

- Se ocupa de modelar un aspecto de la dinámica de un sistema.
- Sólo contiene aquellos elementos esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con su nivel de abstracción; muestra sólo aquellas características esenciales para su comprensión.

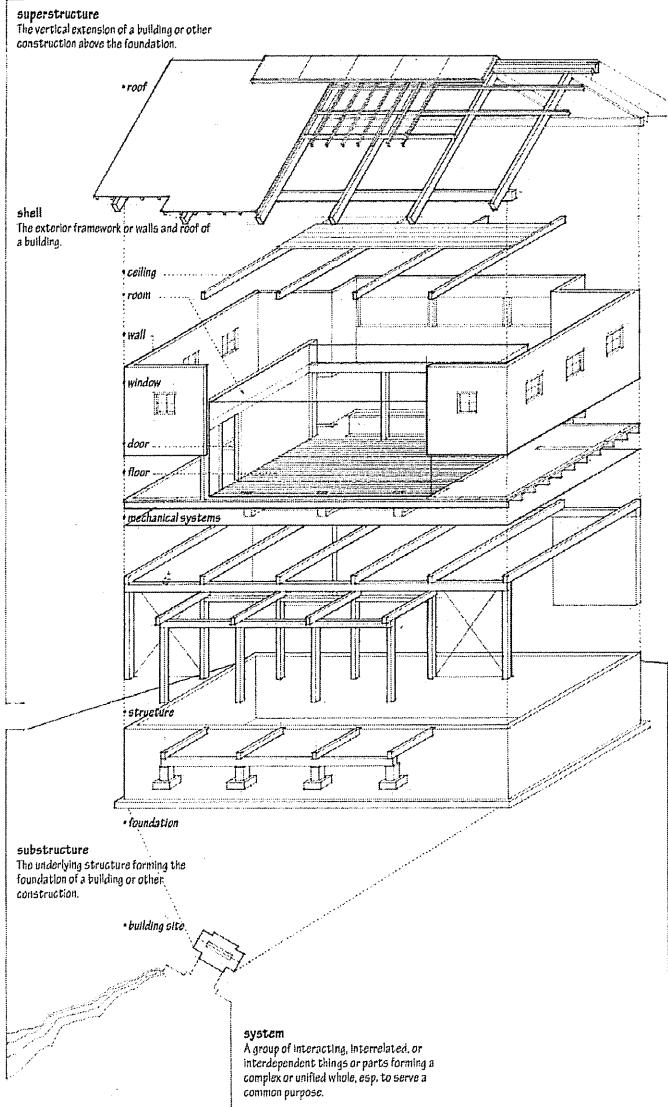
- Utiliza de forma equilibrada máquinas de Mealy y de Moore.

Cuando se dibuje un diagrama de estados:

- Hay que darle un nombre que comunique su propósito.
- Hay que comenzar modelando los estados estables del objeto, y a continuación modelar las transiciones legales entre estados. Deben considerarse las bifurcaciones, la concurrencia y el flujo de objetos como algo secundario, quizás en diagramas separados.
- Hay que organizar los elementos para que se minimicen los cruces de líneas.
- Para los diagramas de estados grandes, hay que considerar las características avanzadas como las submáquinas incluidas en la especificación completa de UML.

LENGUAJE
UNIFICADO DE
MODELADO

Parte 6
**MODELADO
ARQUITECTÓNICO**



En este capítulo

- Artefactos, clases y manifestación.
- Modelado de ejecutables y bibliotecas.
- Modelado de tablas, archivos y documentos.
- Modelado de código fuente.

Los artefactos pertenecen al mundo material de los bits y, por lo tanto, son un bloque de construcción importante cuando se modelan los aspectos físicos de un sistema. Un artefacto es una parte física y reemplazable de un sistema.

Los artefactos se utilizan para modelar los elementos físicos que pueden hallarse en un nodo, tales como ejecutables, bibliotecas, tablas, archivos y documentos. Normalmente, un artefacto representa el empaquetamiento físico de elementos que por su parte son lógicos, tales como clases, interfaces y colaboraciones.

Introducción

El producto final de una constructora es un edificio tangible existente en el mundo real. Los modelos lógicos se construyen para visualizar, especificar y documentar las decisiones sobre la construcción; la localización de las paredes, puertas y ventanas; la distribución de los sistemas eléctrico y de fontanería; y el estilo arquitectónico global. A la hora de construir el edificio, esas paredes, puertas, ventanas y demás elementos conceptuales se convierten en cosas reales, físicas.

Las diferencias entre construir una caseta para un perro y construir un rascacielos se discuten en el Capítulo 1.

Tanto la vista lógica como la física son necesarias. Si se va a construir un edificio desechar para el que el coste de destruir y reconstruir es prácticamente cero (por ejemplo, si se construye una caseta para un perro), probablemente se pueda abordar la construcción física sin hacer ningún modelado lógico. Si, por otro lado, se está construyendo algo duradero, y para lo cual el coste de cambiar o fallar es alto, entonces lo más práctico para gestionar el riesgo es crear tanto los modelos lógicos como los físicos.

Lo mismo ocurre cuando se construye un sistema con gran cantidad de software. El modelado lógico se hace para visualizar, especificar y documentar las decisiones acerca del vocabulario del dominio y sobre cómo colaboran estos elementos tanto estructuralmente como desde el punto de vista del comportamiento. El modelado físico se hace para construir el sistema ejecutable. Mientras que los elementos lógicos pertenecen al mundo conceptual, los elementos físicos pertenecen al mundo de los bits (o sea, en última instancia se encuentran en nodos físicos y pueden ser ejecutados directamente o formar parte, de alguna forma indirecta, de un sistema ejecutable).

En UML, todos estos elementos físicos se modelan como artefactos. Un artefacto es un elemento físico al nivel de la plataforma de implementación.

En el terreno del software, muchos sistemas operativos y lenguajes de programación soportan directamente el concepto de artefacto. Las bibliotecas de código objeto, los ejecutables, los componentes .NET y los Enterprise Java Beans son todos ejemplos de artefactos que se pueden representar directamente en UML. No sólo se pueden utilizar los artefactos para modelar estos tipos de elementos, sino que también pueden utilizarse para representar otros elementos que participan en un sistema en ejecución, tales como tablas, archivos y documentos.

Los estereotipos se discuten en el Capítulo 6.

UML proporciona una representación gráfica de un artefacto, como se muestra en la Figura 26.1. Esta notación canónica permite visualizar un artefacto de forma independiente de cualquier sistema operativo o lenguaje de programación. Con los estereotipos, uno de los mecanismos de extensibilidad de UML, se puede particularizar esta notación para representar tipos específicos de artefactos.

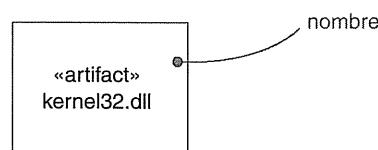


Figura 26.1: Artefactos.

Términos y conceptos

Un *artefacto* es una parte física y reemplazable de un sistema que existe a nivel de la plataforma de implementación. Gráficamente, un artefacto se representa como un rectángulo con la palabra clave «*artifact*».

Nombres

El nombre de un artefacto debe ser único dentro del nodo que lo contiene.

Cada artefacto debe tener un nombre que lo distinga del resto de los artefactos. Un *nombre* es una cadena de texto. Ese nombre solo se denomina *nombre simple*; un *nombre calificado* consta del nombre del artefacto precedido del nombre del paquete en el que se encuentra. Normalmente, un artefacto se dibuja mostrando sólo su nombre, como se ilustra en la Figura 26.2. Al igual que las clases, los artefactos se pueden adornar con valores etiquetados o con compartimentos adicionales que muestran sus detalles, como se ve en la figura.

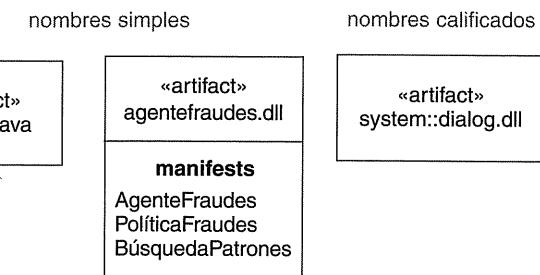


Figura 26.2: Nombres de artefactos simples y calificados.

Nota: El nombre de un artefacto puede ser texto con cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se utilizan para separar el nombre de un artefacto y el del paquete que lo contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de artefactos son nombres cortos o expresiones nominales, extraídos del vocabulario de la implementación y, de acuerdo con el sistema operativo destino, pueden incluir extensiones (tales como .java y .dll).

Artefactos y clases

Las clases se discuten en los Capítulos 4 y 9; las interacciones se discuten en el Capítulo 16.

Las clases y los artefactos son clasificadores. Sin embargo, hay algunas diferencias significativas entre los artefactos y las clases.

- Las clases representan abstracciones lógicas; los artefactos representan elementos físicos del mundo de los bits. Para decirlo brevemente, los artefactos pueden estar en nodos; las clases no.
- Los artefactos representan el empaquetamiento físico de bits sobre la plataforma de implementación.
- Las clases pueden tener atributos y operaciones. Los artefactos pueden implementar clases y métodos, pero no tienen atributos u operaciones por ellos mismos.

Los nodos se discuten en el Capítulo 27.

La primera diferencia es la más importante. Durante el modelado de un sistema, la decisión de si se debería utilizar una clase o un artefacto es sencilla. Si el elemento que se va a modelar reside directamente en un nodo, debe usarse un artefacto; en otro caso, debe usarse una clase. La segunda diferencia también clarifica esto.

La tercera diferencia sugiere una relación entre las clases y los artefactos. En particular, un artefacto es la implementación física de un conjunto de otros elementos lógicos, tales como clases y colaboraciones. Como se muestra en la Figura 26.3, la relación entre un artefacto y las clases que implementa puede representarse explícitamente mediante una relación de manifestación.

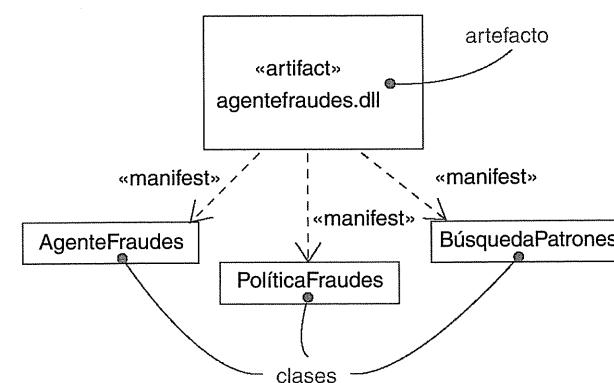


Figura 26.3: Artefactos y clases.

Tipos de artefactos

Se pueden distinguir tres tipos de artefactos.

El primer tipo son los *artefactos de despliegue*. Éstos son los artefactos necesarios y suficientes para formar un sistema ejecutable, tales como las bibliotecas dinámicas (DLL, *Dinamic Link Libraries*) y los ejecutables (EXE). Esta definición de UML de artefacto es lo bastante amplia para cubrir modelos de objetos clásicos, tales como .NET, CORBA y Enterprise Java Beans, así como modelos de objetos alternativos, que quizás impliquen páginas web dinámicas, tablas de bases de datos y ejecutables que utilicen mecanismos de comunicación propietarios.

El segundo tipo son los *artefactos producto del trabajo*. Estos artefactos son básicamente productos que quedan al final del proceso de desarrollo, y consisten en cosas tales como archivos de código fuente y archivos de datos a partir de los cuales se crean los artefactos de despliegue. Estos artefactos no participan directamente en un sistema ejecutable, pero son los productos del trabajo de desarrollo que se utilizan para crear el sistema ejecutable.

El tercer tipo son los *artefactos de ejecución*. Estos artefactos se crean como consecuencia de un sistema en ejecución, tales como un objeto .NET, el cual se instancia a partir de una DLL.

Elementos estándar

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Todos los mecanismos de extensibilidad de UML se aplican a los artefactos. La mayoría de las veces se utilizarán los valores etiquetados para extender las propiedades de un artefacto (por ejemplo, para especificar la versión de un artefacto de desarrollo) y los estereotipos para especificar nuevos tipos de artefactos (tales como artefactos específicos de un sistema operativo).

UML define varios estereotipos estándar que se aplican a los artefactos:

1. executable Especifica un artefacto que se puede ejecutar en un nodo.
2. library Especifica una biblioteca de objetos estática o dinámica.
3. file Especifica un artefacto que representa un documento que contiene código fuente o datos.
4. document Especifica un artefacto que representa un documento.

Se pueden definir otros estereotipos para plataformas y sistemas específicos.

Técnicas comunes de modelado

Modelado de ejecutables y bibliotecas

La mayoría de las veces, los artefactos se utilizan para modelar los artefactos de despliegue que configuran una implementación. Si se está desplegando un sistema trivial cuya implementación consta de un único archivo ejecutable, no es necesario hacer ningún modelado de artefactos. Si, por otro lado, el sistema que se está desplegando se compone de varios ejecutables y bibliotecas de objetos asociadas, hacer el modelado de artefactos ayudará a visualizar, especificar, construir y documentar las decisiones que se han tomado sobre el sistema físico. El modelado de artefactos es aún más importante, si se quieren controlar las versiones y la gestión de configuraciones de esas partes conforme evoluciona el sistema.

Estas decisiones también se ven afectadas por la topología del sistema final, como se discute en el Capítulo 27.

En la mayoría de los sistemas, estos artefactos de despliegue se obtienen a partir de las decisiones que se toman acerca de cómo dividir la implementación física del sistema. Estas decisiones se verán afectadas por varias cuestiones técnicas (tales como la elección de servicios del sistema operativo basados en artefactos), cuestiones de gestión de configuraciones (como las decisiones acerca de qué partes pueden cambiar a lo largo del tiempo) y cuestiones de reutilización (es decir, la decisión de qué artefactos se pueden reutilizar en o desde otros sistemas).

Para modelar ejecutables y bibliotecas:

- Hay que identificar la partición del sistema físico. Se debe considerar el impacto de las cuestiones técnicas, de gestión de configuraciones y de reutilización.
- Hay que modelar como artefactos cualquier ejecutable y biblioteca, utilizando los elementos estándar apropiados. Si la implementación introduce nuevos tipos de artefactos, hay que introducir un estereotipo nuevo apropiado.
- Si es importante manejar las líneas de separación del sistema, hay que modelar las interfaces más importantes que algunos artefactos utilizan y otros realizan.
- Si es necesario para comunicar el objetivo, hay que modelar las relaciones entre esos ejecutables, bibliotecas e interfaces. Lo más frecuente será modelar las dependencias entre esas partes para visualizar el impacto del cambio.

Por ejemplo, la Figura 26.4 muestra un conjunto de artefactos extraídos de una herramienta de productividad personal que se ejecuta en un único computador

personal. Esta figura incluye un ejecutable (`animator.exe`) y cuatro bibliotecas (`dlog.dll`, `wrfrme.dll`, `render.dll` y `raytrce.dll`), los cuales utilizan los elementos estándar de UML para ejecutables y bibliotecas, respectivamente. Este diagrama también muestra las dependencias entre esos artefactos.

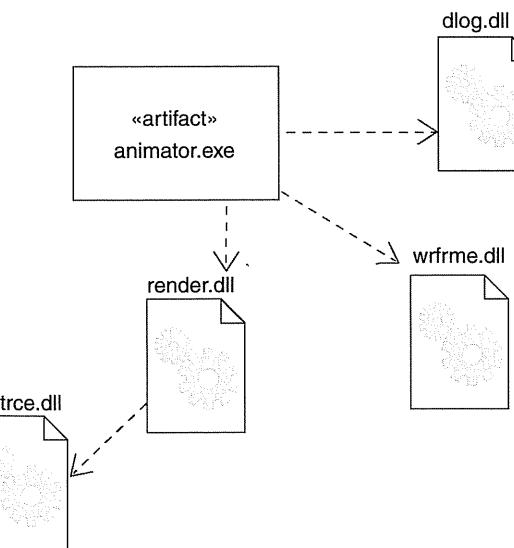


Figura 26.4: Modelado de ejecutables y bibliotecas

Los paquetes se discuten en el Capítulo 12.

El modelado del despliegue se discute en el Capítulo 27.

Conforme se van haciendo más grandes los modelos, se descubre que los artefactos tienden a agruparse en grupos relacionados conceptual y semánticamente. En UML, se pueden utilizar los paquetes para modelar estos grupos de artefactos.

Para los grandes sistemas que se despliegan a través de varios computadores, será conveniente modelar cómo se distribuyen los artefactos, indicando los nodos sobre los que se ubican.

Modelado de tablas, archivos y documentos

El modelado de los ejecutables y las bibliotecas que configuran la implementación física del sistema es útil, pero a menudo resulta que hay varios artefactos de despliegue auxiliares que no son ejecutables ni bibliotecas, pero son críticos para el despliegue físico del sistema. Por ejemplo, una implementación puede incluir archivos de datos, documentos de ayuda, guiones (*scripts*), archivos de diarios (*logs*), archivos de inicialización y archivos de instalación/desinstalación. El modelado de estos artefactos es importante para controlar la configuración del

sistema. Afortunadamente, se pueden utilizar los artefactos de UML para modelar todos estos artefactos.

Para modelar tablas, archivos y documentos:

- Hay que identificar los artefactos auxiliares que forman parte de la implementación física del sistema.
- Hay que modelar estas cosas como artefactos. Si la implementación introduce nuevos tipos de artefactos, hay que introducir un estereotipo nuevo apropiado.
- Si es necesario para comunicar el objetivo, hay que modelar las relaciones entre esos artefactos auxiliares y los demás ejecutables, bibliotecas e interfaces del sistema. Lo más frecuente será modelar las dependencias entre esas partes para visualizar el impacto del cambio.

El modelado de bases de datos lógicas y físicas se discute en los Capítulos 8 y 30, respectivamente.

Por ejemplo, la Figura 26.5 se basa en la figura anterior y muestra las tablas, archivos y documentos que forman parte del sistema desplegado alrededor del ejecutable `animator.exe`. Esta figura incluye un documento (`animator.hlp`), un

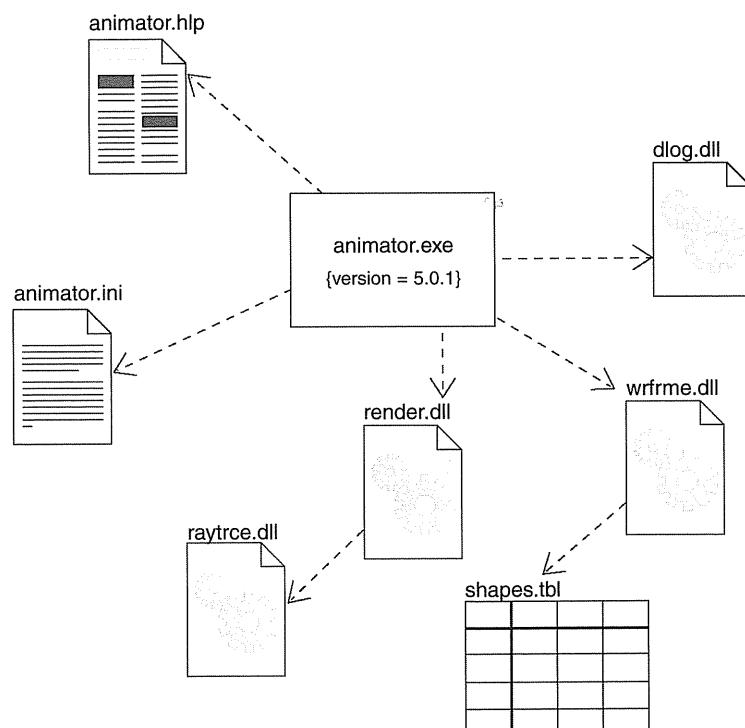


Figura 26.5: Modelado de tablas, archivos y documentos.

archivo simple (`animator.ini`) y una tabla de base de datos (`shapes.tbl`). Este ejemplo ilustra algunos estereotipos e iconos definidos por el usuario para los artefactos.

El modelado de bases de datos puede complicarse cuando se empieza a tratar con muchas tablas, disparadores (*triggers*) y procedimientos almacenados. Para visualizar, especificar, construir y documentar estas características, será necesario modelar el esquema lógico, así como las bases de datos físicas.

Modelado de código fuente

La mayoría de las veces, los artefactos se utilizan para modelar las partes físicas que configuran una implementación. El segundo uso más frecuente de los artefactos es el modelado de la configuración de todos los archivos de código fuente que las herramientas de desarrollo utilizan para crear esos artefactos. Éstos representan los artefactos producto del trabajo del proceso de desarrollo.

Modelar gráficamente el código fuente es particularmente útil para visualizar las dependencias de compilación entre los archivos de código fuente y para gestionar la división y combinación de grupos de estos archivos cuando los caminos del desarrollo se bifurcan y se unen. De esta forma, los artefactos de UML pueden ser la interfaz gráfica para las herramientas de gestión de configuraciones y de control de versiones.

En la mayoría de los sistemas, los archivos de código fuente se obtienen a partir de las decisiones tomadas sobre cómo dividir los archivos necesarios en el entorno de desarrollo. Estos archivos se utilizan para almacenar los detalles de las clases, interfaces, colaboraciones y otros elementos lógicos, como un paso intermedio para crear los artefactos binarios físicos que se derivan de estos elementos utilizando las herramientas. La mayoría de las veces, estas herramientas impondrán un estilo de organización (uno o dos archivos por clase es algo frecuente), pero aun así será conveniente visualizar las relaciones entre esos archivos. Las decisiones sobre cómo manejar el cambio determinarán cómo se organizan esos archivos en grupos utilizando paquetes, y cómo se realiza la gestión de versiones.

Para modelar código fuente:

- Hay que modelar los archivos utilizados para almacenar los detalles de todos los elementos lógicos, junto con sus dependencias de compilación, según las restricciones impuestas por las herramientas de desarrollo.

- Si es importante asociar estos modelos a las herramientas de gestión de configuraciones y de control de versiones, hay que incluir valores etiquetados, tales como la versión, el autor y la información sobre el registro y verificación, para cada archivo que esté bajo el control de la gestión de configuraciones.
- En la medida de lo posible, hay que permitir que las herramientas de desarrollo manejen las relaciones entre esos archivos, y utilizar UML sólo para visualizar y documentar estas relaciones.

Por ejemplo, la Figura 26.6 muestra algunos archivos de código fuente utilizados para construir la biblioteca `render.dll` a partir de los ejemplos anteriores. Esta figura incluye cuatro archivos de cabecera (`render.h`, `rengine.h`, `poly.h` y `colortab.h`) que representan el código fuente de la especificación de ciertas clases. También hay un archivo de implementación (`render.cpp`) que representa la implementación de una de estas cabeceras.

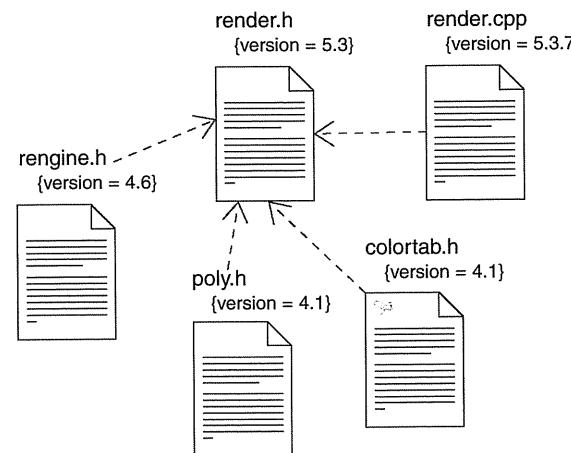


Figura 26.6: Modelado de código fuente.

Los paquetes se discuten en el Capítulo 12; las relaciones trace, un tipo de dependencia, se discute en los Capítulos 5 y 10.

Conforme crecen los modelos, se irá descubriendo que muchos archivos de código fuente tienden a asociarse en grupos relacionados conceptual y semánticamente. La mayoría de las veces, las herramientas de desarrollo colocarán estos grupos en directorios separados. En UML, se pueden utilizar los paquetes para modelar estas agrupaciones de archivos de código fuente.

En UML es posible visualizar la relación de una clase con su archivo de código fuente y, a su vez, la relación de un archivo de código fuente con su ejecutable o biblioteca por medio de las relaciones de traza. Sin embargo, pocas veces se necesitará llegar hasta este nivel de detalle.

Sugerencias y consejos

Cuando se modelan artefactos en UML, debe recordarse que se está modelando en la dimensión física. Un artefacto bien estructurado:

- Implementa directamente un conjunto de clases que colaboran entre sí para llevar a cabo la semántica de esas interfaces con economía e ingenio.
- Está débilmente acoplado en relación con otros artefactos.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 27

DESPLIEGUE

En este capítulo

- Nodos y conexiones.
- Modelado de procesadores y dispositivos.
- Modelado de la distribución de artefactos.
- Ingeniería de sistemas.

Los nodos, al igual que los artefactos, pertenecen al mundo material y son un bloque de construcción importante para modelar los aspectos físicos de un sistema. Un nodo es un elemento físico que existe en tiempo de ejecución y que representa un recurso computacional, que generalmente tiene algo de memoria y, a menudo, capacidad de procesamiento.

Los nodos se utilizan para modelar la topología del hardware sobre el que se ejecuta el sistema.⁴ Un nodo representa normalmente un procesador o un dispositivo sobre el que se pueden desplegar los artefactos. Los buenos nodos representan con claridad el vocabulario del hardware en el dominio de la solución.

Introducción

El modelado de cosas que no son software se discute en el Capítulo 4; las cinco vistas de una arquitectura se discuten en el Capítulo 2.

Los artefactos que se desarrollan o se reutilizan como parte de un sistema con gran cantidad de software deben desplegarse sobre algún hardware para su ejecución. Esto es, en realidad, lo que significa ser un *sistema* con gran cantidad de software: tal sistema incluye tanto software como hardware.

Cuando se diseña un sistema con gran cantidad de software, hay que considerar tanto su dimensión lógica como la física. En la parte lógica aparecen cosas como clases, interfaces, colaboraciones, interacciones y máquinas de estados. En la parte física se encuentran los artefactos (que representan los empaquetamientos

Los estereotipos se discuten en el Capítulo 6.

físicos de esos elementos lógicos) y los nodos (que representan el hardware sobre el que se despliegan y se ejecutan esos artefactos).

UML proporciona una representación gráfica para los nodos, como se muestra en la Figura 27.1. Esta notación canónica permite visualizar un nodo independiente-mente de cualquier hardware específico. Con los estereotipos (uno de los mecanismos de extensibilidad de UML) se puede (y a menudo se hará) particularizar esta notación para representar algunos tipos específicos de procesadores y dispositivos.

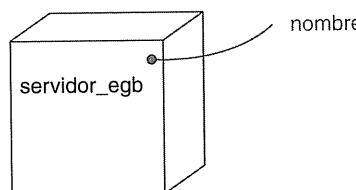


Figura 27.1: Nodos.

Nota: UML está pensado principalmente para modelar sistemas con gran cantidad de software, aunque UML, en combinación con lenguajes textuales de modelado de hardware tales como VHDL, puede ser bastante expresivo para modelar sistemas hardware. UML también es lo suficientemente expresivo para modelar las topologías de sistemas monolíticos, embebidos, cliente/servidor y distribuidos.

Términos y conceptos

Un *nodo* es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, generalmente, tiene alguna memoria y, a menudo, capacidad de procesamiento. Gráficamente, un nodo se representa como un cubo.

Nombres

El nombre de un nodo debe ser único dentro del paquete que lo contiene, como se discute en el Capítulo 12.

Cada nodo debe tener un nombre que lo distinga del resto de los nodos. Un nombre es una cadena de texto. Ese nombre solo se denomina *nombre simple*; un *nombre calificado* consta del nombre del nodo precedido del nombre del paquete en el que se encuentra.

Normalmente, un nodo se dibuja mostrando sólo su nombre, como se ve en la Figura 27.2. Al igual que las clases, los nodos se pueden adornar con valores etiquetados o con compartimentos adicionales para mostrar sus detalles.

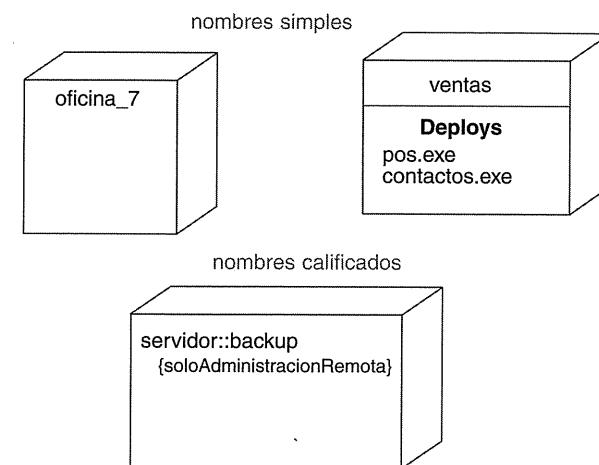


Figura 27.2: Nodos con nombres simples y calificados.

Nota: El nombre de un nodo puede ser texto formado por cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se utilizan para separar el nombre de un nodo y el del paquete que lo contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de nodos son nombres cortos o expresiones nominales extraídos del vocabulario de la implementación.

Nodos y artefactos

Los artefactos se discuten en el Capítulo 26.

En muchos aspectos, los nodos se parecen a los artefactos: ambos tienen nombres; ambos pueden participar en relaciones de dependencia, generalización y asociación; ambos pueden anidarse; ambos pueden tener instancias; ambos pueden participar en interacciones. Sin embargo, hay algunas diferencias significativas entre los nodos y los artefactos:

Los artefactos son los elementos que participan en la ejecución de un sistema; los nodos son los elementos donde se ejecutan los artefactos.

- Los artefactos representan el empaquetamiento físico de los elementos lógicos; los nodos representan el despliegue físico de artefactos.
- La primera diferencia es la más importante. Expresado con sencillez, en los nodos se ejecutan los artefactos; los artefactos son las cosas que se ejecutan en los nodos.

Las relaciones de dependencia se discuten en los Capítulos 5 y 10.

La segunda diferencia sugiere una relación entre clases, artefactos y nodos. En particular, un artefacto es la manifestación de un conjunto de elementos lógicos, tales como clases y colaboraciones, y un nodo es la localización sobre la que se despliegan los artefactos. Una clase puede manifestarse como uno o más artefactos y, a su vez, un artefacto puede desplegarse sobre uno o más nodos. Como se muestra en la Figura 27.3, la relación entre un nodo y el artefacto que despliega puede mostrarse explícitamente a través del anidamiento. La mayoría de las veces, no es necesario visualizar esas relaciones gráficamente, pero se indicarán como una parte de la especificación del nodo, por ejemplo, usando una tabla.

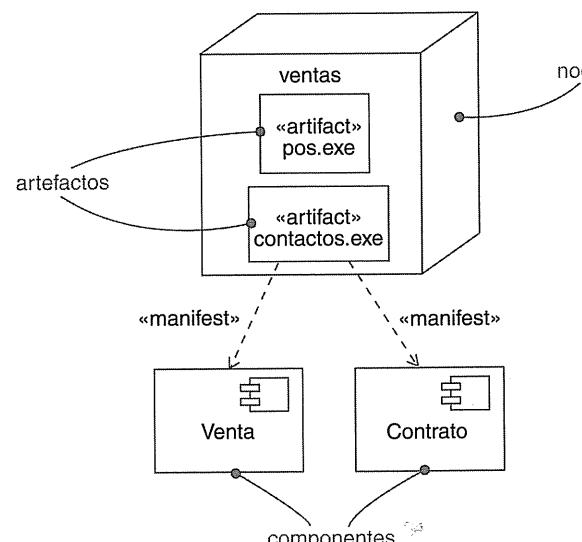


Figura 27.3: Nodos y artefactos.

Un conjunto de objetos o artefactos asignados a un nodo como un grupo se denomina *unidad de distribución*.

Nota: Los nodos también son similares a las clases, por cuanto se pueden especificar atributos y operaciones para ellos. Por ejemplo, se podría especificar que un nodo tiene los atributos `velocidadDelProcesador` y `memoria`, así como las operaciones `encendido`, `apagado` y `suspender`.

Organización de nodos

Los paquetes se discuten en el Capítulo 12.

Los nodos se pueden organizar agrupándolos en paquetes, de la misma forma que se pueden organizar las clases y los artefactos.

Las relaciones se discuten en los Capítulos 5 y 10.

Los nodos también se pueden organizar especificando relaciones de dependencia, generalización y asociación (incluyendo agregación) entre ellos.

Conexiones

El tipo más común de relación entre nodos es la asociación. En este contexto, una asociación representa una conexión física entre nodos, como puede ser una conexión Ethernet, una línea en serie o un bus compartido, como se muestra en la Figura 27.4. Se pueden utilizar asociaciones incluso para modelar conexiones indirectas, tales como un enlace por satélite entre procesadores distintos.

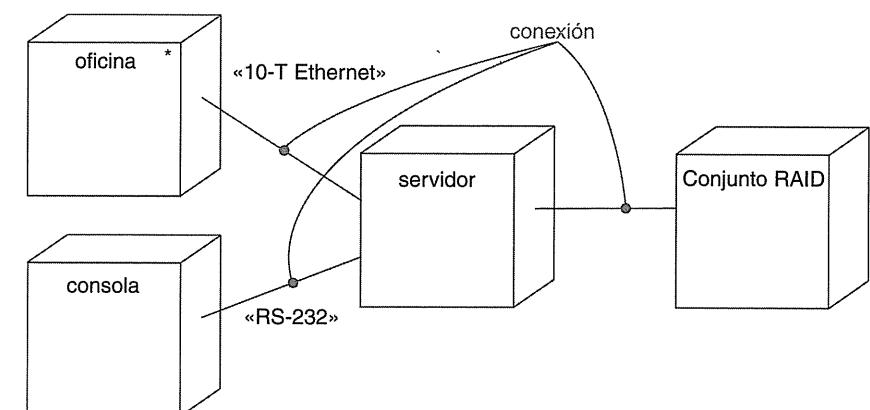


Figura 27.4: Conexiones.

Como los nodos son similares a las clases, se dispone de toda la potencia de las asociaciones. Esto significa que se pueden incluir roles, multiplicidad y restricciones. Como se muestra en la figura anterior, estas asociaciones se deberán estereotipar si se quieren modelar nuevos tipos de conexiones (por ejemplo, distinguir entre una conexión Ethernet 10-T y una conexión en serie RS-232).

Técnicas comunes de modelado

Modelado de procesadores y dispositivos

La mayoría de las veces, los nodos se utilizan para el modelado de los procesadores y los dispositivos que conforman la topología de sistemas monolíticos, embebidos, cliente/servidor o distribuidos.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Puesto que todos los mecanismos de extensibilidad de UML se pueden aplicar a los nodos, a menudo se utilizarán estereotipos para especificar nuevos tipos de nodos que se pueden usar para representar tipos específicos de procesadores y dispositivos. Un *procesador* es un nodo con capacidad de procesamiento, es decir, que puede ejecutar un artefacto. Un *dispositivo* es un nodo sin capacidad de procesamiento (al menos, ninguna modelada a este nivel de abstracción) y, en general, representa algo que interactúa con el mundo real.

Para modelar procesadores y dispositivos:

- Hay que identificar los elementos computacionales de la vista de despliegue del sistema y modelar cada uno como un nodo.
- Si estos elementos representan procesadores y dispositivos genéricos, hay que estereotiparlos como tales. Si son tipos de procesadores y dispositivos que forman parte del vocabulario del dominio, entonces hay que especificar un estereotipo apropiado con un icono para cada uno.
- Al igual que sucede con el modelado de clases, hay que considerar los atributos y operaciones que se pueden aplicar a cada nodo.

Por ejemplo, en la Figura 27.5 se ha estereotipado cada nodo del diagrama anterior. El *servidor* es un nodo estereotipado como un procesador genérico; la *terminal* y la *consola* son nodos estereotipados como tipos especiales de procesadores; y el *conjunto RAID* es un nodo estereotipado como un tipo especial de dispositivo.

Nota: Los nodos son probablemente los bloques de construcción de UML más estereotipados. Cuando, como parte de la ingeniería de sistemas, se modela la vista de despliegue de un sistema con gran cantidad de software, es muy útil proporcionar señales visuales que sean significativas para la audiencia esperada. Si se está modelando un procesador que es un tipo común de computador, se debe representar con un ícono que tenga el aspecto de ese computador. Si se está modelando un dispositivo bastante común, tal como un teléfono móvil, un fax, un módem o una cámara, se debe representar con un ícono que tenga el aspecto de ese dispositivo.

La semántica de la localización se discute en el Capítulo 24.

Las instancias se discuten en el Capítulo 13; los diagramas de objetos se discuten en el Capítulo 14.

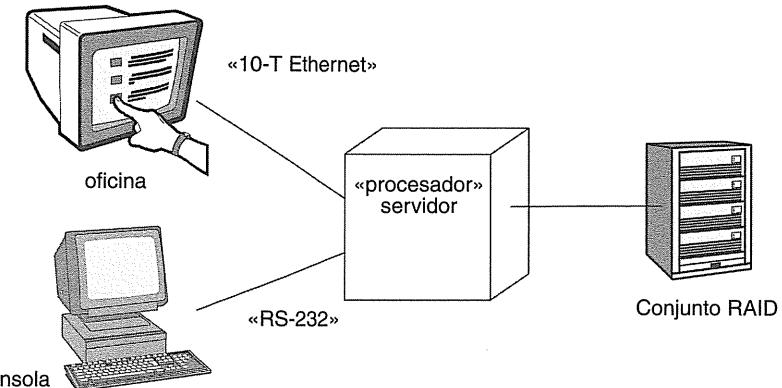


Figura 27.5: Procesadores y dispositivos.

Modelado de la distribución de artefactos

Cuando se modela la topología de un sistema, a menudo es útil visualizar o especificar la distribución física de sus artefactos a través de los procesadores y dispositivos que configuran el sistema.

Para modelar la distribución de artefactos:

- Hay que ubicar cada artefacto significativo del sistema en un nodo determinado.
- Hay que tener en cuenta la duplicación de ubicaciones para artefactos. No es infrecuente que el mismo tipo de artefactos (tales como ejecutables y bibliotecas específicas) resida simultáneamente en varios nodos.
- Hay que representar esta localización de una de estas tres formas:
 1. No haciéndola visible, pero dejándola como una parte de la especificación del modelo (es decir, en la especificación de cada nodo).
 2. Conectando cada nodo con el artefacto que despliega, a través de relaciones de dependencia.
 3. Listando los artefactos desplegados sobre un nodo en un compartimento adicional.

Utilizando la tercera forma, la Figura 27.6 parte de los diagramas anteriores y especifica los artefactos ejecutables que hay en cada nodo. Este diagrama es un poco diferente de los anteriores en el hecho de que es un diagrama de objetos, que representa instancias específicas de cada nodo. En este caso, las instancias de *conjunto RAID* y *terminal* son ambas anónimas y las otras dos instancias tienen nombres (*c* la *consola* y *s* el *servidor*). Cada procesador en esta

figura se representa con un compartimento adicional que muestra los artefactos que despliega. El objeto servidor también se representa con sus atributos (*velocidadDelProcesador* y *memoria*) y los valores de éstos. El compartimento de despliegue puede mostrar una lista textual de nombres de artefactos, o puede mostrar símbolos de artefactos anidados.

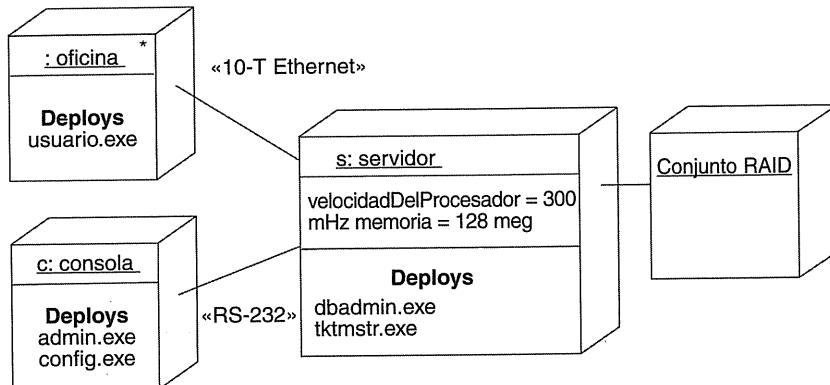


Figura 27.6: Modelado de la distribución de artefactos.

Sugerencias y consejos

Un nodo bien estructurado:

- Proporciona una abstracción bien definida de alguna cosa extraída del vocabulario del hardware en el dominio de la solución.
- Se descompone sólo hasta el nivel necesario para comunicar la intención al lector.
- Sólo muestra aquellos atributos y operaciones relevantes para el dominio que se está modelando.
- Despliega directamente un conjunto de artefactos que residen en el nodo.
- Está conectado con otros nodos de forma que refleje la topología de un sistema del mundo real.

Cuando se dibuje un nodo en UML:

- A nivel del proyecto o de la organización global, hay que definir un conjunto de estereotipos con los iconos apropiados para proporcionar señales visuales significativas para los lectores.
- Hay que mostrar sólo aquellos atributos y operaciones (si los hay) que sean necesarios para comprender el significado de ese nodo en el contexto dado.



Capítulo 28

COLABORACIONES

LENGUAJE
UNIFICADO DE
MODELADO



En este capítulo

- Colaboraciones, realizaciones e interacciones.
- Modelado de la realización de un caso de uso.
- Modelado de la realización de una operación.
- Modelado de un mecanismo.
- Materialización de interacciones.

En el contexto de la arquitectura de un sistema, una colaboración permite nombrar a un bloque conceptual que incluye aspectos tanto estáticos como dinámicos. Una colaboración denota una sociedad de clases, interfaces y otros elementos que colaboran para proporcionar algún comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos.

Las colaboraciones se utilizan para especificar la realización de casos de uso y operaciones, y para modelar los mecanismos significativos desde el punto de vista de la arquitectura del sistema.

Introducción

Piense en el edificio más bello que haya visto jamás, quizás el Taj Mahal o Notre Dame. Ambas estructuras exhiben una calidad difícil de definir con un nombre. En muchos aspectos, ambas estructuras son arquitectónicamente simples, aunque también son bastante complejas. En cada una de ellas podemos reconocer inmediatamente una simetría consistente. Mirando con más atención, podemos ver detalles que son bellos por sí mismos y que colaboran para producir una belleza y funcionalidad mayor que las partes individuales.

Ahora piense en el edificio más horrible que haya visto jamás, quizás el puesto de comida rápida de la esquina. En él se puede observar una disonancia visual

de estilos arquitectónicos (un toque de modernismo combinado con un techo georgiano, todo decorado con un gusto horrible, con colores chillones que ofenden a la vista). Normalmente, estos edificios son una manipulación total, con una funcionalidad limitada y casi ninguna forma.

¿Cuál es la diferencia entre estos dos tipos de arquitectura civil? En primer lugar, en los edificios de calidad se descubre una armonía de diseño que falta en los otros. La arquitectura de calidad utiliza un pequeño conjunto de estilos arquitectónicos que son aplicados de forma consistente. Por ejemplo, el Taj Mahal utiliza por todas partes elementos geométricos complejos, simétricos y equilibrados. En segundo lugar, en los edificios de calidad aparecen patrones comunes que trascienden los elementos individuales del edificio. Por ejemplo, en Notre Dame, ciertos muros son de carga y sirven para soportar la cúpula de la catedral. Algunos de esos mismos muros, en combinación con otros detalles arquitectónicos, sirven como parte del sistema de evacuación de agua y residuos del edificio.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2.

Lo mismo ocurre con el software. Un sistema con gran cantidad de software que sea de calidad no es sólo funcionalmente correcto, sino que exhibe una armonía y un equilibrio en el diseño que lo hace flexible al cambio. Esta armonía y equilibrio provienen, la mayoría de las veces, del hecho de que todos los sistemas orientados a objetos bien estructurados están llenos de patrones. Si observamos cualquier sistema orientado a objetos de calidad, veremos elementos que colaboran de formas bien definidas para proporcionar algún comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. En un sistema bien estructurado, muchos de los elementos participarán en diferentes mecanismos, combinados de diferentes formas.

Los patrones y frameworks se discuten en el Capítulo 29.

Nota: Un patrón proporciona una buena solución a un problema común en algún contexto. En cualquier sistema bien estructurado, se puede descubrir un amplio espectro de patrones, incluidos algunos usos particulares del lenguaje de programación (*idioms*), que representan estilos comunes de programar, mecanismos (patrones de diseño que representan los bloques conceptuales de la arquitectura de un sistema) y *frameworks* (patrones arquitectónicos que proporcionan plantillas extensibles para las aplicaciones dentro de un dominio).

El modelado estructural se discute en las Partes 2 y 3; el modelado del comportamiento se discute en las Partes 4 y 5; las interacciones se discuten en el Capítulo 16.

En UML, los mecanismos se modelan con colaboraciones. Una colaboración proporciona un nombre a los bloques de construcción de interacciones del sistema, incluyendo elementos tanto estructurales como de comportamiento. Por ejemplo, podría haber un sistema de información de gestión distribuido cuyas bases de datos estuviesen repartidas entre varios nodos. Desde la perspectiva del usuario, la actualización de la información parece una acción atómica; desde la perspectiva

interna no es tan sencillo, porque tal acción debe afectar a muchas máquinas. Para dar la ilusión de que es algo sencillo, se puede diseñar un mecanismo de transacciones con el que el cliente pueda dar un nombre a lo que parece una transacción atómica simple, incluso entre varias bases de datos. Tal mecanismo implicaría la colaboración de varias clases con el fin de llevar a cabo una transacción. Muchas de estas clases también estarán implicadas en otros mecanismos, tales como los mecanismos para hacer persistente la información. Esta colección de clases (la parte estructural), junto con sus interacciones (la parte de comportamiento), forman un mecanismo, que en UML se puede representar como una colaboración.

Los casos de uso se discuten en el Capítulo 17; las operaciones se discuten en los Capítulos 4 y 9.

Las colaboraciones no sólo proporcionan un nombre para los mecanismos de un sistema; también sirven como la realización de casos de uso y operaciones.

UML proporciona una representación gráfica para las colaboraciones, como se muestra en la Figura 28.1. Esta notación permite visualizar los bloques de construcción estructurales y de comportamiento de un sistema, sobre todo porque en ellas se pueden solapar las clases, interfaces y otros elementos del sistema.

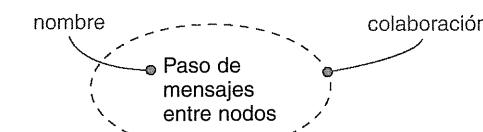


Figura 28.1: Colaboraciones.

Los diagramas de clases se discuten en el Capítulo 8; los diagramas de interacción se discuten en el Capítulo 19.

Nota: Esta notación permite visualizar una colaboración desde el exterior como un único bloque. A menudo, lo más interesante es lo que hay dentro de esta notación. Si se mira dentro de una colaboración, aparecen otros diagramas, sobre todo diagramas de clases (para la parte estructural de la colaboración) y diagramas de interacción (para la parte de comportamiento de la colaboración).

Términos y conceptos

La notación de las colaboraciones es intencionalmente similar a la de los casos de uso, como se discute en el Capítulo 17.

Una *colaboración* es una sociedad de clases, interfaces y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Una colaboración también es la especificación de cómo un elemento, tal como un clasificador (una clase, interfaz, componente, nodo o caso de uso) o una operación, es realizado mediante un con-

junto de clasificadores y asociaciones que desempeñan roles específicos utilizados de una forma específica. Gráficamente, una colaboración se representa como una elipse con el borde discontinuo.

Nombres

El nombre de una colaboración debe ser único dentro del paquete que lo contiene, como se discute en el Capítulo 12.

Cada colaboración debe tener un nombre que la distinga de otras colaboraciones. Un *nombre* es una cadena de texto. Ese nombre solo se denomina *nombre simple*; un *nombre calificado* consta del nombre de la colaboración precedido del nombre del paquete en el que se encuentra. Normalmente, una colaboración se dibuja mostrando sólo su nombre, como se ve en la figura anterior.

Nota: El nombre de una colaboración puede ser texto formado por cualquier número de letras, números y ciertos signos de puntuación (excepto signos como los dos puntos, que se utilizan para separar el nombre de una colaboración y el del paquete que la contiene) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de las colaboraciones son nombres cortos o grupos de palabras basadas en un nombre, extraídos del vocabulario del sistema que se está modelando. Normalmente se pone en mayúsculas la primera letra del nombre de una colaboración, como en Transacción o Cadena de responsabilidad.

Estructura

Los elementos estructurales se discuten en las Partes 2 y 3.

Las colaboraciones tienen dos aspectos: una parte estructural que especifica las clases, interfaces y otros elementos que colaboran para llevar a cabo la colaboración a la que se da nombre, y una parte de comportamiento que especifica la dinámica de cómo interactúan esos elementos.

La parte estructural de una colaboración es una estructura interna (compuesta) que puede incluir cualquier combinación de clasificadores, como clases, interfaces, componentes y nodos. Dentro de una colaboración, estos clasificadores pueden organizarse con las relaciones habituales de UML, incluyendo asociaciones, generalizaciones y dependencias. De hecho, los aspectos estructurales de una colaboración pueden utilizar todo el abanico de recursos de modelado estructural de UML.

Sin embargo, al contrario que una clase estructurada, una colaboración no contiene a ninguno de sus elementos estructurales. En vez de ello, una colaboración simplemente hace referencia o utiliza clases, interfaces, componentes, nodos y

Los elementos estructurales se discuten en las Partes 2 y 3. Los clasificadores se discuten en el Capítulo 9; las relaciones se discuten en los Capítulos 5 y 10; la estructura interna se discute en el Capítulo 15; los paquetes se discuten en el Capítulo 12; los subsistemas se discuten en el Capítulo 32; los casos de uso se discuten en el Capítulo 17.

Los diagramas de clases se discuten en el Capítulo 8.

otros elementos estructurales que han sido declarados en otro sitio. Por eso, una colaboración da nombre a un bloque conceptual (no a un bloque físico) de la arquitectura de un sistema. Por lo tanto, una colaboración puede atravesar muchos niveles de un sistema. Además, un mismo elemento puede aparecer en más de una colaboración (y otros elementos no formarán parte de ninguna colaboración).

Por ejemplo, dado un sistema de ventas basado en la Web, descrito por una docena de casos de uso o más (tales como Comprar Artículos, Devolver Artículos y Consultar Pedido), cada caso de uso será realizado por una única colaboración. Además, cada una de estas colaboraciones compartirá algunos elementos estructurales con las otras colaboraciones (como las clases Cliente y Pedido), pero se organizarán de forma diferente. También existirán colaboraciones que tendrán que ver con aspectos más internos del sistema, las cuales representarán mecanismos significativos desde el punto de vista arquitectónico. Por ejemplo, en este mismo sistema de ventas, podría haber una colaboración llamada Paso de mensajes entre nodos que especificaría los detalles del envío seguro de mensajes entre nodos.

Dada una colaboración que proporciona un nombre a un bloque conceptual de un sistema, se puede mirar dentro para ver los detalles estructurales de sus partes. Por ejemplo, la Figura 28.2 muestra el resultado de mirar dentro de la colaboración Paso de mensajes entre nodos, y la aparición del conjunto de clases que es representado como un diagrama de clases.

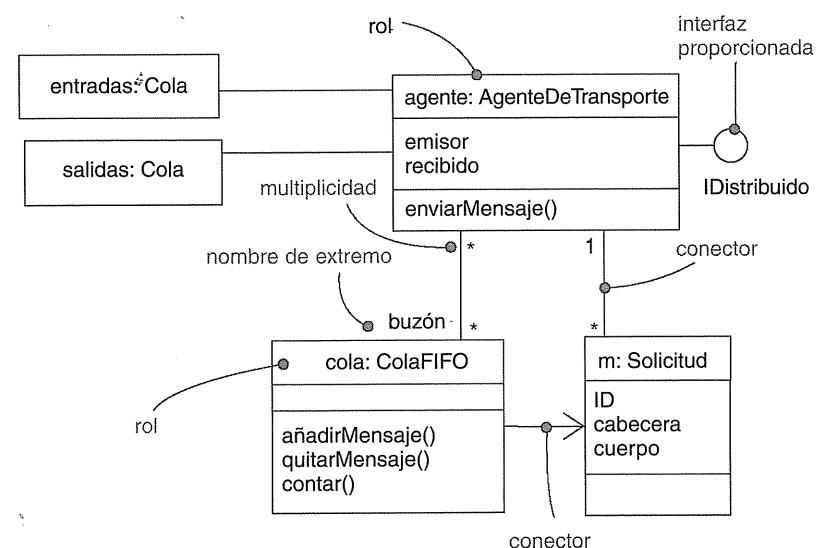


Figura 28.2: Aspectos estructurales de una colaboración.

Comportamiento

Los diagramas de interacción se discuten en el Capítulo 19; las instancias se discuten en el Capítulo 13; la estructura compuesta se discute en el Capítulo 15.

Mientras que la parte estructural de una colaboración se representa normalmente mediante un diagrama de estructura compuesta, la parte de comportamiento de una colaboración se representa normalmente mediante un diagrama de interacción. Un diagrama de interacción especifica una interacción, la cual representa un comportamiento formado por un conjunto de mensajes que se intercambian entre un conjunto de objetos dentro de un contexto, con el fin de conseguir un propósito específico. El contexto de una interacción está dado por la colaboración que la contiene, la cual establece las clases, interfaces, componentes, nodos y otros elementos estructurales cuyas instancias pueden participar en esa interacción.

La parte de comportamiento de una colaboración puede venir especificada por uno o más diagramas de interacción. Si se quiere destacar la ordenación temporal de los mensajes, se puede utilizar un diagrama de secuencia. Si se quiere destacar las relaciones estructurales entre los objetos que colaboran, se puede utilizar un diagrama de comunicación. Cualquiera de los dos diagramas es apropiado porque, a casi todos los efectos, ambos son semánticamente equivalentes.

Esto significa que, cuando se modela una sociedad de clases y se nombra su interacción como una colaboración, se puede mirar dentro de esa colaboración para ver los detalles de su comportamiento. Por ejemplo, al mirar dentro de la colaboración llamada Paso de mensajes entre nodos, aparecería el diagrama de interacción que se muestra en la Figura 28.3.

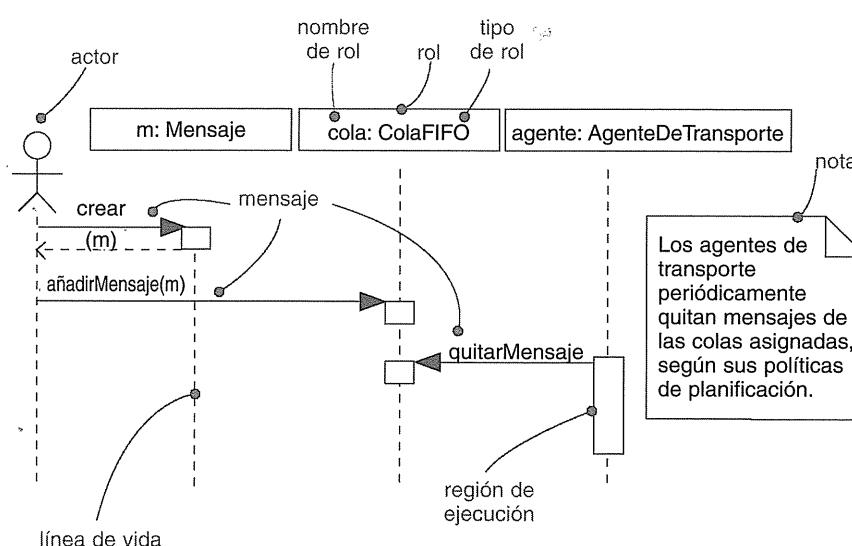


Figura 28.3: Aspectos de comportamiento de una colaboración.

Nota: Las partes de comportamiento de una colaboración deben ser consistentes con sus partes estructurales. Esto significa que los roles que aparecen en las interacciones de una colaboración deben coincidir con los roles de su estructura interna. Análogamente, los mensajes con nombre de una interacción deben referirse a operaciones visibles en la parte estructural de la colaboración. Se puede tener más de una interacción asociada a una colaboración, cada una de las cuales puede mostrar un aspecto diferente (pero consistente) de su comportamiento.

Organización de colaboraciones

El núcleo de la arquitectura de un sistema se encuentra en sus colaboraciones, ya que los mecanismos que configuran un sistema representan decisiones de diseño importantes. Cualquier sistema orientado a objetos bien estructurado se compone de un conjunto relativamente pequeño y sistemático de colaboraciones; así que es importante organizar bien las colaboraciones. Hay dos tipos de relaciones que tienen que ver con las colaboraciones y que será necesario considerar en el modelado.

Los casos de uso se discuten en el Capítulo 17; las operaciones se discuten en los Capítulos 4 y 9; las relaciones de realización se discuten en el Capítulo 10.

Los clasificadores se discuten en el Capítulo 9.

En primer lugar, está la relación entre una colaboración y el elemento al que realiza. Una colaboración puede realizar un clasificador o una operación, lo que significa que la colaboración especifica la realización estructural y de comportamiento de ese clasificador u operación. Por ejemplo, un caso de uso (que denota un conjunto de secuencias de acciones que un sistema ejecuta) puede ser realizado por una colaboración. Ese caso de uso, incluidos sus actores, y los casos de uso con los que está relacionado, proporcionan un contexto para la colaboración. Análogamente, una operación (que denota la implementación de un servicio) puede ser realizada por una colaboración. Esa operación, incluyendo sus parámetros y el posible valor de retorno, también proporciona un contexto para la colaboración. La relación entre un caso de uso o una operación y la colaboración que lo realiza se modela como una relación de realización.

Nota: Una colaboración puede realizar cualquier tipo de clasificador, incluyendo clases, casos de uso, interfaces y componentes. Una colaboración que modele un mecanismo del sistema también puede ir sola y, por lo tanto, su contexto es el sistema global.

En segundo lugar, está la relación entre colaboraciones. Unas colaboraciones pueden refinar a otras colaboraciones, y esta relación también se modela como

un refinamiento. Las relaciones de refinamiento entre colaboraciones normalmente reflejan las relaciones de refinamiento entre los casos de uso que representan.

La Figura 28.4 ilustra estos dos tipos de relaciones.

Los paquetes se discuten en el Capítulo 12.

Nota: Las colaboraciones, como cualquier otro elemento de modelo de UML, pueden agruparse en paquetes. Normalmente, esto sólo será necesario en los sistemas muy grandes.

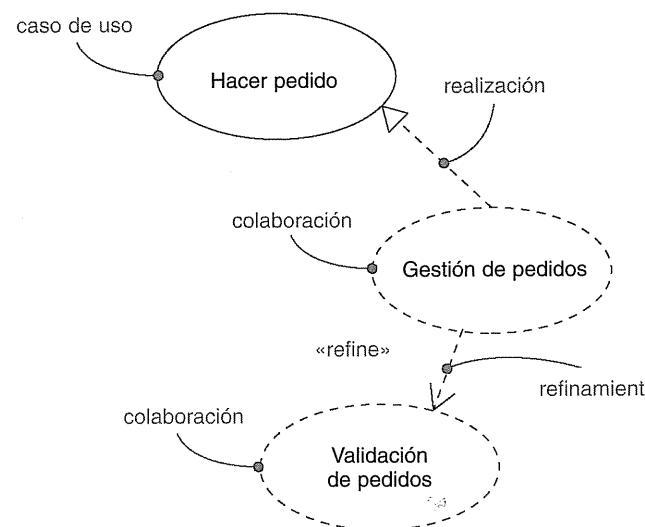


Figura 28.4: Organización de colaboraciones.

Técnicas comunes de modelado

Modelado de roles

Las interacciones se discuten en los Capítulos 16 y 19; la estructura interna se discute en el Capítulo 15.

Los objetos representan elementos individuales en una situación o en ejecución. Son útiles dentro de ejemplos concretos, pero la mayoría de las veces queremos representar partes genéricas dentro de un contexto. Una parte dentro de un contexto se denomina *rol*. Quizás la cosa más importante para la que se utilizan los roles sea para modelar las interacciones dinámicas. Cuando se modelan esas interacciones, normalmente no se modelan instancias concretas del mundo real,

sino roles dentro de un patrón reutilizable, dentro del cual los roles son básicamente representantes (*proxies*) de los objetos que aparecerán dentro de las instancias individuales del patrón. Por ejemplo, si quisieramos modelar la forma en la que una aplicación de ventanas reacciona a un evento del ratón, dibujaríamos un diagrama de interacción con roles cuyos tipos incluirían ventanas, eventos y manejadores.

Para modelar roles:

- Hay que identificar un contexto dentro del cual interactúen los objetos.
- Hay que identificar aquellos roles necesarios y suficientes para visualizar, especificar, construir o documentar el contexto que se está modelando.
- Hay que representar esos roles en UML como roles en un contexto estructurado. Si es posible, hay que dar un nombre a cada rol. Si no hay un nombre significativo para el rol, hay que representarlo como un rol anónimo.
- Hay que exponer las propiedades necesarias y suficientes de cada rol para modelar el contexto.
- Hay que representar los roles y sus relaciones en un diagrama de interacción o en un diagrama de clases.

Nota: La diferencia semántica entre objetos concretos y roles es sutil, pero no es difícil. Para ser preciso, un rol de UML es una parte predefinida de un clasificador estructurado, como una clase estructurada o una colaboración. Un rol no es un objeto, sino una descripción; un rol está ligado a un valor dentro de cada instancia de un clasificador estructurado. Un rol, por tanto, se corresponde con muchos valores posibles, como ocurre con un atributo. Los objetos concretos aparecen en ejemplos específicos, como los diagramas de objetos, los diagramas de componentes y los diagramas de despliegue. Los roles aparecen en descripciones genéricas, como los diagramas de interacción y los diagramas de actividades.

Los diagramas de interacción se discuten en el Capítulo 19; los diagramas de actividades se discuten en el Capítulo 20.

La Figura 28.5 muestra un diagrama de interacción que ilustra un escenario parcial para iniciar una llamada telefónica en el contexto de una centralita. Existen cuatro roles: *a* (un AgenteDeLlamada), *c* (una Conexión), y *t1* y *t2* (ambas instancias de Terminal). Estos cuatro roles son representantes conceptuales de objetos concretos que pueden existir en el mundo real.

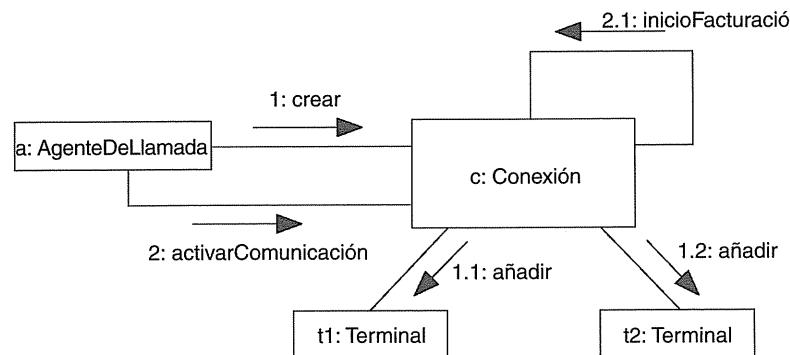


Figura 28.5: Modelado de roles.

Nota: Este ejemplo es una colaboración, que representa una sociedad de objetos y otros elementos que colaboran para proporcionar un comportamiento cooperativo que es mayor que la suma de todos los elementos. Las colaboraciones presentan dos aspectos: uno estructural (que muestra los roles de los clasificadores y sus relaciones) y otro dinámico (que representa las interacciones entre esas instancias prototípicas).

Modelado de la realización de un caso de uso

Los casos de uso se discuten en el Capítulo 17.

Uno de los propósitos para los que se utilizan las colaboraciones es el modelado de la realización de un caso de uso. Normalmente, el análisis del sistema estará dirigido por la identificación de los casos de uso del sistema, pero cuando finalmente se llegue a la implementación, será necesario realizar estos casos de uso con estructuras y comportamientos concretos. En general, cada caso de uso debería ser realizado por una o más colaboraciones. En el sistema global, los clasificadores involucrados en una colaboración ligada a un caso de uso también participarán en otras colaboraciones. Así, los contenidos estructurales de las colaboraciones tienden a solaparse unos con otros.

Para modelar la realización de un caso de uso:

- Hay que identificar aquellos elementos estructurales necesarios y suficientes para desarrollar la semántica del caso de uso.
- Hay que capturar la organización de esos elementos estructurales en diagramas de clases.

- Hay que considerar los escenarios individuales que ese caso de uso representa. Cada escenario representa un recorrido concreto a través del caso de uso.
- Hay que capturar la dinámica de esos escenarios en diagramas de interacción. Si se quiere destacar la ordenación temporal de los mensajes hay que utilizar diagramas de secuencia. Si se quiere destacar las relaciones estructurales entre los objetos que colaboran hay que utilizar diagramas de comunicación.
- Hay que organizar esos elementos estructurales y de comportamiento como una colaboración que se pueda conectar al caso de uso a través de una realización.

Por ejemplo, la Figura 28.6 muestra un conjunto de casos de uso extraídos de un sistema de validación de tarjetas de crédito, incluyendo los casos de uso principales Hacer pedido y Generar factura, junto con otros dos casos de uso subordinados, Detectar fraude en tarjeta y Validar transacción. Aunque la mayoría de las veces no será necesario modelar explícitamente esta relación (será algo que se deje a las herramientas), esta figura modela explícitamente la realización de Hacer pedido por la colaboración Gestión de pedidos. A su vez, esta colaboración puede detallarse más en sus aspectos estructural y de comportamiento, lo que lleva a los diagramas de clases y de interacción. Un caso de uso se conecta a sus escenarios a través de la relación de realización.

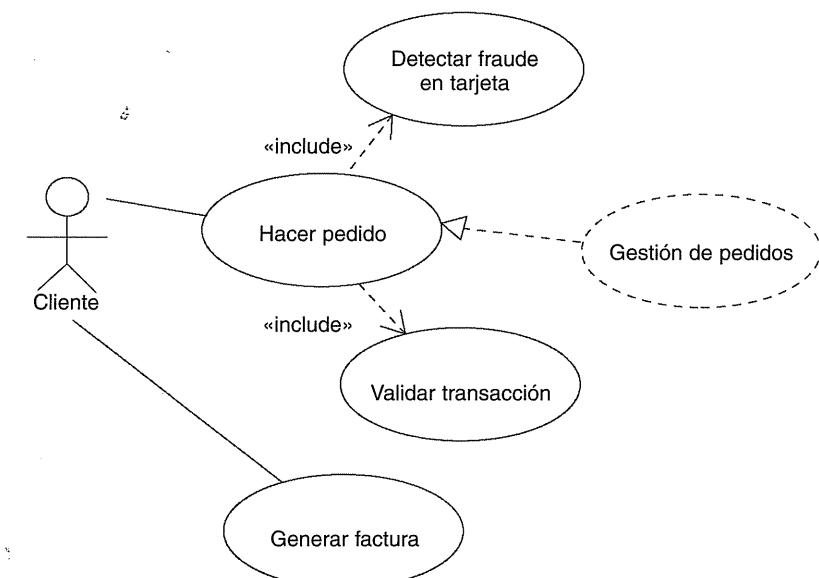


Figura 28.6: Modelado de la realización de un caso de uso.

En la mayoría de los casos no será necesario modelar explícitamente la relación entre un caso de uso y la colaboración que lo realiza, sino que se tenderá a dejar esto en la especificación del modelo. Entonces las herramientas podrán usar esa conexión para ayudar al usuario a navegar entre un caso de uso y su realización.

Modelado de la realización de una operación

Las operaciones se discuten en los Capítulos 4 y 9.

Otro propósito para el que se utilizan las colaboraciones es el modelado de la realización de una operación. En muchos casos, la realización de una operación se puede especificar pasando directamente al código. Sin embargo, para aquellas operaciones que requieren la colaboración de varios objetos, es mejor modelar su implementación a través de colaboraciones antes de pasar al código.

Los parámetros, el valor de retorno y los objetos locales a una operación establecen el contexto de su realización. Por lo tanto, estos elementos son visibles para la parte estructural de la colaboración que realiza la operación, del mismo modo que los actores son visibles para la parte estructural de una colaboración que realiza un caso de uso. La relación entre estos elementos se puede modelar mediante un diagrama de estructura compuesta, que especifica la parte estructural de una colaboración.

Para modelar la implementación de una operación:

- Hay que identificar los parámetros, el valor de retorno y otros objetos visibles en la operación. Éstos se convierten en los roles de la colaboración.
- Si la operación es trivial, se puede representar su implementación directamente en código, que se puede ocultar en la especificación del modelo, o se puede visualizar explícitamente en una nota.
- Si la operación tiene una gran componente algorítmica, hay que modelar su realización mediante un diagrama de actividades.
- Si la operación es compleja o requiere un diseño detallado, hay que representar su implementación con una colaboración. Se pueden detallar más las partes estructural y de comportamiento de la colaboración mediante diagramas de clases y de interacción, respectivamente.

Las notas se discuten en el Capítulo 6.

Por ejemplo, en la Figura 28.7 se ve la clase activa *MarcoGráfico* mostrando tres de sus operaciones. La función *progreso* es lo bastante sencilla y se implementa en código directamente, como se especifica en la nota adjunta. Sin

Las clases activas se discuten en el Capítulo 23.

embargo, la operación *dibujar* es mucho más complicada, así que su implementación se realiza mediante la colaboración *Trazado de Rayos*. Aunque no se muestra aquí, es posible mirar dentro de la colaboración para ver sus aspectos estructurales y de comportamiento.

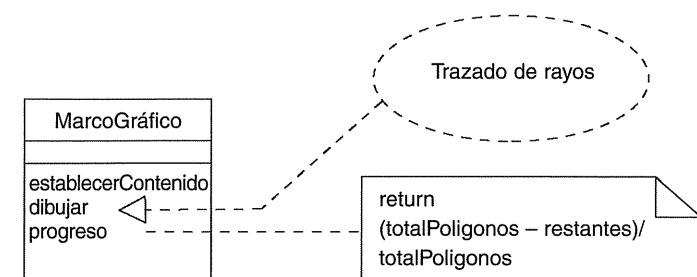


Figura 28.7: Modelado de la realización de una operación.

Los diagramas de actividades se discuten en el Capítulo 20.

Nota: También se puede modelar una operación utilizando diagramas de actividades, que son básicamente diagramas de flujo. Así que para aquellas operaciones con una gran componente algorítmica que queramos modelar de manera explícita, la mejor elección suelen ser los diagramas de actividades. No obstante, si la operación requiere la participación de muchos objetos, podemos usar las colaboraciones, porque permiten modelar tanto los aspectos estructurales de la operación como los de comportamiento.

Modelado de un mecanismo

Los patrones y frameworks se discuten en el Capítulo 29; un ejemplo del modelado de un mecanismo se discute en el mismo capítulo.

En cualquier sistema orientado a objetos bien estructurado aparece una amplia variedad de patrones. Por un lado, aparecen construcciones que representan patrones de utilización del lenguaje de implementación (*idioms*). En el otro extremo, aparecen patrones arquitectónicos y *frameworks* que configuran el sistema global e imponen un estilo particular. Entre ambos extremos, aparecen mecanismos que representan patrones de diseño comunes, por medio de los cuales interactúan los elementos del sistema de diferentes formas comunes. Un mecanismo se puede representar en UML como una colaboración.

Los mecanismos son colaboraciones que pueden encontrarse aisladas; en ese caso, su contexto no es un único caso de uso o una operación, sino el sistema global. Cualquier elemento visible en esa parte del sistema es candidato a participar en un mecanismo.

Mecanismos de esta naturaleza representan decisiones de diseño importantes desde el punto de vista arquitectónico, y no deberían ser tratados a la ligera. Normalmente, el arquitecto del sistema diseñará sus mecanismos, y éstos irán evolucionando con cada nueva versión del sistema. Al final, el sistema parecerá simple (debido a que estos mecanismos materializan interacciones comunes), comprensible (debido a que se puede comprender el sistema a partir de sus mecanismos) y flexible (cuando se ajusta cada mecanismo, se está ajustando el sistema global).

Para modelar un mecanismo:

- Hay que identificar los principales mecanismos que configuran la arquitectura del sistema. Estos mecanismos están dirigidos por un estilo arquitectónico global que se quiere imponer a la implementación, además del estilo apropiado al dominio del problema.
- Hay que representar cada uno de estos mecanismos como una colaboración.
- Hay que detallar las partes estructural y de comportamiento de cada colaboración. Hay que procurar compartir, donde esto sea posible.
- Hay que validar estos mecanismos en las primeras etapas del ciclo de vida del desarrollo (tienen una importancia estratégica), pero, conforme se va aprendiendo más sobre los detalles de la implementación, deben evolucionar apropiadamente.

Sugerencias y consejos

Cuando se modelan colaboraciones en UML, debe recordarse que cada colaboración debería representar la realización de un caso de uso o una operación, o bien se encontrará aislada como un mecanismo del sistema. Una colaboración bien estructurada:

- Consta de aspectos tanto estructurales como de comportamiento.
- Proporciona una abstracción nítida de alguna interacción identificable del sistema.
- No suele ser totalmente independiente, sino que se solapa con los elementos estructurales de otras colaboraciones.
- Es comprensible y simple.

Cuando se dibuje una colaboración en UML:

- Hay que representar explícitamente una colaboración sólo cuando sea necesario comprender sus relaciones con otras colaboraciones, clasificadores, operaciones o el sistema global. En otro caso, hay que utilizar las colaboraciones, pero deben mantenerse en la especificación del sistema.
- Hay que organizar las colaboraciones según el clasificador o la operación que representan, o en paquetes asociados al sistema global.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 29

PATRONES Y *FRAMEWORKS*

En este capítulo

- Patrones y *frameworks*.
- Modelado de patrones de diseño.
- Modelado de patrones arquitectónicos.
- Obtención de patrones asequibles.

Todos los sistemas bien estructurados están llenos de patrones. Un patrón proporciona una solución buena a un problema común en un contexto dado. Un mecanismo es un patrón de diseño que se aplica a una sociedad de clases; un *framework* es normalmente un patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio.

Los patrones se utilizan para especificar los mecanismos y *frameworks* que configuran la arquitectura del sistema. Un patrón se puede hacer asequible identificando de manera clara todos los elementos variables que puede ajustar un usuario del patrón para aplicarlo en un contexto particular.

Introducción

El modelado de cosas que no son software se discute en el Capítulo 4; las cinco vistas de una arquitectura se discuten en el Capítulo 2.

Es sorprendente imaginar la cantidad de maneras en las que se puede combinar un montón de tablones de madera para construir una casa. En manos de un constructor de San Francisco, podríamos ver cómo se transforman esos tablones en una casa de estilo victoriano, con un tejado a dos aguas y paredes de colores llamativos, como una casa de los cuentos de los niños. En manos de un constructor de Maine, esos mismos tablones podrían transformarse en una casa en forma de caja de zapatos, con paredes a base de listones de madera pintados de blanco y con formas rectangulares por todas partes.

Desde el exterior, estas dos casas representan estilos arquitectónicos claramente diferentes. Cada constructor, a partir de su experiencia, debe elegir el estilo que mejor se adapte a las necesidades de su cliente, y después adaptar ese estilo a los deseos del cliente y a las restricciones del terreno y las ordenanzas municipales.

Los estereotipos se discuten en el Capítulo 6.

En el interior, cada constructor también debe diseñar la casa para solucionar algunos problemas comunes. Sólo hay unas pocas formas de diseñar la estructura para que soporte el techo; sólo hay unas pocas formas de diseñar un muro de carga que también deba tener aberturas para puertas y ventanas. Cada constructor debe seleccionar los mecanismos apropiados que solucionen estos problemas comunes, adaptados a un estilo arquitectónico global y a las restricciones de las ordenanzas de construcción locales.

La construcción de un sistema con gran cantidad de software es algo parecido. Cada vez que se mira por encima el código fuente, se detectan ciertos mecanismos comunes que configuran la forma en que se organizan las clases y otras abstracciones. Por ejemplo, en un sistema dirigido por eventos, una forma común de organizar los manejadores de eventos es utilizar el patrón de diseño *Cadena de responsabilidad*. Si se mira por encima de estos mecanismos, aparecen estructuras comunes que configuran la arquitectura completa del sistema. Por ejemplo, en los sistemas de información, la utilización de una arquitectura de tres capas es una forma común de lograr una separación de intereses clara entre la interfaz de usuario del sistema, la información persistente y los objetos y reglas del negocio.

Las colaboraciones se discuten en el Capítulo 28; los paquetes se discuten en el Capítulo 12.

En UML, normalmente se modelarán patrones de diseño (también llamados mecanismos), los cuales se pueden representar como colaboraciones. De forma similar, los patrones arquitectónicos se modelarán normalmente como frameworks, que se pueden representar como paquetes estereotipados.

UML proporciona una representación gráfica para ambos tipos de patrones, como se muestra en la Figura 29.1.

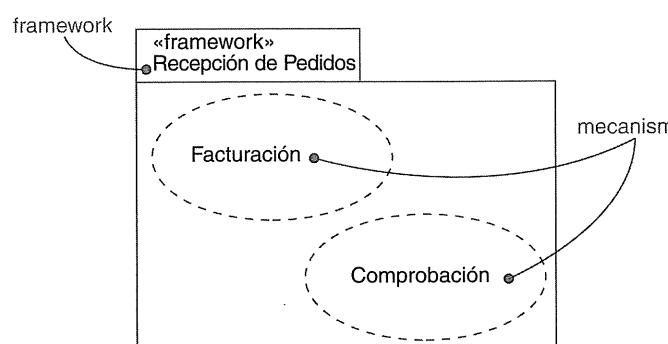


Figura 29.1: Mecanismos y frameworks.

Términos y conceptos

Un *mecanismo* es un patrón de diseño que se aplica a una sociedad de clases. Un *framework* es un patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio.

Patrones y arquitectura

La arquitectura del software se discute en el Capítulo 2.

Tanto si se está diseñando un sistema nuevo como si se está modificando uno existente, en realidad nunca hay que empezar desde cero. Más bien, la experiencia y las convenciones llevan a aplicar formas comunes de resolver problemas comunes. Por ejemplo, si se está construyendo un sistema que interactúa mucho con el usuario, una forma ya probada de organizar las abstracciones es utilizar un patrón modelo-vista-control, con el cual se separan de forma clara los objetos (el modelo) de su presentación (la vista) y los agentes que los mantienen sincronizados (el control). Análogamente, si se va a construir un sistema para resolver criptogramas, una forma probada de organizar el sistema es utilizar una arquitectura de pizarra (*blackboard*), que es apropiada para atacar problemas intratables de manera oportunista.

Los dos ejemplos anteriores son ejemplos de patrones (soluciones comunes a problemas comunes en un contexto determinado). En todos los sistemas bien estructurados aparecen muchos patrones a distintos niveles de abstracción. Los patrones de diseño especifican la estructura y el comportamiento de una sociedad de clases; los patrones arquitectónicos especifican la estructura y el comportamiento de un sistema completo.

Los patrones son parte de UML simplemente porque son una parte importante del vocabulario de un desarrollador. Al hacer explícitos los patrones en un sistema, se hace que éste sea mucho más comprensible y fácil de modificar y mantener. Por ejemplo, si alguien está manejando un gran bloque de código fuente con el propósito de extenderlo, tendrá que pelear bastante tiempo con él, intentando imaginar cómo encaja todo. Por otro lado, si cuando le pasan ese código fuente alguien le dice: "Estas clases colaboran mediante un mecanismo de publicación y suscripción", le será mucho más fácil comprender cómo funciona. La misma idea se aplica al sistema global. El decir: "Este sistema se organiza como un conjunto de tuberías y filtros", explica bastante sobre la arquitectura del sistema, que en otro caso sería difícil comprender simplemente inspeccionando las clases individuales.

Los patrones ayudan a visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software. Se puede hacer ingeniería directa con un sistema, seleccionando un conjunto apropiado de patrones y aplicándolos a las abstracciones específicas del dominio. También se puede hacer ingeniería inversa sobre un sistema descubriendo los patrones que contiene, aunque éste es un proceso que dista mucho de ser perfecto. Aún mejor, cuando se entrega un sistema, se puede especificar qué patrones incorpora, de forma que cuando alguien trata de reutilizar o adaptar el sistema más tarde, los patrones están claramente identificados.

En la práctica, hay dos tipos de patrones interesantes (patrones de diseño y *frameworks*) y UML proporciona medios para modelar ambos. Cuando se modela cualquiera de los dos tipos de patrones, normalmente aparecerán aislados en el contexto de algún paquete, a excepción de las relaciones de dependencia que los ligan a otras partes del sistema.

Mecanismos

Un mecanismo es tan sólo otro nombre para un patrón de diseño que se aplica a una sociedad de clases. Por ejemplo, un problema de diseño común que aparece en Java es adaptar una clase que sabe cómo responder a un conjunto de eventos, de forma que responda a un conjunto ligeramente diferente, sin alterar la clase original. Una solución común a este problema es el patrón adaptador, un patrón de diseño estructural que convierte una interfaz en otra. Este patrón es tan común que tiene sentido darle un nombre y modelarlo, de forma que se pueda utilizar cada vez que aparezca un problema similar.

Cuando se modela, estos mecanismos aparecen de dos formas.

Las colaboraciones se discuten en el Capítulo 28.

En primer lugar, como se muestra en la figura anterior, un mecanismo simplemente nombra un conjunto de abstracciones que colaboran entre sí para llevar a cabo algún comportamiento común e interesante. Estos mecanismos se modelan como colaboraciones simples, ya que tan sólo dan un nombre a una sociedad de clases. Cuando se mira de cerca esa colaboración, se pueden ver tanto los aspectos estructurales (normalmente representados como diagramas de clases) como los aspectos de comportamiento (normalmente representados como diagramas de interacción). Las colaboraciones de esta naturaleza pueden compartir las abstracciones individuales del sistema; una clase dada puede ser miembro de varias colaboraciones.

En segundo lugar, como se muestra en la Figura 29.2, un mecanismo nombra una plantilla formada por un conjunto de abstracciones que colaboran entre sí

para llevar a cabo algún comportamiento común e interesante. Estos mecanismos se modelan como colaboraciones parametrizadas, que se representan en UML de forma parecida a como se representan las clases plantilla. Cuando se observa de cerca la colaboración, se pueden ver los aspectos tanto estructurales como de comportamiento. Cuando se observa de lejos la colaboración, se puede ver cómo se aplica ese patrón en el sistema, ligando los parámetros a abstracciones existentes en el sistema. Cuando se modela un mecanismo como una colaboración parametrizada, se identifican los elementos variables que se utilizan para adaptar ese patrón a través de sus parámetros. Las colaboraciones de esta naturaleza pueden aparecer varias veces en el sistema, ligadas a diferentes conjuntos de abstracciones. En este ejemplo, los parámetros Sujeto y Observador del patrón se han ligado a las clases concretas ColaLlamadas y BarraDesplazamiento, respectivamente.

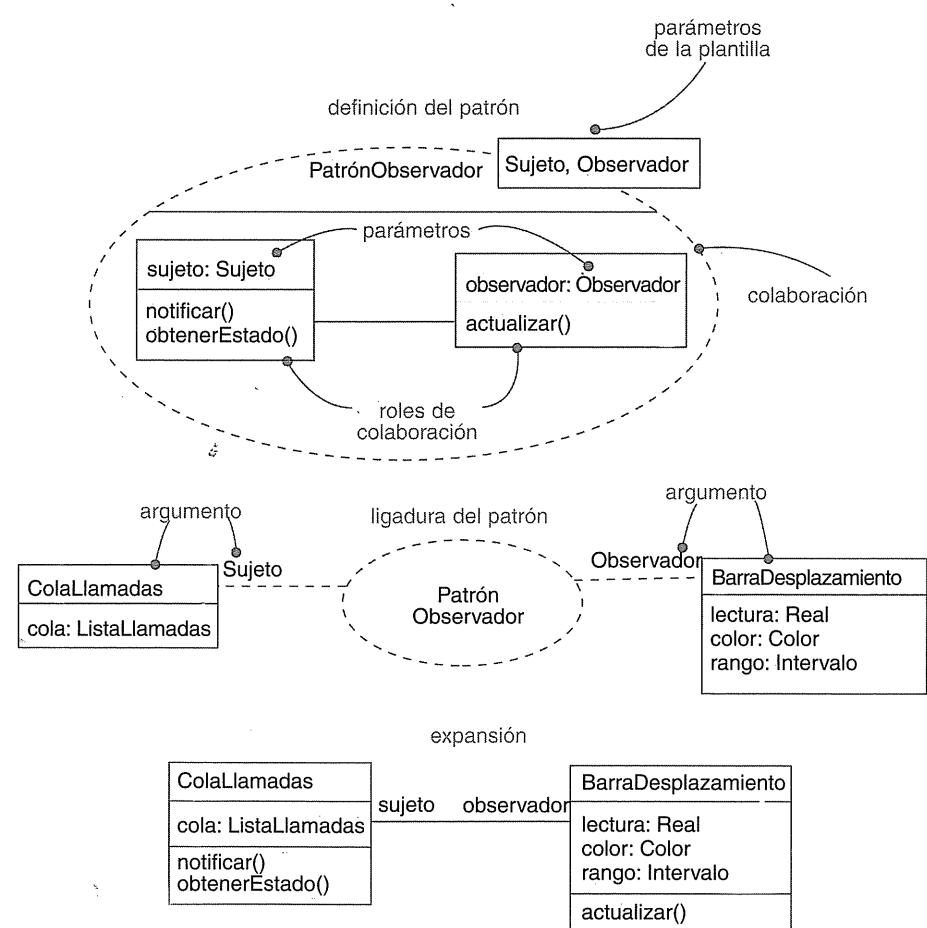


Figura 29.2: Mecanismos.

Nota: La decisión acerca de si un mecanismo se modela como una colaboración simple o una parametrizada es bastante sencilla. Se utilizará una colaboración simple si lo único que se está haciendo es dar un nombre a una sociedad de clases del sistema que colaboran entre sí; se utilizará una colaboración parametrizada si se pueden abstractar los aspectos estructural y de comportamiento esenciales del mecanismo de forma totalmente independiente del dominio, los cuales se pueden ligar, posteriormente, a las abstracciones en un contexto dado.

Frameworks

Un *framework* es un patrón arquitectónico que proporciona una plantilla extensible para aplicaciones dentro de un dominio. Por ejemplo, un patrón arquitectónico común que aparece en los sistemas de tiempo real es un administrador cíclico (*cyclic executive*), que divide el tiempo en intervalos y subintervalos, durante los cuales el procesamiento tiene lugar con limitaciones de tiempo muy estrictas. La elección de este patrón frente a la otra alternativa (una arquitectura dirigida por eventos) determina el aspecto global del sistema. Como este patrón (y su alternativa) es tan común, tiene sentido asociarle un nombre como *framework*.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2.

Un *framework* es algo más grande que un mecanismo. De hecho, se puede pensar en un *framework* como en una microarquitectura que incluye un conjunto de mecanismos que colaboran para resolver un problema común en un dominio común. Cuando se especifica un *framework*, se especifica el esqueleto de una arquitectura, junto a los elementos variables, que se muestran a los usuarios que quieren adaptar el *framework* a su propio contexto.

Los paquetes se discuten en el Capítulo 12; los estereotipos se discuten en el Capítulo 6.

En UML, un *framework* se modela como un paquete estereotipado. Cuando se mira dentro del paquete se pueden ver mecanismos existentes en cualquiera de las diferentes vistas de la arquitectura de un sistema. Por ejemplo, no sólo se pueden encontrar colaboraciones parametrizadas, sino que también se pueden encontrar casos de uso (que explican cómo utilizar el *framework*), así como colaboraciones simples (que proporcionan conjuntos de abstracciones sobre las que se puede construir; por ejemplo, creando subclases).

Los eventos se discuten en el Capítulo 21.

La Figura 29.3 ilustra un *framework*, llamado AdministradorCíclico. Entre otras cosas, este *framework* incluye una colaboración (*EventosComunes*) que contiene un conjunto de clases evento, junto con un mecanismo (*GestorDeEventos*) para procesar estos eventos de forma cíclica. Un cliente que utilice este *framework* (como Marcapasos) podría construir a partir

de las abstracciones existentes en *EventosComunes* creando subclases y también podría emplear una instancia del mecanismo *GestorDeEventos*.

Nota: Los *frameworks* son diferentes de las bibliotecas de clases. Una biblioteca de clases contiene abstracciones que las abstracciones del desarrollador pueden instanciar o invocar; un *framework* contiene abstracciones que pueden instanciar o invocar a las abstracciones del desarrollador. Ambos tipos de conexión constituyen los elementos variables del *framework*, que deben ser ajustados para adaptar el *framework* al contexto específico del problema que se está modelando.

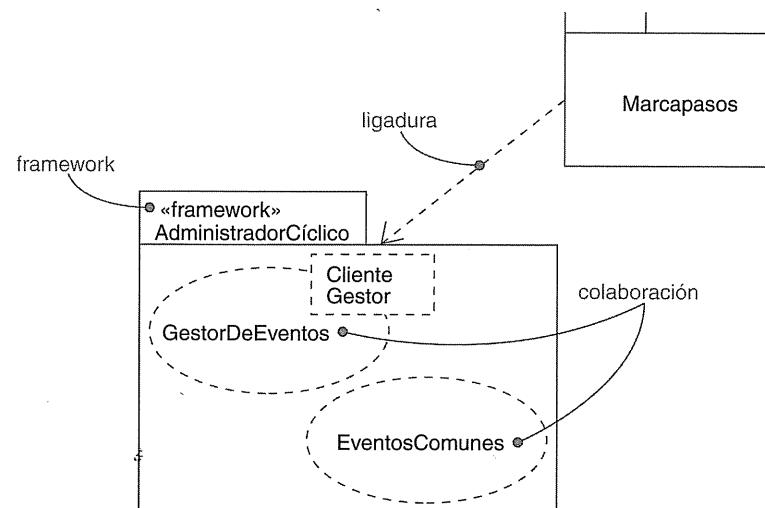


Figura 29.3: Frameworks.

Técnicas comunes de modelado

Modelado de patrones de diseño

Una de las cosas para las que se utilizan los patrones es para modelar patrones de diseño. Cuando se modela un mecanismo como éste, hay que tener en cuenta tanto su vista interna como la externa.

Cuando se ve desde fuera, un patrón de diseño se representa como una colaboración parametrizada. Como colaboración, un patrón proporciona un conjunto de abstracciones cuya estructura y comportamiento colaboran entre sí para llevar a cabo alguna función útil. Los parámetros de la colaboración identifican los elementos que un usuario de este patrón debe ligar a elementos concretos. Esto hace del patrón de diseño una plantilla que se utiliza en un contexto particular, proporcionando elementos que conformen con los parámetros de la plantilla.

Cuando se ve desde dentro, un patrón de diseño es simplemente una colaboración y se representa con sus partes estructural y de comportamiento. Normalmente, la parte interna de esta colaboración se modelará con un conjunto de diagramas de clases (para el aspecto estructural) y con un conjunto de interacciones (para el aspecto de comportamiento). Los parámetros de la colaboración nombran ciertos elementos estructurales, los cuales son instanciados con abstracciones de un contexto particular cuando se liga el patrón de diseño en ese contexto.

Para modelar un patrón de diseño:

- Hay que identificar la solución común al problema común y materializarla como un mecanismo.
- Hay que modelar el mecanismo como una colaboración, proporcionando sus aspectos tanto estructurales como de comportamiento.
- Hay que identificar los elementos del patrón de diseño que deben ser ligados a elementos concretos en un contexto específico y mostrarlos como parámetros de la colaboración.

Por ejemplo, la Figura 29.4 muestra una utilización del patrón de diseño Command (como se describe en Gamma *et al.*, *Design Patterns*, Reading, Massachusetts: Addison-Wesley, 1995¹). Como establece su documentación, este patrón “encapsula una solicitud como un objeto, permitiendo, por lo tanto, parametrizar clientes con diferentes solicitudes, almacenar en una cola o registrar solicitudes y soportar operaciones que se pueden deshacer”. Como indica el modelo, este patrón de diseño tiene tres parámetros que deben ser ligados a elementos de un contexto determinado cuando se aplique el patrón. Este modelo muestra dos posibles ligaduras, en las que se ligan OrdenPegar y OrdenAbrir a dos instancias diferentes del patrón.

¹ Traducción al español: *Patrones de Diseño*, Addison-Wesley, 2003.

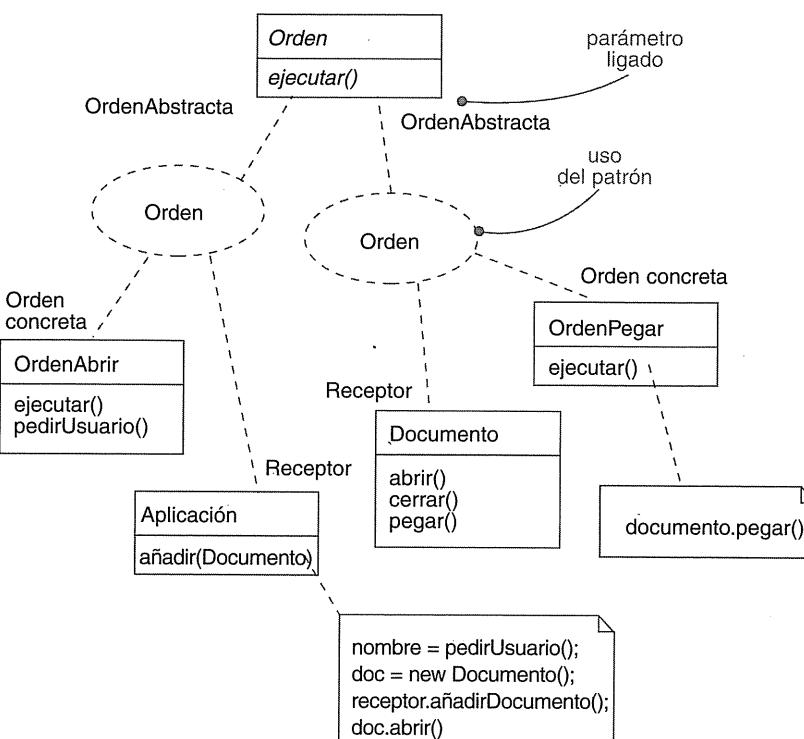


Figura 29.4: Modelado de un patrón de diseño.

Los parámetros son la OrdenAbstracta, que debe ser ligada a la misma superclase abstracta en cada caso; la OrdenConcreta, que se liga a las diferentes clases específicas en diferentes ligaduras; y el Receptor, que se liga a la clase sobre la cual actúa la orden. La clase Orden podría ser creada por el patrón, pero al parametrizarla se permite que se creen distintas jerarquías de órdenes.

Hay que observar que OrdenPegar y OrdenAbrir son subclases de la clase Orden. Probablemente, el sistema utilizará este patrón varias veces, quizás con diferentes ligaduras. Lo que hace que el desarrollo con patrones sea tan potente es la capacidad de reutilizar un patrón de diseño como éste como un elemento de modelado de primera clase.

Para completar el modelo de un patrón de diseño, deben especificarse sus partes estructural y de comportamiento, las cuales representan la parte interna de la colaboración.

Las colaboraciones se discuten en el Capítulo 28; los diagramas de clases se discuten en el Capítulo 8; los diagramas de interacción se discuten en el Capítulo 19.

Por ejemplo, la Figura 29.5 muestra un diagrama de clases que representa la estructura de este patrón de diseño. Este diagrama usa clases cuyos nombres coinciden con los parámetros del patrón. La Figura 29.6 muestra un diagrama de secuencia que representa el comportamiento de este patrón de diseño. Debe tenerse en cuenta que el diagrama tan sólo sugiere una posibilidad: un patrón de diseño no es una cosa inflexible.

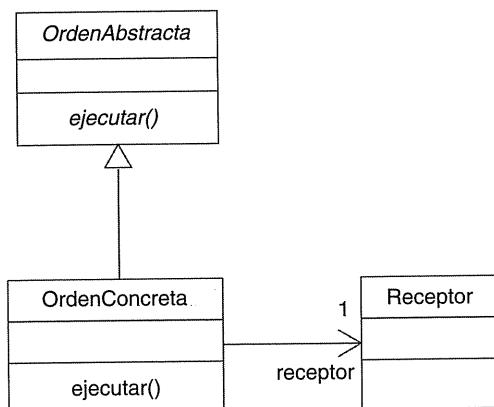


Figura 29.5: Modelado de los aspectos estructurales de un patrón de diseño.

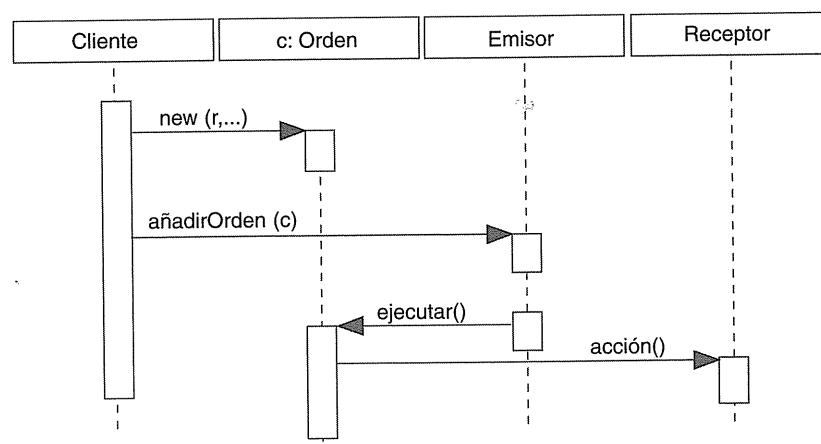


Figura 29.6: Modelado de los aspectos de comportamiento de un patrón de diseño.

Los paquetes se discuten en el Capítulo 12.

Modelado de patrones arquitectónicos

La otra cosa para la que se utilizarán patrones será para modelar patrones arquitectónicos. Cuando se modela un *framework*, se está modelando en realidad la infraestructura de una arquitectura completa que se intenta reutilizar y adaptar a algún contexto.

Un *framework* se representa como un paquete estereotipado. Como paquete, un *framework* proporciona un conjunto de elementos, que incluyen (aunque, por supuesto, no se limitan a) clases, interfaces, casos de uso, componentes, nodos, colaboraciones e incluso otros *frameworks*. De hecho, en un *framework* se colocarán todas las abstracciones que colaboran para proporcionar una plantilla extensible para aplicaciones dentro de un dominio. Algunos de esos elementos serán públicos y representarán recursos sobre los cuales los clientes pueden construir. Éstos son las partes del *framework* que pueden conectarse a las abstracciones del contexto. Algunos de estos elementos públicos serán patrones de diseño y representarán recursos a los que se ligarán los clientes. Éstas son las partes del *framework* que se llenan al establecer ligaduras con el patrón de diseño. Por último, algunos de estos elementos serán protegidos o privados, y representarán elementos encapsulados del *framework* que se ocultan a su vista externa.

La arquitectura del software se discute en el Capítulo 2.

Cuando se modela un patrón arquitectónico, hay que recordar que un *framework* es, de hecho, una descripción de una arquitectura, si bien es incompleta y quizás parametrizada. Como tal, todo lo que sabemos acerca de modelar arquitecturas bien estructuradas se aplica al modelado de *frameworks* bien estructurados. Los mejores *frameworks* no se diseñan de forma aislada; hacer esto sería una garantía de fracaso. En cambio, los mejores *frameworks* se obtienen de arquitecturas que han demostrado que funcionan, y los *frameworks* evolucionan para encontrar los elementos variables necesarios y suficientes para hacerlos adaptables a otros dominios.

Para modelar un patrón arquitectónico:

- Hay que extraer el *framework* de una arquitectura existente que haya sido probada.
- Hay que modelar el *framework* como un paquete estereotipado, que contiene todos los elementos (y especialmente los patrones de diseño) necesarios y suficientes para describir las diferentes vistas de este *framework*.

- Hay que mostrar puntos de conexión, las interfaces y los parámetros necesarios para adaptar el *framework* en forma de patrones de diseño y colaboraciones. La mayoría de las veces, esto significa dejar claro al usuario del patrón qué clases deben ser extendidas, qué operaciones deben ser implementadas y qué señales deben ser manejadas.

Por ejemplo, la Figura 29.7 muestra una especificación del patrón arquitectónico Blackboard (Pizarra) (como se describe en Buschmann *et al.*, *Pattern-Oriented Software Architecture*, New York, NY: Wiley, 1996). Como establece su documentación, este patrón “aborda problemas que no tienen una solución determinista factible para la transformación de datos simples en estructuras de datos de alto nivel”. El corazón de esta arquitectura es el patrón de diseño Pizarra, que determina cómo colaboran las FuentesDeConocimiento, una Pizarra y un Controlador. Este *framework* también incluye el patrón de diseño Motor de Razonamiento, que especifica un mecanismo general que dirige a cada FuenteDeConocimiento. Por último, como se muestra en la figura, este *framework* muestra un caso de uso, Aplicar nuevas fuentes de conocimiento, que explica a un cliente cómo adaptar el propio *framework*.

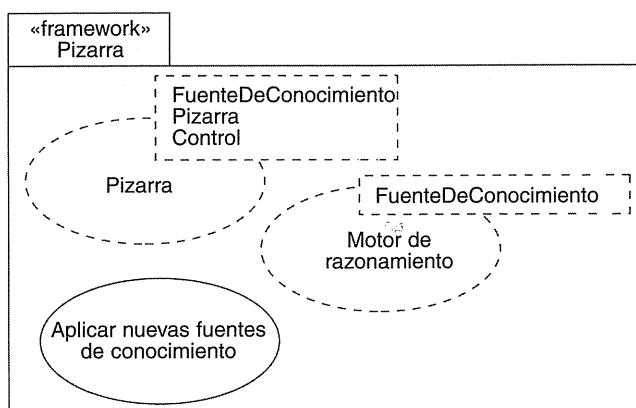


Figura 29.7: Modelado de un patrón arquitectónico.

Nota: En la práctica, el modelado completo de un *framework* es una tarea tan grande como el modelado completo de la arquitectura de un sistema. En ciertos aspectos, la tarea es incluso más difícil, porque para conseguir que el *framework* sea accesible, también hay que mostrar los elementos variables del *framework* y quizás incluso proporcionar metacasos de uso (tales como Aplicar nuevas fuentes de conocimiento) que expliquen cómo adaptar el *framework*, así como casos de uso normales que expliquen cómo se comporta el *framework*.

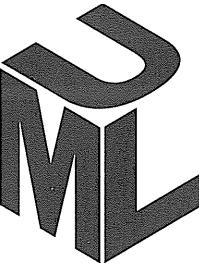
Sugerencias y consejos

Cuando se modelan patrones en UML, debe recordarse que éstos funcionan en muchos niveles de abstracción, desde clases individuales hasta la forma del sistema global. Los tipos de patrones más interesantes son los mecanismos y los *frameworks*. Un patrón bien estructurado:

- Soluciona un problema común de una forma común.
- Consta de aspectos tanto estructurales como de comportamiento.
- Muestra los elementos variables a través de los cuales se adaptan esos aspectos para aplicarlos en un contexto determinado.
- Es atómico, es decir, no puede ser fácilmente descompuesto en patrones más pequeños.
- Abarca diferentes abstracciones individuales del sistema.

Cuando se dibuja un patrón en UML:

- Hay que mostrar los elementos del patrón que hay que adaptar para aplicarlo en un contexto.
- Hay que hacerlo asequible proporcionando casos de uso para su utilización, así como para su adaptación.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 30

DIAGRAMAS DE ARTEFACTOS

En este capítulo

- Modelado de código fuente.
- Modelado de versiones ejecutables.
- Modelado de bases de datos físicas.
- Modelado de sistemas adaptables.
- Ingeniería directa e inversa.

Los diagramas de despliegue, el otro tipo de diagramas que se utilizan para modelar los aspectos físicos de un sistema orientado a objetos, se discute en el Capítulo 31.

Los diagramas de artefactos son uno de los dos tipos de diagramas que aparecen cuando se modelan los aspectos físicos de los sistemas orientados a objetos. Un diagrama de artefactos muestra la organización y las dependencias entre un conjunto de artefactos.

Los diagramas de artefactos se utilizan para modelar la vista de implementación estática de un sistema. Esto implica modelar las cosas físicas que residen en un nodo, como ejecutables, bibliotecas, tablas, archivos y documentos. Los diagramas de artefactos son fundamentalmente diagramas de clases que se centran en los artefactos de un sistema.

Los diagramas de artefactos no sólo son importantes para visualizar, especificar y documentar sistemas basados en artefactos, sino también para construir sistemas ejecutables mediante ingeniería directa e inversa.

Introducción

Al construir una casa, hay que hacer algo más que crear planos. Recordemos que los planos son importantes porque ayudan a visualizar, especificar y documentar el tipo de casa que se quiere construir, de forma que se construya la casa adecuada en el momento adecuado y al precio adecuado. Sin embargo, alguna vez hay

que convertir los planos de planta y los planos de alzado en paredes, suelos y techos reales hechos de madera, piedra o metal. La casa no se construirá sólo a partir de estos materiales simples, sino que también se incorporarán artefactos preconstruidos, como armarios, ventanas, puertas y conductos de ventilación. Si se está remodelando la casa, se reutilizarán incluso artefactos más grandes, como habitaciones enteras y elementos estructurales.

Lo mismo ocurre con el software. Se crean diagramas de casos de uso para razonar sobre el comportamiento deseado del sistema. El vocabulario del sistema se especifica mediante diagramas de clases. Se crean diagramas de secuencia, diagramas de colaboración, diagramas de estados y diagramas de actividades para especificar la forma en que los elementos del vocabulario colaboran entre sí para llevar a cabo su comportamiento. Finalmente, estos planos lógicos serán convertidos en cosas que pertenecerán al mundo de los bits, tales como ejecutables, bibliotecas, tablas, archivos y documentos. Algunos de estos artefactos tendrán que ser construidos desde cero, pero también se acabará reutilizando algunos artefactos existentes de nuevas formas.

Con UML, los diagramas de artefactos se utilizan para visualizar los aspectos estáticos de estos artefactos físicos y sus relaciones, y para especificar sus detalles para la construcción, como se muestra en la Figura 30.1.

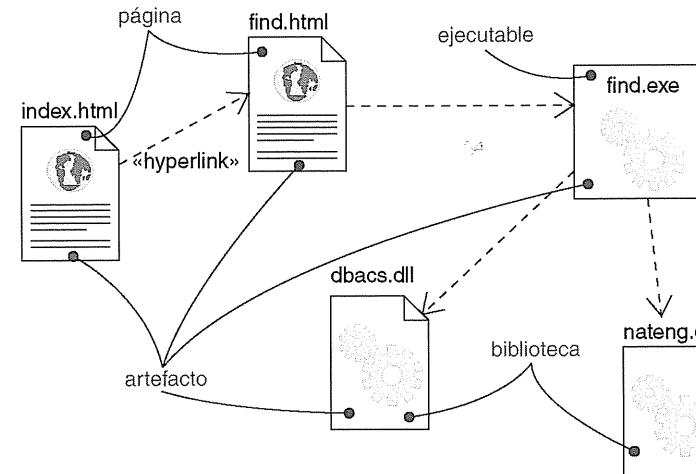


Figura 30.1: Un diagrama de artefactos.

Términos y conceptos

Un *diagrama de artefactos* muestra un conjunto de artefactos y sus relaciones. Gráficamente, un diagrama de artefactos es una colección de nodos y arcos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de artefactos es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de artefactos de los otros tipos de diagramas es su contenido particular.

Contenidos

Los artefactos se discuten en el Capítulo 26; las interfaces se discuten en el Capítulo 11; las relaciones se discuten en los Capítulos 5 y 10; los paquetes se discuten en el Capítulo 12; los sistemas y subsistemas se discuten en el Capítulo 32; las instancias se discuten en el Capítulo 13; los diagramas de clases se discuten en el Capítulo 8; las vistas de implementación, en el contexto de la arquitectura del software, se discuten en el Capítulo 2.

Normalmente, los diagramas de artefactos contienen:

- Artefactos.
- Relaciones de dependencia, generalización, asociación y realización.

Al igual que los demás diagramas, los diagramas de artefactos pueden contener notas y restricciones.

Usos comunes

Los diagramas de artefactos se utilizan para modelar la vista de implementación estática de un sistema. Esta vista se ocupa principalmente de la gestión de configuraciones de las partes de un sistema, formada por artefactos que pueden ensamblarse de varias formas para producir un sistema ejecutable.

Cuando se modela la vista de implementación estática de un sistema, normalmente se utilizarán los diagramas de artefactos de una de las cuatro maneras siguientes.

1. Para modelar código fuente.

Con la mayoría de los lenguajes de programación orientados a objetos actuales, el código se produce utilizando entornos integrados de desarrollo, que almacenan el código fuente en archivos. Los diagramas de artefactos se pueden utilizar para modelar la configuración de estos archivos, los cuales representan los artefactos obtenidos como productos del trabajo, y para configurar el sistema de gestión de configuraciones.

2. Para modelar versiones ejecutables.

Una versión es un conjunto de artefactos relativamente consistente y completo que se entrega a un usuario interno o externo. En el contexto de los artefactos,

una versión se centra en las partes necesarias para entregar un sistema en ejecución. Cuando se modela una versión mediante diagramas de artefactos, se están visualizando, especificando y documentando las decisiones acerca de las partes físicas que constituyen el software (es decir, sus artefactos de despliegue).

La persistencia se discute en el Capítulo 24; el modelado de esquemas lógicos de bases de datos se discute en el Capítulo 8.

3. Para modelar bases de datos físicas.

Una base de datos física puede ser vista como la realización concreta de un esquema, y que pertenece al mundo de los bits. En efecto, los esquemas ofrecen una API para la información persistente; el modelo de una base de datos física representa el almacenamiento de esa información en las tablas de una base de datos relacional o las páginas de una base de datos orientada a objetos. Los diagramas de artefactos se utilizan para representar estos y otros tipos de bases de datos físicas.

4. Para modelar sistemas adaptables

Algunos sistemas son bastante estáticos; sus artefactos entran en escena, participan en la ejecución y desaparecen. Otros sistemas son más dinámicos, e implican agentes móviles o artefactos que migran con el propósito de equilibrar la carga o la recuperación de fallos. Los diagramas de artefactos se utilizan, junto a algunos de los diagramas de UML, para modelar el comportamiento, con el fin de representar a estos tipos de sistemas.

Técnicas comunes de modelado

Modelado de código fuente

Si se desarrolla software en Java, normalmente se guardará el código fuente en archivos .java. Si se desarrolla software con C++, normalmente se guardará el código fuente en archivos de cabecera (archivos .h) y cuerpos (archivos .cpp). Si se utiliza IDL para desarrollar aplicaciones COM+ o CORBA, una interfaz de la vista de diseño se distribuirá en cuatro archivos de código fuente: la propia interfaz, el proxy cliente, el stub servidor y una clase puente. Conforme crezca la aplicación, sin importar el lenguaje que se utilice, se irán organizando estos archivos en grupos más grandes. Además, durante la fase de construcción del desarrollo, probablemente se acabará creando nuevas versiones de algunos de estos archivos por cada nueva versión incremental que se produzca, y será deseable mantener estas versiones bajo el control de un sistema de gestión de configuraciones.

El estereotipo file para los artefactos se discute en el Capítulo 26.

La mayoría de las veces, no será necesario modelar directamente este aspecto de un sistema, sino que se dejará que el entorno de desarrollo siga la pista de estos archivos y sus relaciones. Sin embargo, a veces es útil visualizar estos archivos de código fuente y sus relaciones mediante diagramas de artefactos. Los diagramas de artefactos, cuando se utilizan de esta forma, sólo contienen artefactos que son el producto del trabajo, estereotipados como archivos, junto a sus relaciones de dependencia. Por ejemplo, podría aplicarse ingeniería inversa sobre un conjunto de archivos de código fuente para visualizar la red de dependencias de compilación. También se puede ir en la otra dirección, al especificar las relaciones entre los archivos de código fuente y luego utilizar estos modelos como entrada para las herramientas de compilación, tales como make en Unix. Análogamente, quizás se quieran utilizar los diagramas de artefactos para visualizar la historia de un conjunto de archivos de código fuente que estén bajo una gestión de configuraciones. Al extraer información del sistema de gestión de configuraciones, como el número de veces que un archivo de código fuente ha sido verificado en un período de tiempo, se puede utilizar esa información para colorear los diagramas de artefactos, mostrando los “puntos calientes” del cambio entre los archivos de código fuente y las zonas de mayor movimiento arquitectónico.

Para modelar el código fuente de un sistema:

- Hay que identificar, bien sea a través de ingeniería directa o inversa, el conjunto de archivos de código fuente de interés, y modelarlos como artefactos estereotipados como archivos.
- En los sistemas más grandes, hay que utilizar paquetes para mostrar los grupos de archivos de código fuente.
- Hay que considerar mostrar un valor etiquetado que indique información como el número de versión del archivo de código fuente, su autor y la fecha de la última modificación. Se pueden utilizar herramientas para manejar el valor de esta etiqueta.
- Hay que modelar las dependencias de compilación entre estos archivos mediante dependencias. De nuevo, hay que utilizar las herramientas para ayudar a generar y manejar estas dependencias.

El estereotipo trace para dependencias se discute en el Capítulo 10.

Por ejemplo, la Figura 30.2 muestra cinco archivos de código fuente. signal.h es un archivo de cabecera. Se muestran tres de sus versiones, que se remontan desde las nuevas versiones hacia atrás hasta sus ascendentes más antiguos. Cada variante de este archivo de código fuente se representa con un valor etiquetado que muestra su número de versión.

Este archivo de cabecera (`signal.h`) se utiliza por otros dos archivos (`interp.cpp`) y (`signal.cpp`), los cuales son cuerpos. Uno de estos archivos (`interp.cpp`) tiene una dependencia de compilación con otra cabecera (`irq.h`); a su vez, `device.cpp` tiene una dependencia de compilación con `interp.cpp`. Dado este diagrama de artefactos, es fácil seguir el rastro del impacto de los cambios. Por ejemplo, si se modifica el archivo de código fuente `signal.h`, será necesaria la recompilación de otros tres archivos: `signal.cpp`, `interp.cpp` y, transitivamente, `device.cpp`. Como se muestra en este diagrama, el archivo `irq.h` no se ve afectado.

Los diagramas como éste pueden ser fácilmente generados, mediante ingeniería inversa, a partir de la información contenida en las herramientas de gestión de configuraciones del entorno de desarrollo.

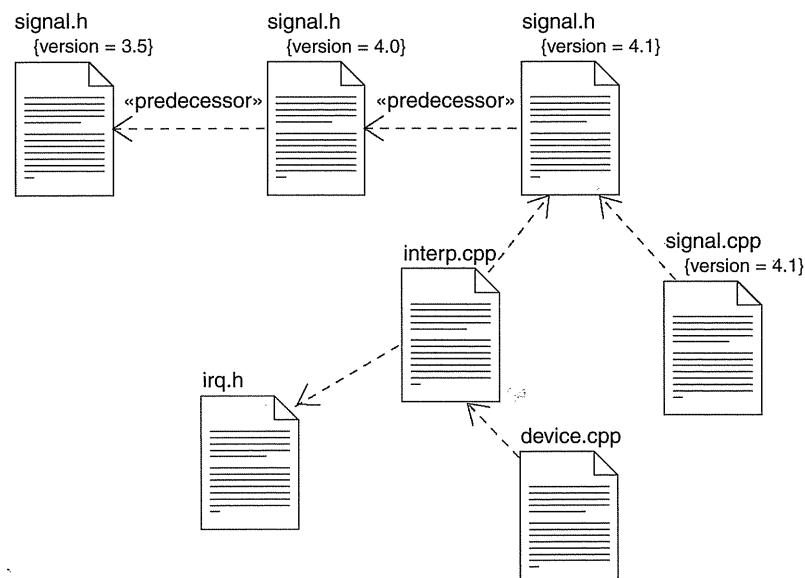


Figura 30.2: Modelado de código fuente.

Modelado de una versión ejecutable

La distribución de una aplicación simple es algo fácil: se copia en un disco un único archivo ejecutable, y los usuarios simplemente ejecutan ese programa. Para estos tipos de aplicaciones, no se necesitan diagramas de artefactos, porque no hay nada difícil que visualizar, especificar, construir o documentar.

La distribución a los usuarios de una nueva versión de cualquier otra cosa que no sea una aplicación simple no es tan fácil. Se necesita el ejecutable principal (normalmente, un archivo .exe), pero también se necesitan sus partes auxiliares, tales como bibliotecas (normalmente archivos .dll si se trabaja en el contexto de COM+, o archivos .class y .jar si se trabaja en el contexto de Java), bases de datos y archivos de ayuda y de recursos. En los sistemas distribuidos, probablemente haya varios ejecutables y otras partes distribuidas entre varios nodos. Si se trabaja con un sistema de aplicaciones, habrá algunos de estos artefactos que serán únicos en cada aplicación, pero otros muchos se compartirán entre varias aplicaciones. Conforme va evolucionando el sistema, el control de la configuración de estos artefactos es una actividad importante a la vez que difícil, porque los cambios en los artefactos asociados con una aplicación pueden afectar al funcionamiento de otras aplicaciones.

Por este motivo, los diagramas de artefactos se utilizan para visualizar, especificar construir y documentar la configuración de las versiones ejecutables, incluyendo los artefactos de despliegue que forman cada versión y las relaciones entre esos artefactos. Los diagramas de artefactos se pueden utilizar para hacer ingeniería directa con un nuevo sistema y para hacer ingeniería inversa con un sistema existente.

Cuando se crean diagramas de artefactos de esta naturaleza, realmente tan sólo se modela una parte de los elementos y relaciones que constituyen la vista de implementación del sistema. Por esta razón, cada diagrama de artefactos debería centrarse en un conjunto de artefactos para ser tratados a un mismo tiempo.

Para modelar una versión ejecutable:

- Hay que identificar el conjunto de artefactos que se pretende modelar. Normalmente, esto implicará a algunos o a todos los artefactos existentes en un nodo, o a la distribución de estos conjuntos de artefactos a través de todos los nodos del sistema.
- Hay que considerar el estereotipo de cada artefacto de este conjunto. Para la mayoría de los sistemas, habrá un pequeño número de diferentes tipos de artefactos (tales como ejecutables, bibliotecas, tablas, archivos y documentos). Los mecanismos de extensibilidad de UML se pueden utilizar para proporcionar señales visuales para estos estereotipos.
- Por cada artefacto de este conjunto, hay que considerar sus relaciones con los vecinos. La mayoría de las veces, esto incluirá las interfaces que son exportadas (realizadas) por ciertos artefactos e importadas (utilizadas) por otros. Si se quiere mostrar las líneas de separación del sistema,

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6; las interfaces se discuten en el Capítulo 11.

hay que modelar explícitamente esas interfaces. Si se quiere modelar a un nivel mayor de abstracción, hay que omitir esas relaciones, mostrando sólo las dependencias entre los artefactos.

Por ejemplo, la Figura 30.3 modela parte de la versión ejecutable de un robot autónomo. Esta figura se centra en los artefactos de despliegue asociados con las funciones de movimiento y cálculo del robot. Se puede ver un artefacto (`motor.dll`) que manifiesta un componente `Dirección`, el cual exporta una interfaz (`IMotor`) que a su vez es utilizada por otro componente `Trayectoria` que es manifestado por otro artefacto (`trayectoria.dll`). La dependencia entre los componentes `Trayectoria` y `Dirección` induce una dependencia entre los artefactos `trayectoria.dll` y `motor.dll` que los implementan. Hay otro artefacto en este diagrama (`colision.dll`) que también manifiesta un componente, aunque los detalles se omiten: `trayectoria.dll` se representa con una dependencia directa hacia `colision.dll`.

Hay muchos más artefactos implicados en este sistema. Sin embargo, este diagrama sólo se centra en aquellos artefactos de despliegue que están directamente implicados en el movimiento del robot. Nótese que en esta arquitectura basada en artefactos se podría sustituir una versión específica de `motor.dll` con otra que manifiestase el mismo componente o uno que manifieste un componente diferente pero que soporte la misma interfaz (y quizás otras adicionales), y `trayectoria.dll` todavía funcionaría correctamente.

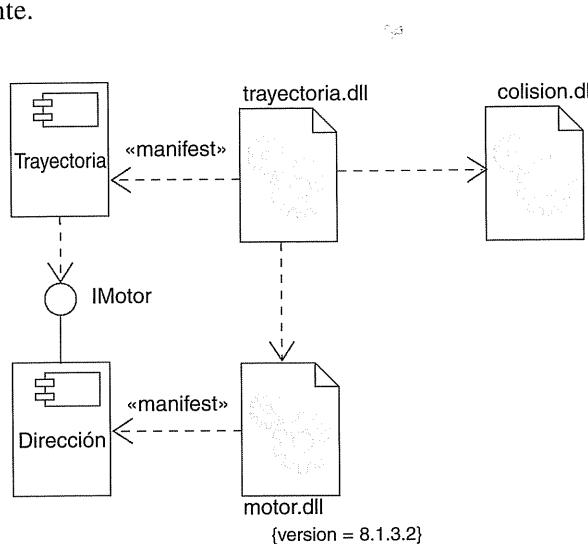


Figura 30.3: Modelado de una versión ejecutable.

Modelado de una base de datos física

El modelado del esquema lógico de una base de datos se discute en el Capítulo 8.

El diseño físico de bases de datos está fuera del alcance de este libro; lo importante aquí es simplemente mostrar cómo se pueden modelar las tablas de la base de datos utilizando UML.

Un esquema lógico de base de datos captura el vocabulario de los datos persistentes de un sistema, junto con la semántica de sus relaciones. Físicamente, estos elementos se almacenan en una base de datos para una recuperación posterior, bien sea en una base de datos relacional, en una orientada a objetos o en una base de datos híbrida objeto-relacional. UML se adapta bien al modelado de bases de datos físicas, así como a los esquemas lógicos de bases de datos.

La correspondencia entre un esquema lógico de base de datos y una base de datos orientada a objetos es bastante directa, porque incluso las jerarquías de herencia complejas se pueden hacer persistentes directamente. Sin embargo, la correspondencia de un esquema lógico de bases de datos con una base de datos relacional no es tan simple. En presencia de la herencia, hay que tomar decisiones acerca de cómo hacer corresponder las clases con tablas. Normalmente, se puede aplicar una de las tres estrategias siguientes, o una combinación de ellas:

1. (Una tabla por clase hoja) Definir una tabla separada por cada clase. Éste es un enfoque sencillo pero ingenuo, porque introduce grandes problemas de mantenimiento cuando se añaden nuevas clases hijas o se modifican clases padres.
2. (Combinación) Definir una única tabla por jerarquía de herencia, de forma que todas las instancias de cualquier clase en una jerarquía tengan el mismo estado. El inconveniente con este enfoque es que se acaba almacenando información superflua en muchas instancias.
3. (División en tablas) Separar el estado relativo a las clases padre e hija en tablas diferentes. Este enfoque se corresponde mejor con la jerarquía de herencia, pero el inconveniente es que la navegación por los datos requiere muchas operaciones de unión (*joins*) de tablas.

Cuando se diseña una base de datos física, también hay que tomar decisiones acerca de cómo asociar las operaciones definidas en el esquema lógico de la base de datos. Las bases de datos orientadas a objetos hacen que esta asociación sea bastante transparente. Pero con las bases de datos relacionales hay que tomar algunas decisiones acerca de cómo se implementarán estas operaciones lógicas. De nuevo, existen varias alternativas:

1. Las simples operaciones ABMC (Altas, Bajas, Modificaciones, Consultas) se pueden implementar con llamadas estándar SQL u ODBC.
2. Los comportamientos más complejos (como las reglas del negocio) se pueden asociar a disparadores o procedimientos almacenados.

Dadas estas líneas generales, para modelar una base de datos física:

- Hay que identificar las clases del modelo que representan el esquema lógico de la base de datos.
- Hay que seleccionar una estrategia para hacer corresponder estas clases con tablas. También habrá que considerar la distribución física de las bases de datos. La estrategia de correspondencia se verá afectada por la localización en la que se quiere que se encuentren los datos en el sistema desplegado.
- Para visualizar, especificar, construir y documentar la correspondencia, hay que crear un diagrama de artefactos que contenga artefactos estereotipados como tablas.
- Donde sea posible, hay que utilizar herramientas que ayuden a transformar el diseño lógico en un diseño físico.

La Figura 30.4 muestra un conjunto de tablas de una base de datos, extraídas del sistema de información de una universidad. Se puede ver una base de datos (*universidad.db*, que se representa como un artefacto estereotipado como *database*) que se compone de cinco tablas: *estudiante*, *clase*, *profesor*, *departamento* y *curso* (representadas como artefactos estereotipados como *table*, uno de los elementos estándar de UML). En el correspondiente esquema lógico de base de datos no había herencia, así que la correspondencia con este diseño de base de datos física es bastante sencilla.

Aunque en la figura no se muestra, se puede especificar el contenido de cada tabla. Los artefactos pueden tener atributos, así que una construcción bastante común cuando se modelan bases de datos físicas es utilizar estos atributos para especificar las columnas de cada tabla. Análogamente, los artefactos pueden tener operaciones, y éstas pueden utilizarse para denotar procedimientos almacenados.

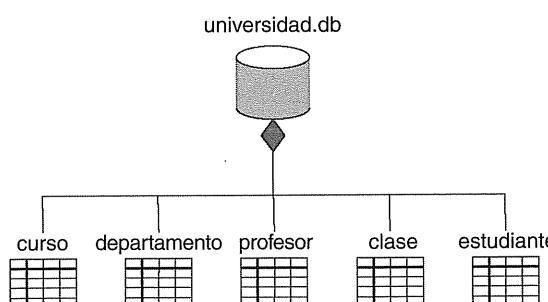


Figura 30.4: Modelado de una base de datos física.

Modelado de sistemas adaptables

Todos los diagramas de artefactos mostrados hasta ahora han sido utilizados para modelar vistas estáticas. Sus artefactos pasan su vida completa en un nodo. Ésta es la situación más común que suele aparecer, pero especialmente en el dominio de los sistemas distribuidos y complejos es necesario modelar vistas dinámicas. Por ejemplo, podría tenerse un sistema que hiciera copias de sus bases de datos a través de varios nodos, cambiando la base de datos principal cuando se produzca una caída del servidor. Análogamente, si se está modelando un funcionamiento globalmente distribuido del tipo 24x7 (es decir, un sistema que funciona 24 horas al día, 7 días a la semana), probablemente se encontrarán agentes móviles, artefactos que migran de un nodo a otro para llevar a cabo alguna transacción. Para modelar estas vistas dinámicas, será necesario utilizar una combinación de diagramas de artefactos, diagramas de objetos y diagramas de interacción.

Para modelar un sistema adaptable:

- Hay que considerar la distribución física de los artefactos que pueden migrar de un nodo a otro. La localización de la instancia de un artefacto se puede especificar marcándola con un valor etiquetado de localización, que se puede representar en un diagrama de artefactos.
- Si se quiere modelar las acciones que hacen que migre un artefacto, hay que crear el correspondiente diagrama de interacción que contenga instancias de artefactos. Un cambio de localización se puede describir dibujando la misma instancia más de una vez, pero con diferentes valores para su estado, que incluye la localización.

Por ejemplo, la Figura 30.5 modela la replicación de la base de datos de la figura anterior. Se muestran dos instancias del artefacto *universidad.db*. Ambas instancias son anónimas, y ambas tienen valores diferentes en su valor etiquetado de localización. También hay una nota, que especifica explícitamente qué instancia es una réplica de la otra.

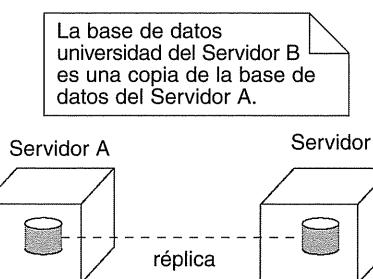


Figura 30.5: Modelado de sistemas adaptables.

El atributo location se discute en el Capítulo 24; los diagramas de objetos se discuten en el Capítulo 14.

Si se quieren mostrar los detalles de cada base de datos, se pueden representar en su forma canónica (un artefacto estereotipado como **database**).

Los diagramas de interacción se discuten en el Capítulo 19.

Aunque no se representa aquí, se puede utilizar un diagrama de interacción para modelar el cambio de la base de datos principal a otra distinta.

Ingeniería directa e inversa

Hacer ingeniería directa e inversa con artefactos es bastante inmediato, ya que los artefactos son en sí mismos cosas físicas (ejecutables, bibliotecas, tablas, archivos y documentos) y, por lo tanto, son cosas cercanas al sistema en ejecución. Al hacer ingeniería directa con una clase o una colaboración, realmente se hace ingeniería directa hacia un artefacto que representa el código fuente, una biblioteca binaria o un ejecutable para esa clase o colaboración. Asimismo, cuando se hace ingeniería inversa a partir de código fuente, bibliotecas binarias o ejecutables, realmente se hace ingeniería inversa hacia un artefacto o un conjunto de artefactos que, a su vez, se corresponden con clases o colaboraciones.

Una decisión que debe tomarse es la elección acerca de si al aplicar ingeniería directa (creación de código a partir de un modelo) a una clase o a una colaboración, se obtendrá código fuente, una biblioteca en formato binario o un ejecutable. Los modelos lógicos se convertirán en código si se tiene interés en controlar la gestión de configuraciones de los archivos que son, posteriormente, manipulados por un entorno de desarrollo. Los modelos lógicos se convertirán directamente en bibliotecas binarias o ejecutables si se tiene interés por gestionar los artefactos que se desplegarán en un sistema ejecutable. En algunos casos, se decidirá hacer ambas cosas. Una clase o una colaboración puede venir dada por código fuente, así como por una biblioteca binaria o un ejecutable.

Para hacer ingeniería directa con un diagrama de artefactos:

- Hay que identificar las clases o colaboraciones que implementa cada artefacto. Esto debe mostrarse con una relación de manifestación.
- Hay que elegir el formato para cada artefacto. La elección se hará básicamente entre código fuente (una forma manipulable por herramientas de desarrollo) o una biblioteca binaria o un ejecutable (una forma que puede introducirse en un sistema ejecutable).

Los diagramas de clases obtenidos por ingeniería inversa se discuten en el Capítulo 8.

- Hay que utilizar herramientas para aplicar ingeniería directa a los modelos.

Construir un diagrama de artefactos mediante ingeniería inversa (creación de un modelo a partir de código) no es un proceso perfecto, ya que siempre hay una pérdida de información. A partir del código fuente, se pueden obtener las clases; esto es lo que se hace con más frecuencia. Cuando se haga ingeniería inversa desde el código hacia los artefactos se revelarán dependencias de compilación entre esos archivos. Para las bibliotecas ejecutables, lo más que se puede esperar es representar la biblioteca como un artefacto y descubrir sus interfaces a través de ingeniería inversa. Éste es el segundo uso más frecuente que se hace de los diagramas de artefactos. De hecho, ésta es una forma útil de abordar un conjunto de nuevas bibliotecas que de otra forma estarían pobremente documentadas. Para los ejecutables, lo mejor que se puede esperar obtener es la representación del ejecutable como un artefacto y luego desensamblar su código (algo que rara vez será necesario, a menos que se trabaje en lenguaje ensamblador).

Para hacer ingeniería inversa con un diagrama de artefactos:

- Hay que elegir el código sobre el que se quiere aplicar ingeniería inversa. A partir del código fuente pueden obtenerse los artefactos y las clases. A partir de las bibliotecas binarias se pueden descubrir sus interfaces. Los ejecutables son los menos apropiados para aplicar ingeniería inversa.
- Mediante una herramienta, hay que señalar el código al que se aplicará la ingeniería inversa. Se debe usar la herramienta para generar un nuevo modelo o modificar uno existente que se haya obtenido con ingeniería directa.
- Mediante la herramienta, hay que crear un diagrama de artefactos inspeccionando el modelo. Por ejemplo, podría comenzarse con uno o más artefactos, y después expandir el diagrama siguiendo las relaciones o los artefactos vecinos. Hay que mostrar u ocultar los detalles del contenido de este diagrama de artefactos, según sea necesario para comunicar su propósito.

Por ejemplo, la Figura 30.6 proporciona un diagrama de artefactos que representa la aplicación de ingeniería inversa sobre el artefacto ActiveX `vbrun.dll`. Como se muestra en la figura, el artefacto realiza once interfaces. Dado este diagrama, se puede comenzar a comprender la semántica del artefacto explorando, a continuación, los detalles de sus interfaces.

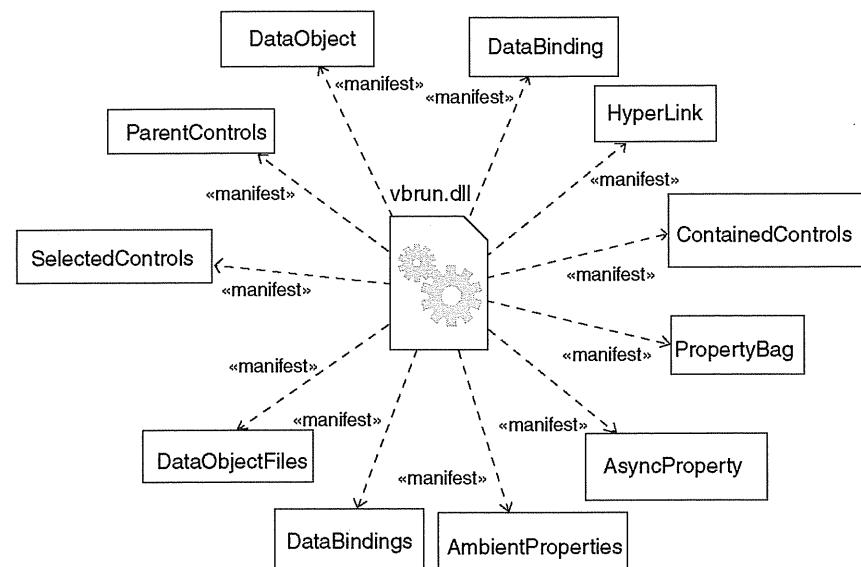


Figura 30.6: Ingeniería inversa.

Especialmente cuando se hace ingeniería inversa a partir de código fuente, y a veces cuando se hace ingeniería inversa a partir de bibliotecas binarias y de ejecutables, esto se hará en el contexto de un sistema de gestión de configuraciones. Esto significa que a menudo se estará trabajando con versiones específicas de archivos o bibliotecas, siendo todas las versiones de una configuración compatibles con las otras. En esos casos, se incluirá un valor etiquetado que represente la versión del artefacto, que puede obtenerse a partir del sistema de gestión de configuraciones. De esta forma, UML se puede utilizar para visualizar la historia de un artefacto a través de varias versiones.

Sugerencias y consejos

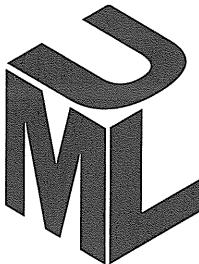
Cuando se crean diagramas de artefactos con UML, debe recordarse que cada diagrama de artefactos es sólo una presentación de la vista de implementación estática de un sistema. Esto significa que un único diagrama de artefactos no necesita capturar todo sobre la vista de implementación de un sistema. En su conjunto, todos los diagramas de artefactos de un sistema representan una vista de implementación estática del sistema que es completa; individualmente, cada uno representa un aspecto.

Un diagrama de artefactos bien estructurado:

- Se utiliza para comunicar un aspecto de la vista de implementación estática de un sistema.
- Contiene sólo aquellos elementos que son esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con el nivel de abstracción, mostrando sólo aquellos adornos que son esenciales para su comprensión.
- No es tan minimalista que deje de ofrecer al lector información sobre los aspectos importantes de la semántica.

Cuando se dibuja un diagrama de artefactos:

- Hay que darle un nombre que comunique su propósito.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que organizar sus elementos espacialmente, de modo que los que estén cercanos semánticamente también lo estén físicamente.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre las características importantes del diagrama.
- Hay que usar los elementos estereotipados con cuidado. Debe elegirse un pequeño conjunto de iconos comunes para el proyecto o la empresa y utilizarlos de forma consistente.



LENGUAJE
UNIFICADO DE
MODELADO

Capítulo 31

DIAGRAMAS DE DESPLIEGUE

En este capítulo

- Modelado de un sistema embebido
- Modelado de un sistema cliente/servidor.
- Modelado de un sistema totalmente distribuido.
- Ingeniería directa e inversa.

Los diagramas de artefactos, el otro tipo de diagramas que se utilizan para modelar los aspectos físicos de un sistema orientado a objetos, se discute en el Capítulo 31.

Los diagramas de despliegue son uno de los dos tipos de diagramas que aparecen cuando se modelan los aspectos físicos de los sistemas orientados a objetos. Un diagrama de despliegue muestra la configuración de los nodos que participan en la ejecución y de los artefactos que residen en ellos.

Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema. La mayoría de las veces, esto implica modelar la topología del hardware sobre el que se ejecuta el sistema. Los diagramas de despliegue son fundamentalmente diagramas de clases que se ocupan de modelar los nodos de un sistema.

Los diagramas de despliegue no sólo son importantes para visualizar, especificar y documentar sistemas embebidos, sistemas cliente/servidor y sistemas distribuidos, sino también para gestionar sistemas ejecutables mediante ingeniería directa e inversa.

Introducción

Cuando se construye un sistema con gran cantidad de software, la atención principal del desarrollador se centra en diseñar y desplegar el software. Sin embargo, para un ingeniero de sistemas, la atención principal está en el hardware y el software del sistema y en el manejo del equilibrio entre ambos. Mientras que los desarrolladores de software trabajan con artefactos en cierto modo intangi-

bles, como modelos y código, los desarrolladores de sistemas trabajan con un hardware bastante tangible.

UML se centra principalmente en ofrecer facilidades para visualizar, especificar, construir y documentar artefactos software, pero también ha sido diseñado para cubrir los artefactos hardware. Esto no equivale a decir que UML sea un lenguaje de descripción de hardware de propósito general, como VHDL. Más bien, UML ha sido diseñado para modelar muchos de los aspectos hardware de un sistema a un nivel suficiente para que un ingeniero de software pueda especificar la plataforma sobre la que se ejecutará el software del sistema, y para que un ingeniero de sistemas pueda manejar la frontera entre el hardware y el software del sistema. En UML, los diagramas de clases y los diagramas de artefactos se utilizan para razonar sobre la estructura del software. Los diagramas de secuencia, los diagramas de comunicación, los diagramas de estados y los diagramas de actividades se utilizan para especificar el comportamiento del software. Cuando se trata del hardware y el software del sistema, se utilizan los diagramas de despliegue para razonar sobre la topología de procesadores y dispositivos sobre los que se ejecuta el software.

Con UML, los diagramas de despliegue se utilizan para visualizar los aspectos estáticos de estos nodos físicos y sus relaciones y para especificar sus detalles para la construcción, como se muestra en la Figura 31.1.

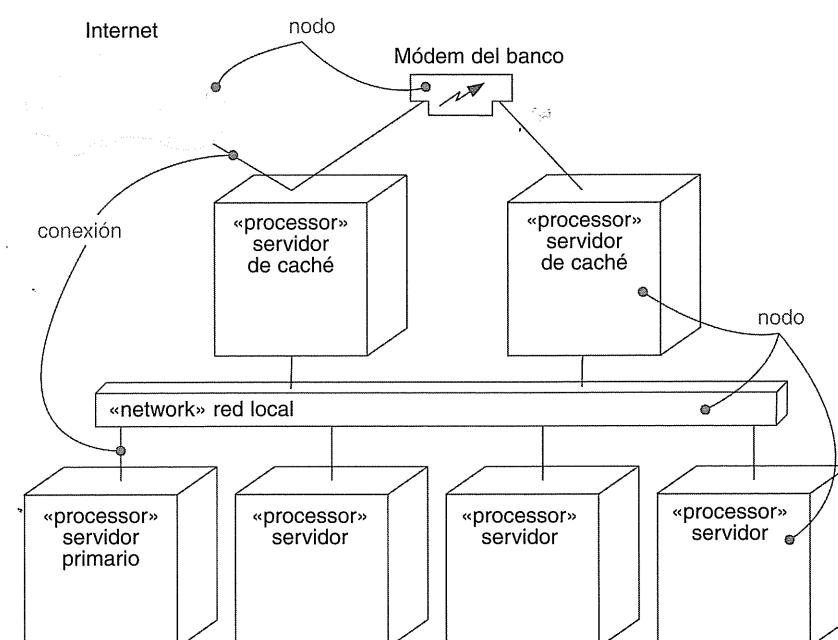


Figura 31.1: Un diagrama de despliegue.

Términos y conceptos

Un *diagrama de despliegue* es un diagrama que muestra la configuración de los nodos de procesamiento y de los artefactos que residen en ellos. Gráficamente, un diagrama de despliegue es una colección de nodos y arcos.

Propiedades comunes

Las propiedades generales de los diagramas se discuten en el Capítulo 7.

Un diagrama de despliegue es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de despliegue de los otros tipos de diagramas es su contenido particular.

Contenidos

Los nodos se discuten en el Capítulo 27; las relaciones se discuten en los Capítulos 5 y 10; los artefactos se discuten en el Capítulo 26; los paquetes se discuten en el Capítulo 12; los subsistemas se discuten en el Capítulo 32; las instancias se discuten en el Capítulo 13; los diagramas de clases se discuten en el Capítulo 8.

Normalmente, los diagramas de despliegue contienen:

- Nodos.
- Relaciones de dependencia y asociación.

Al igual que los demás diagramas, los diagramas de despliegue pueden contener notas y restricciones.

Los diagramas de despliegue también pueden contener artefactos, cada uno de los cuales debe residir en algún nodo. Los diagramas de despliegue también pueden contener paquetes o subsistemas, los cuales se utilizan para agrupar elementos del modelo en bloques más grandes. A veces también se colocarán instancias en los diagramas de despliegue, especialmente cuando se quiera visualizar una instancia de una familia de topologías hardware.

Nota: En muchos sentidos, un diagrama de despliegue es un tipo especial de diagrama de clases que se ocupa de modelar los nodos de un sistema.

Usos comunes

Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema. Esta vista abarca principalmente la distribución, la entrega y la instalación de las partes que configuran el sistema físico.

Las vistas de despliegue en el contexto de la arquitectura del software se discuten en el Capítulo 2.

Hay varios tipos de sistemas para los que son innecesarios los diagramas de despliegue. Si se desarrolla un software que reside en una máquina e interactúa sólo con dispositivos estándar en esa máquina, que ya son gestionados por el sistema operativo (por ejemplo, el teclado, la pantalla y el módem de un computador personal), se pueden ignorar los diagramas de despliegue. Por otro lado, si se desarrolla un software que interactúa con dispositivos que normalmente no gestiona el sistema operativo o si el sistema está distribuido físicamente sobre varios procesadores, entonces la utilización de los diagramas de despliegue puede ayudar a razonar sobre la relación entre el software y el hardware del sistema.

Cuando se modela la vista de despliegue estática de un sistema, normalmente se utilizan los diagramas de despliegue de una de las tres siguientes maneras.

1. Para modelar sistemas embebidos.

Un sistema embebido es una colección de hardware con gran cantidad de software que interactúa con el mundo físico. Los sistemas embebidos involucran software que controla dispositivos como motores, actuadores y pantallas y que, a su vez, están controlados por estímulos externos tales como entradas de sensores, movimientos y cambios de temperatura. Los diagramas de despliegue se pueden utilizar para modelar los dispositivos y los procesadores que comprenden un sistema embebido.

2. Para modelar sistemas cliente/servidor.

Un sistema cliente/servidor es una arquitectura muy extendida que se basa en hacer una clara separación de intereses entre la interfaz de usuario del sistema (que reside en el cliente) y los datos persistentes del sistema (que residen en el servidor). Los sistemas cliente/servidor son un extremo del espectro de los sistemas distribuidos y requieren tomar decisiones sobre la conectividad de red de los clientes a los servidores y sobre la distribución física de los artefactos software del sistema a través de los nodos. La topología de tales sistemas se puede modelar mediante diagramas de despliegue.

3. Para modelar sistemas completamente distribuidos.

En el otro extremo del espectro de los sistemas distribuidos se encuentran aquellos que son ampliamente, si no totalmente, distribuidos y que, normalmente, incluyen varios niveles de servidores. Tales sistemas contienen a menudo varias versiones de los artefactos software, algunos de los cuales pueden incluso migrar de un nodo a otro. El diseño de tales sistemas requiere tomar decisiones que

Las colaboraciones se discuten en el Capítulo 28.

permitan un cambio continuo de la topología del sistema. Los diagramas de despliegue se pueden utilizar para visualizar la topología actual del sistema y la distribución de artefactos, para razonar sobre el impacto de los cambios en esa topología.

Técnicas comunes de modelado

Modelado de un sistema embebido

Los nodos y los dispositivos se discuten en el Capítulo 27.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

El desarrollo de un sistema embebido es mucho más que un problema software. Hay que manejar el mundo físico, en el cual existen partes móviles que pueden fallar y en las que las señales tienen ruido y el comportamiento no es lineal. Cuando se modela un sistema así, hay que tener en cuenta su interfaz con el mundo real, y esto significa razonar acerca de dispositivos no usuales, así como nodos.

Los diagramas de despliegue son útiles para facilitar la comunicación entre los ingenieros de hardware del proyecto y los desarrolladores de software. Cuando se utilizan nodos que se han estereotipado para que tengan el aspecto de dispositivos familiares, se pueden crear diagramas comprensibles para ambos grupos de personas. Los diagramas de despliegue también son útiles para razonar acerca de los compromisos entre el hardware y el software. Los diagramas de despliegue se utilizan para visualizar, especificar, construir y documentar las decisiones de ingeniería del sistema.

Para modelar un sistema embebido:

- Hay que identificar los dispositivos y nodos propios del sistema.
- Hay que proporcionar señales visuales, especialmente para los dispositivos poco usuales, mediante los mecanismos de extensibilidad de UML, definiendo estereotipos específicos del sistema con los iconos apropiados. Como mínimo, habrá que distinguir los procesadores (que contienen artefactos software) y los dispositivos (los cuales no contienen software escrito por nosotros, a ese nivel de abstracción).
- Hay que modelar las relaciones entre esos procesadores y dispositivos en un diagrama de despliegue. Análogamente, hay que especificar la relación entre los artefactos en la vista de implementación del sistema y los nodos en la vista de despliegue del sistema.
- Si es necesario, hay que detallar cualquier dispositivo inteligente, modelando su estructura con un diagrama de despliegue más pormenorizado.

Por ejemplo, la Figura 31.2 muestra el hardware de un sencillo robot autónomo. Se puede ver un nodo (Placa base Pentium) estereotipado como un procesador. Rodeando a este nodo hay ocho dispositivos, cada uno estereotipado como un dispositivo y representado con un icono que ofrece una señal visual clara de su equivalente en el mundo real.

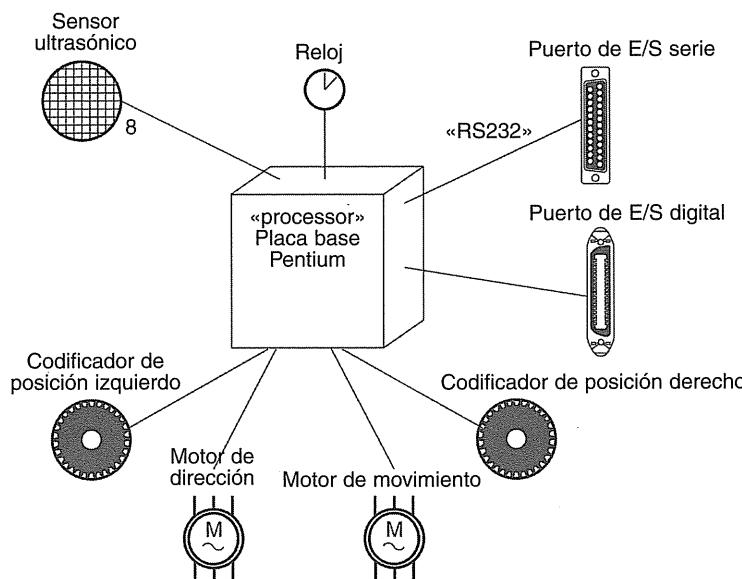


Figura 31.2: Modelado de un sistema embebido.

Modelado de un sistema cliente/servidor

Desde el momento en que se empieza a desarrollar un sistema cuyo software no se encuentra en un único procesador, hay que enfrentarse a un montón de decisiones: ¿Cuál es la mejor forma de distribuir los artefactos software a través de los nodos? ¿Cómo se comunican los distintos artefactos? ¿Cómo se tratan las averías y el ruido? En un extremo del amplio espectro de los sistemas distribuidos se encuentran los sistemas cliente/servidor, en los cuales hay una clara separación de intereses entre la interfaz de usuario del sistema (normalmente manejada por el cliente) y los datos (normalmente manejados por el servidor).

Hay muchas variaciones sobre este tema. Por ejemplo, puede decidirse tener un cliente *ligero*, lo que significa que tiene poca capacidad de procesamiento y hace poco más que manejar la interfaz de usuario y la visualización de la información.

Los clientes ligeros ni siquiera pueden albergar muchos artefactos, pero pueden diseñarse para que carguen artefactos del servidor, en función de las necesidades, como ocurre con los Enterprise Java Beans. Por otro lado, puede decidirse tener un cliente *pesado*, lo que significa que tiene una gran capacidad de procesamiento y hace algo más que la simple visualización. Un cliente pesado normalmente se encarga de algunas de las reglas del negocio y de la lógica del sistema. La elección entre clientes ligeros y pesados es una decisión arquitectónica sobre la que influyen varios factores de naturaleza técnica, económica y organizativa.

En cualquier caso, la división de un sistema en sus partes cliente y servidor implica tomar algunas decisiones difíciles acerca de dónde colocar físicamente sus artefactos software y cómo imponer una distribución equilibrada de responsabilidades entre ellos. Por ejemplo, la mayoría de los sistemas de información de gestión tienen básicamente arquitecturas de tres capas, lo que significa que la interfaz gráfica de usuario del sistema, la lógica del negocio y la base de datos están distribuidas físicamente. Las decisiones acerca de dónde colocar la interfaz gráfica de usuario del sistema y la base de datos son normalmente bastante obvias, de forma que la parte difícil consiste en decidir dónde se ubica la lógica del negocio.

Los diagramas de despliegue de UML se pueden utilizar para visualizar, especificar y documentar las decisiones sobre la topología del sistema cliente/servidor y sobre cómo se distribuyen los artefactos software entre el cliente y el servidor. Normalmente, es deseable crear un diagrama de despliegue para el sistema global, junto con otros diagramas más detallados que profundicen en partes individuales del sistema.

Para modelar un sistema cliente/servidor:

- Hay que identificar los nodos que representan los procesadores cliente y servidor del sistema.
- Hay que destacar aquellos dispositivos relacionados con el comportamiento del sistema. Por ejemplo, es deseable modelar los dispositivos especiales tales como lectores de tarjetas de crédito, lectores de códigos de barras y otros dispositivos de visualización de información distintos de los monitores, porque es probable que su localización en la topología hardware del sistema sea importante desde el punto de vista de la arquitectura.
- Hay que proporcionar señales visuales para esos procesadores y dispositivos a través de los estereotipos.

- Hay que modelar la topología de esos nodos en un diagrama de despliegue. Análogamente, hay que especificar la relación entre los artefactos de la vista de implementación del sistema y los nodos de la vista de despliegue.

Los paquetes se discuten en el Capítulo 12; la multiplicidad se discute en el Capítulo 10.

Por ejemplo, la Figura 31.3 muestra la topología de un sistema de recursos humanos, que sigue una arquitectura clásica cliente/servidor. Esta figura describe explícitamente la división cliente/servidor mediante los paquetes denominados **clientes** y **servidores**. El paquete **clientes** contiene dos nodos (**consola** y **terminal**), ambos estereotipados y distinguibles visualmente. El paquete **servidores** contiene dos tipos de nodos (**servidor de caché** y **servidor**), y ambos han sido adornados con algunos de los artefactos que residen en ellos. También puede notarse que **servidor de caché** y **servidor** han sido marcados con multiplicidades explícitas, que especifican cuántas instancias de cada uno se esperan en una configuración de despliegue particular. Por ejemplo, este diagrama indica que podría haber dos o más **servidores de caché** en cualquier instancia desplegada del sistema.

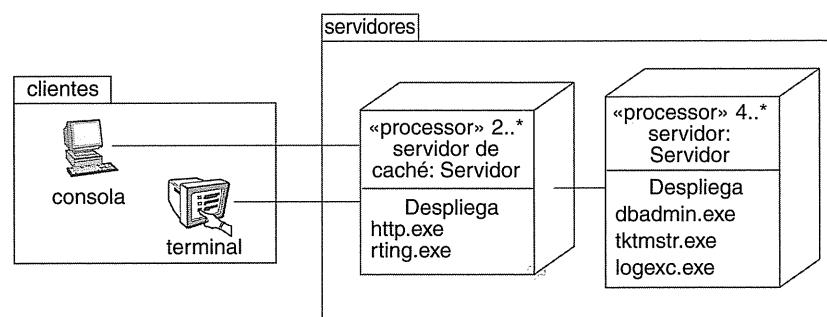


Figura 31.3: Modelado de un sistema cliente/servidor.

Modelado de un sistema completamente distribuido

Los sistemas distribuidos pueden tener muchas formas, desde los simples sistemas con dos procesadores hasta aquellos que comprenden muchos nodos distribuidos geográficamente. Estos últimos no suelen ser estáticos. Los nodos se añaden y se eliminan conforme cambia el tráfico en la red y se producen fallos en los procesadores; se pueden establecer nuevos caminos de comunicación, más rápidos, en paralelo con canales más lentos y antiguos que son puestos finalmente fuera de servicio. No sólo puede cambiar la topología de estos sistemas, sino también la distribución del software. Por ejemplo, puede que las

tablas de una base de datos sean replicadas en varios servidores, sólo para ser movidas en el caso de que sea necesario, dependiendo del tráfico. En algunos sistemas distribuidos por todo el mundo, los artefactos pueden seguir al sol, migrando de servidor en servidor conforme va comenzando cada jornada laboral en unas partes del mundo y va terminando en otras.

La visualización, especificación y documentación de la topología de los sistemas completamente distribuidos de esta naturaleza son actividades valiosas para el administrador del sistema, que debe mantener control sobre todos los elementos computacionales de una empresa. Los diagramas de despliegue de UML se pueden utilizar para razonar acerca de la topología de tales sistemas. Cuando se documentan sistemas completamente distribuidos mediante diagramas de despliegue, se muestran los detalles de los dispositivos de red del sistema, cada uno de los cuales se puede representar como un nodo estereotipado.

Para modelar un sistema completamente distribuido:

- Hay que identificar los dispositivos y los procesadores del sistema igual que para los sistemas cliente/servidor más simples.
- Si es necesario razonar acerca del rendimiento de la red del sistema o del impacto de los cambios en la red, hay que asegurarse de modelar los dispositivos de comunicación al nivel de detalle suficiente para hacer esos razonamientos.
- Hay que prestar una atención especial a las agrupaciones lógicas de nodos, que pueden especificarse mediante paquetes.
- Hay que modelar los dispositivos y procesadores mediante diagramas de despliegue. Donde sea posible, se deben utilizar herramientas que descubran la topología del sistema moviéndose a través de la red.
- Si es necesario centrarse en la dinámica del sistema, hay que introducir diagramas de casos de uso para especificar los tipos de comportamientos que sean de interés, y se deben extender esos casos de uso con diagramas de interacción.

Los paquetes se discuten en el Capítulo 12.

Los casos de uso se discuten en el Capítulo 17; los diagramas de interacción se discuten en el Capítulo 19; las instancias se discuten en el Capítulo 13.

Nota: Cuando se modela un sistema completamente distribuido, es frecuente representar la propia red como un nodo. Por ejemplo, Internet puede ser representada como un nodo (como en la Figura 31.1, en la que se muestra como un nodo estereotipado). También se puede representar una red de área local (*Local Area Network*, LAN) o una red de área extendida (*Wide Area Network*, WAN) de la misma forma (como se muestra en la Figura 31.1). En cada caso, se pueden utilizar los atributos y las operaciones del nodo para capturar propiedades acerca de la red.

La Figura 31.4 muestra la topología de una configuración específica de un sistema completamente distribuido. En particular, este diagrama de despliegue es también un diagrama de objetos, ya que sólo contiene instancias. Se pueden ver tres consolas (instancias anónimas del nodo estereotipado **consola**), las cuales están conectadas a Internet (obviamente, un nodo único). A su vez, hay tres instancias de **servidores regionales**, las cuales sirven como intermediarios para el acceso a los **servidores nacionales**, de los cuales sólo se muestra uno. Como indica la nota, los servidores nacionales están conectados entre sí, pero sus relaciones no se muestran en el diagrama.

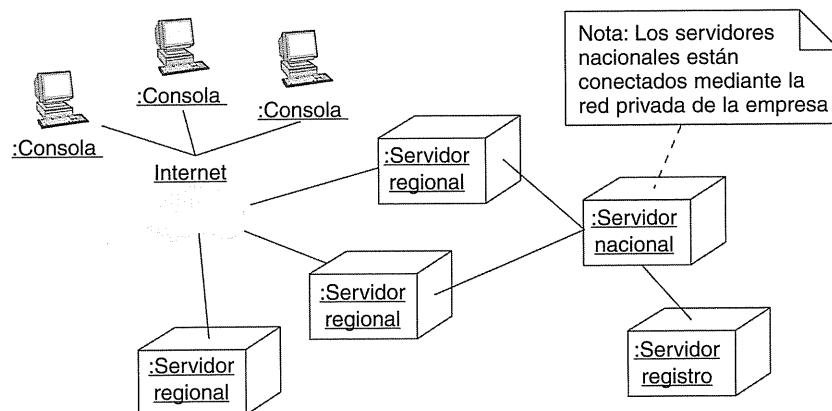


Figura 31.4: Modelado de un sistema completamente distribuido.

En este diagrama, Internet se ha representado como un nodo estereotipado.

Ingeniería directa e inversa

Se puede hacer muy poca ingeniería directa (creación de código a partir de modelos) con los diagramas de despliegue. Por ejemplo, después de especificar la distribución física de artefactos a través de los nodos en un diagrama de despliegue, es posible utilizar herramientas que lleven a estos artefactos al mundo real. Para los administradores de sistemas, la utilización de UML de esta forma los ayuda a visualizar lo que puede ser una tarea muy complicada.

La ingeniería inversa (creación de modelos a partir de código) desde el mundo real hacia los diagramas de despliegue es de enorme valor, especialmente en los sistemas completamente distribuidos que están sujetos a un cambio constante. Se puede proporcionar un conjunto de nodos estereotipados de acuerdo con el vocabulario de los administradores de red del sistema, con el fin de adaptar

UML a su dominio particular. La ventaja de utilizar UML es que ofrece un lenguaje estándar que cubre no sólo sus necesidades, sino también las necesidades de los desarrolladores de software del proyecto.

Para obtener un diagrama de despliegue mediante ingeniería inversa:

- Hay que elegir la parte sobre la que se quiere aplicar la ingeniería inversa. En algunos casos, se explorará la red completa; en otros, se puede limitar la búsqueda.
- Hay que elegir también la fidelidad de la ingeniería inversa. En algunos casos, es suficiente con modelar hasta el nivel de todos los procesadores del sistema; en otros, también se modelarán los periféricos de red.
- Hay que utilizar una herramienta que se mueva a través del sistema, descubriendo la topología hardware. Esa topología debe registrarse en un modelo de despliegue.
- Al mismo tiempo, hay que utilizar herramientas similares a la anterior para descubrir los artefactos existentes en cada nodo, que también se pueden registrar en un modelo de despliegue. Habrá que hacer una búsqueda inteligente, ya que incluso un simple computador personal puede contener gigabytes de artefactos, muchos de los cuales no serán relevantes para el sistema.
- Mediante las herramientas de modelado, hay que crear un diagrama de despliegue consultando el modelo. Por ejemplo, podría comenzarse por visualizar la topología cliente/servidor básica, y después extender el diagrama poblando ciertos nodos con artefactos de interés que residen en ellos. Hay que mostrar u ocultar los detalles del contenido de este diagrama de despliegue, según sea necesario para comunicar su propósito.

Sugerencias y consejos

Cuando se crean diagramas de despliegue en UML, debe recordarse que cada diagrama de artefactos es sólo una presentación de la vista de despliegue estática de un sistema. Esto significa que un único diagrama de despliegue no necesita capturar todo sobre la vista de despliegue de un sistema. En su conjunto, todos los diagramas de despliegue de un sistema representan completamente la vista de despliegue estática del sistema; individualmente, cada uno representa un aspecto.

Un diagrama de despliegue bien estructurado:

- Se ocupa de modelar un aspecto de la vista de despliegue estática de un sistema.
- Contiene sólo aquellos elementos que son esenciales para comprender ese aspecto.
- Proporciona detalles de forma consistente con el nivel de abstracción, mostrando sólo aquellos adornos que son esenciales para su comprensión.
- No es tan minimalista que no ofrezca información al lector sobre los aspectos importantes de la semántica.

Cuando se dibuje un diagrama de despliegue:

- Hay que darle un nombre que comunique su propósito.
- Hay que distribuir sus elementos para minimizar los cruces de líneas.
- Hay que organizar sus elementos espacialmente para que los que estén cercanos semánticamente también lo estén físicamente.
- Hay que usar notas y colores como señales visuales para llamar la atención sobre las características importantes del diagrama.
- Hay que usar los elementos estereotipados con cuidado. Hay que elegir un pequeño conjunto de iconos comunes para el proyecto o la empresa y utilizarlos de forma consistente.



Capítulo 32

SISTEMAS Y MODELOS

En este capítulo

- Sistemas, subsistemas, modelos y vistas.
- Modelado de la arquitectura de un sistema.
- Modelado de sistemas de sistemas.
- Organización de los artefactos del desarrollo.

UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software. UML se utiliza para modelar sistemas. Un modelo es una simplificación de la realidad (una abstracción de un sistema) creado para comprender mejor el sistema. Un sistema, posiblemente descompuesto en una colección de subsistemas, es un conjunto de elementos organizados para lograr un propósito y descrito por un conjunto de modelos, quizás desde diferentes puntos de vista. Cosas tales como las clases, las interfaces, los componentes y los nodos son partes importantes del modelo de un sistema. En UML, los modelos se utilizan para organizar estas y todas las demás abstracciones de un sistema. Conforme se pasa a dominios más complejos, resulta que un sistema a un nivel de abstracción dado aparece como un subsistema a otro nivel mayor. En UML, se pueden modelar sistemas y subsistemas, lo que permite pasar sin saltos bruscos de un nivel a otro más alto.

Los modelos bien estructurados ayudan a visualizar, especificar, construir y documentar un sistema complejo desde diferentes aspectos, aunque relacionados. Los sistemas bien estructurados son cohesivos funcional, lógica y físicamente, y están construidos a partir de subsistemas débilmente acoplados.

Introducción

No hay que pensar demasiado para construir una caseta de un perro. Las necesidades del perro son simples, así que para satisfacer a cualquier perro, excepto a los más exigentes, simplemente hay que hacerla.

Las diferencias entre construir una caseta para el perro y construir un edificio se discuten en el Capítulo 1.

La construcción de una casa o un rascacielos es algo que hay que pensar mucho más. Las necesidades de una familia o los inquilinos de un edificio no son tan simples, así que, incluso para satisfacer a los clientes menos exigentes, no se puede comenzar a construir la casa sin más. En vez de ello, se debe hacer algo de modelado. Los diferentes usuarios mirarán el problema desde diferentes puntos de vista y con diferentes intereses. Por eso, en los edificios complejos, hay que terminar haciendo planos de planta, planos de alzado, planos de calefacción/refrigeración, planos eléctricos, planos de cañerías y quizás incluso planos de redes. No existe un único modelo que pueda capturar de forma adecuada todos los aspectos interesantes de un edificio complejo.

Los diagramas se discuten en el Capítulo 7; las cinco vistas de una arquitectura software se discuten en el Capítulo 2.

En UML, todas las abstracciones de un sistema con gran cantidad de software se organizan en modelos, cada uno de los cuales representa un aspecto relativamente independiente, aunque importante, del sistema que se está desarrollando. Los diagramas se utilizan para visualizar colecciones interesantes de estas abstracciones. La consideración de las cinco vistas de una arquitectura es una forma particularmente útil de canalizar la atención de los diferentes interesados en un sistema software. En conjunto, estos modelos colaboran para proporcionar una descripción completa de la estructura y comportamiento de un sistema.

En los sistemas más grandes, los elementos pueden descomponerse de forma significativa en subsistemas separados, cada uno de los cuales aparece como un sistema más pequeño cuando se considera desde un nivel de abstracción más bajo.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6; los paquetes se discuten en el Capítulo 12.

UML proporciona una representación gráfica para los sistemas y los subsistemas, como se muestra en la Figura 32.1. Esta notación permite visualizar la descomposición de un sistema en subsistemas más pequeños. Gráficamente, un sistema y un subsistema se representan con el símbolo de un componente estereotipado. Los modelos y las vistas tienen una representación gráfica especial (aparte de representarlos como paquetes estereotipados), pero no se suele utilizar, ya que son cosas manipuladas por las herramientas que se utilizan para organizar los diferentes aspectos de un sistema.

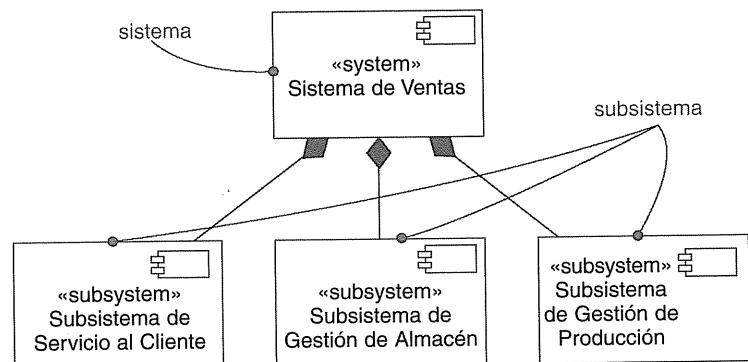


Figura 32.1: Sistemas y subsistemas.

Términos y conceptos

Un *sistema*, posiblemente descompuesto en una colección de subsistemas, es un conjunto de elementos organizados para lograr un propósito, y está descrito por un conjunto de modelos, quizás desde diferentes puntos de vista. Un *subsistema* es una agrupación de elementos, algunos de los cuales constituyen una especificación del comportamiento ofrecido por otros elementos contenidos. Gráficamente, un sistema y un subsistema se representan con el símbolo de un componente estereotipado. Un *modelo* es una simplificación de la realidad, una abstracción de un sistema, creado para comprender mejor el sistema. Una *vista* es una proyección de un modelo, que se ve desde una perspectiva o punto de vista y omite entidades que no son relevantes desde esta perspectiva.

Sistemas y subsistemas

Los estereotíos se discuten en el Capítulo 6; los paquetes se discuten en el Capítulo 12; las clases se discuten en los Capítulos 4 y 9; los casos de uso se discuten en el Capítulo 17; las máquinas de estados se discuten en el Capítulo 22; las colaboraciones se discuten en el Capítulo 28.

Un sistema es aquello que se está desarrollando y para lo cual se construyen los modelos. Un sistema comprende todos los artefactos que constituyen aquello que se está desarrollando, incluyendo todos los modelos y elementos de modelado, tales como clases, interfaces, componentes, nodos y sus relaciones. Todo lo que se necesita para visualizar, especificar, construir y documentar un sistema forma parte de ese sistema, y todo lo que no hace falta para visualizar, especificar, construir y documentar un sistema se encuentra fuera de él.

En UML, un sistema se representa como un componente estereotipado, como se muestra en la figura anterior. Como componente estereotipado, un sistema contiene elementos. Si se mira dentro de un sistema, se verán todos sus modelos y los elementos individuales de modelado (incluyendo los diagramas), que quizás se descompongan en subsistemas. Como clasificador, un sistema puede tener instancias (un sistema puede desplegarse en múltiples instancias en el mundo real), atributos y operaciones (los actores externos al sistema pueden actuar sobre él como un todo), casos de uso, máquinas de estados y colaboraciones, los cuales pueden especificar el comportamiento del sistema. Un sistema puede incluso realizar interfaces, lo que es importante cuando se construyen sistemas de sistemas.

Un subsistema es simplemente una parte de un sistema, y se utiliza para descomponer un sistema complejo en partes casi independientes. Un sistema a un nivel dado de abstracción puede ser un subsistema de otro sistema a un mayor nivel de abstracción.

La agregación y la generalización se discuten en los Capítulos 5 y 10.

La relación principal entre los sistemas y los subsistemas es la composición. Un sistema (el todo) puede contener cero o más subsistemas (las partes). También puede haber relaciones de generalización entre sistemas y subsistemas. Mediante la generalización se pueden modelar familias de subsistemas, algunas de las cuales representan categorías generales de sistemas y otras representan adaptaciones específicas de esos sistemas. Los subsistemas pueden tener varias conexiones entre sí.

Nota: Un sistema representa el elemento de más alto nivel en un contexto dado; los subsistemas que configuran un sistema proporcionan una partición completa y sin solapamientos del sistema global. Un sistema es un subsistema al nivel superior.

Modelos y vistas

Un modelo es una simplificación de la realidad, siendo ésta definida en el contexto del sistema que se está modelando. En definitiva, un modelo es una abstracción de un sistema. Un subsistema representa una partición de los elementos de un sistema más grande en partes independientes; un modelo es una partición de las abstracciones que visualizan, especifican, construyen y documentan ese sistema. La diferencia es sutil pero importante. Un sistema se descompone en subsistemas para que se puedan desarrollar y desplegar esas partes de manera casi independiente; las abstracciones de un sistema o subsistema se descomponen en modelos para que se puedan comprender mejor las cosas que se están desarrollando y desplegando. Así como un sistema complejo, tal como un avión, puede tener muchas partes (por ejemplo, los subsistemas de estructura, propulsión, aerodinámica y alojamiento de pasajeros), esos subsistemas y el sistema global pueden modelarse desde puntos de vista muy diferentes (por ejemplo, las perspectivas de los modelos estructural, dinámico, eléctrico y de calefacción/refrigeración).

Los paquetes se discuten en el Capítulo 12.

Un modelo contiene un conjunto de paquetes. Sin embargo, casi nunca es necesario modelar de forma explícita los modelos. Sin embargo, como las herramientas necesitan manipular modelos, ellas utilizarán normalmente la notación de paquetes para representar un modelo.

Las cinco vistas de una arquitectura software se discuten en el Capítulo 2.

Un modelo contiene paquetes, que a su vez contienen elementos. Los modelos asociados con un sistema o un subsistema partitionan completamente los elementos de ese sistema o subsistema, lo que significa que cada elemento pertenece exactamente a un paquete. Normalmente, los artefactos de un sistema o un subsistema se organizarán en un conjunto de modelos no solapados, de

acuerdo con las cinco vistas de la arquitectura software descritas en otro capítulo.

Los diagramas se discuten en el Capítulo 7.

Un modelo (por ejemplo, un modelo de procesos) puede llegar a contener tantos artefactos (como clases activas, relaciones e interacciones) que en el caso de un sistema grande sería imposible incluir todos esos artefactos a la vez. Se puede pensar en una vista como en una proyección de un modelo. Por cada modelo se tendrán varios diagramas que permitirán observar los elementos contenidos en el modelo. Una vista incluye un subconjunto de las cosas contenidas en un modelo; una vista normalmente no puede cruzar los límites de un modelo. Como se describe en la siguiente sección, no hay relaciones directas entre los modelos, aunque pueden aparecer relaciones de traza entre elementos de diferentes modelos.

Nota: UML no especifica qué modelos se deberían utilizar para visualizar, especificar, construir y documentar un sistema, aunque el Proceso Unificado de Rational sugiere un conjunto de modelos que han demostrado su validez.

Traza

Las relaciones se discuten en los Capítulos 5 y 10.

La especificación de las relaciones entre elementos tales como clases, interfaces, componentes y nodos es una parte estructural importante de cualquier modelo. La especificación de las relaciones entre elementos tales como documentos, diagramas y paquetes de diferentes modelos es una parte importante de la gestión de los artefactos del desarrollo de sistemas complejos, muchos de los cuales pueden existir en diferentes versiones.

Las dependencias se discuten en el Capítulo 5; los estereotipos se discuten en el Capítulo 6.

En UML, las relaciones conceptuales entre elementos de diferentes modelos pueden modelarse mediante una relación de traza; una traza no puede aplicarse entre elementos del mismo modelo. Una traza se representa como una dependencia estereotipada. A menudo se puede ignorar la dirección de esta dependencia, aunque normalmente se dirigirá hacia el elemento más antiguo o más específico, como se muestra en la Figura 32.2. Los dos usos más comunes de la relación de traza son seguir la evolución del sistema desde los requisitos hasta la implementación (y todos los artefactos en medio) y seguir la evolución de una versión a otra.

Nota: La mayoría de las veces no se deseará representar las relaciones de traza de forma explícita, sino, más bien, tratarlas como hipervínculos.

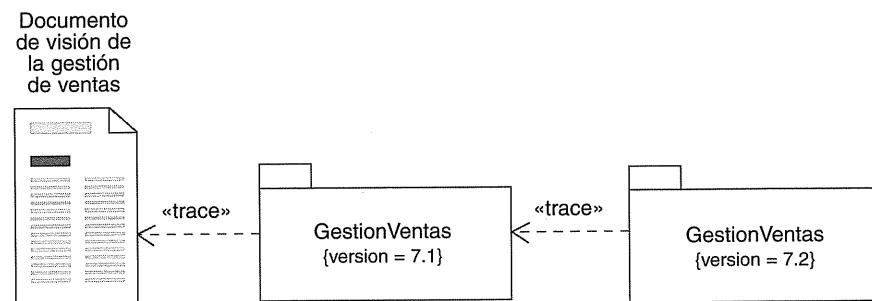


Figura 32.2. Relaciones de traza.

Técnicas comunes de modelado

Modelado de la arquitectura de un sistema

La arquitectura y el modelado se discuten en el Capítulo 1.

La mayoría de las veces, los sistemas y los modelos se utilizarán para organizar los elementos que se emplean para visualizar, especificar, construir y documentar la arquitectura de un sistema. En última instancia, esto afecta prácticamente a todos los artefactos que aparecen en un proyecto de desarrollo de software. Cuando se modela la arquitectura de un sistema, se capturan las decisiones sobre los requisitos del sistema, sus elementos lógicos y sus elementos físicos. También se modelarán los aspectos estructurales y de comportamiento de los sistemas, y los patrones que configuran estas vistas. Por último, habrá que centrarse en las líneas de separación entre subsistemas y en el seguimiento de la evolución del sistema desde los requisitos hasta su despliegue.

Para modelar la arquitectura de un sistema:

- Hay que identificar las vistas que se usarán para representar la arquitectura. La mayoría de las veces, se incluirán una vista de casos de uso, una vista de diseño, una vista de interacción, una vista de implementación y una vista de despliegue, como se muestra en la Figura 32.3.
- Hay que especificar el contexto del sistema, incluyendo los actores que lo rodean.

Si es necesario, hay que descomponer el sistema en sus subsistemas elementales.

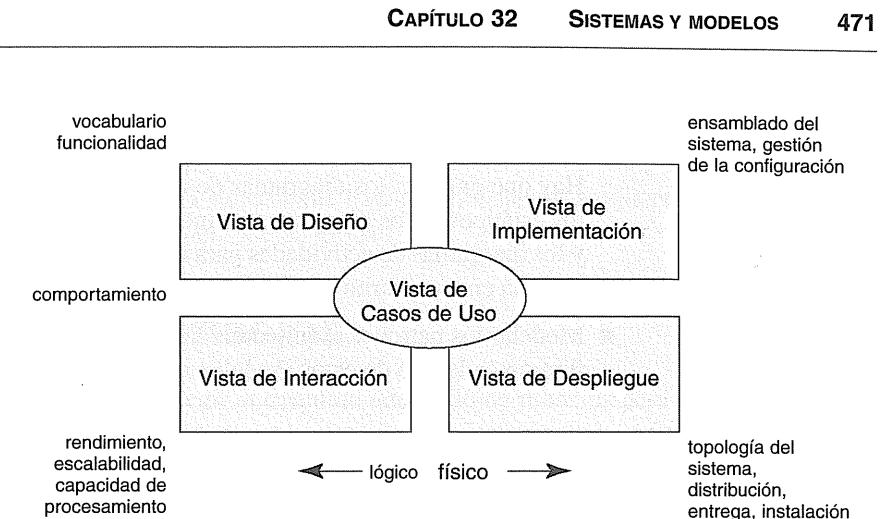


Figura 32.3: Modelado de la arquitectura de un sistema.

Las siguientes actividades se aplican a los sistemas, así como a sus subsistemas:

- Especificar una vista de casos de uso del sistema, incluyendo los casos de uso que describen el comportamiento del sistema, tal y como será visto por los usuarios finales, analistas y realizadores de pruebas. Hay que emplear los diagramas de casos de uso para modelar los aspectos estáticos, y los diagramas de interacción, los diagramas de estados y los diagramas de actividades para modelar los aspectos dinámicos.
- Especificar una vista de diseño del sistema, incluyendo las clases, interfaces y colaboraciones que forman el vocabulario del sistema y su solución. Hay que emplear los diagramas de clases y de objetos para modelar aspectos estáticos, y los diagramas de interacción, los diagramas de estados y los diagramas de actividades para modelar los aspectos dinámicos.
- Especificar una vista de interacción del sistema, incluyendo los hilos, procesos y mensajes que forman los mecanismos de concurrencia y sincronización del sistema. Hay que aplicar los mismos diagramas que en la vista de diseño, pero prestando especial atención a las clases activas y los objetos que representan hilos y procesos, así como a los mensajes y al flujo de control.
- Especificar una vista de implementación del sistema, incluyendo los artefactos que se utilizan para ensamblar y poner en marcha el sistema físico. Hay que emplear los diagramas de artefactos para modelar los aspectos estáticos, y los diagramas de interacción, los diagramas de estados y los diagramas de actividades para modelar los aspectos dinámicos.

- Especificar una vista de despliegue del sistema, incluyendo los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Hay que emplear los diagramas de despliegue para modelar los aspectos estáticos, y los diagramas de interacción, los diagramas de estados y los diagramas de actividades para modelar los aspectos dinámicos del sistema en su entorno de ejecución.
- Modelar los patrones arquitectónicos y los patrones de diseño que configuran cada uno de estos modelos mediante colaboraciones.

El Proceso Unificado de Rational se discute en el Apéndice B.

Los mecanismos de extensibilidad de UML se discuten en el Capítulo 6.

Es importante darse cuenta de que la arquitectura de un sistema no se crea de una sola vez. Más bien, un proceso bien estructurado para UML implicará el refinamiento sucesivo de la arquitectura de un sistema de un modo que esté dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

Para todos los sistemas, excepto los más triviales, habrá que gestionar distintas versiones de los artefactos del sistema. Los mecanismos de extensibilidad de UML (y en particular los valores etiquetados) se pueden utilizar para capturar las decisiones relacionadas con la versión de cada elemento.

Modelado de sistemas de sistemas

Un sistema a un nivel de abstracción aparecerá como un subsistema a un nivel de abstracción más alto. Del mismo modo, un subsistema a un nivel de abstracción aparecerá como todo un sistema desde la perspectiva del equipo encargado de crearlo.

Todos los sistemas complejos exhiben este tipo de jerarquía. Conforme se pasa a sistemas de complejidad cada vez mayor, se hace necesario descomponer los esfuerzos en subsistemas, cada uno de los cuales puede desarrollarse de forma casi independiente, e ir creciendo iterativa e incrementalmente hasta llegar al sistema global. El desarrollo de un subsistema es prácticamente igual que el desarrollo de un sistema.

Para modelar un sistema o un subsistema:

- Hay que identificar las principales partes funcionales del sistema que pueden desarrollarse, publicarse y desplegarse de forma casi independiente. Cuestiones técnicas, organizativas, legales y los sistemas ya existentes determinarán a menudo cómo se establecen los límites de cada subsistema.
- Para cada subsistema, hay que especificar su contexto, al igual que se hace para el sistema global; los actores en torno a un subsistema

incluyen a los de todos los subsistemas vecinos, así que todos deben diseñarse para colaborar.

- Para cada subsistema, hay que modelar su arquitectura al igual que se hace con el sistema global.

Sugerencias y consejos

Es importante elegir el conjunto adecuado de modelos para visualizar, especificar, construir y documentar un sistema. Un modelo bien estructurado:

- Proporciona una simplificación de la realidad desde un punto de vista bien definido y relativamente independiente.
- Es autocontenido, en el sentido de que no requiere de otros contenidos para comprender su semántica.
- Está débilmente acoplado con otros modelos a través de relaciones de traza.
- En conjunto (con otros modelos vecinos) proporciona una descripción completa de los artefactos de un sistema.

Análogamente, es importante descomponer los sistemas complejos en subsistemas bien estructurados. Un sistema bien estructurado:

- Es cohesivo funcional, lógica y físicamente.
- Puede descomponerse en subsistemas casi independientes, que son sistemas por sí mismos, vistos a un nivel de abstracción más bajo.
- Puede visualizarse, especificarse, estructurarse y documentarse a través de un conjunto de modelos interrelacionados que no se solapan.

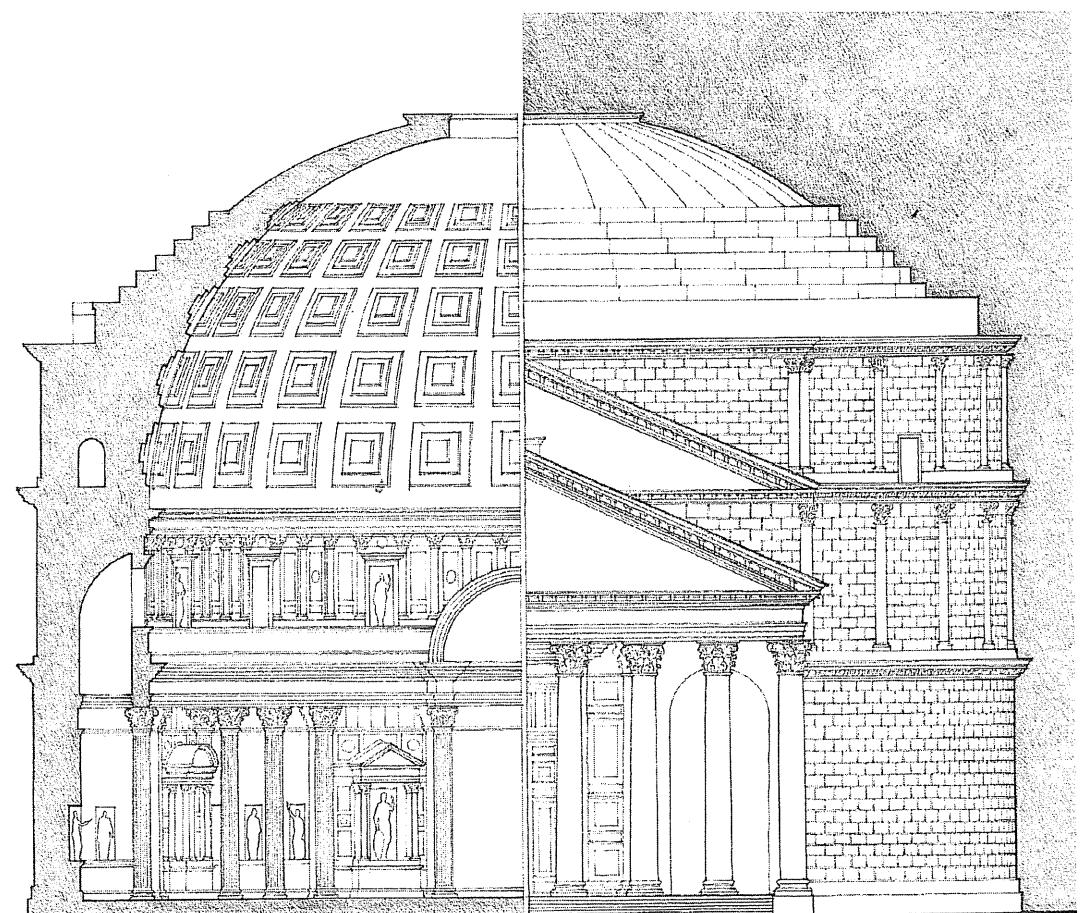
UML tiene un símbolo gráfico para un modelo, pero es mejor evitarlo. Hay que modelar el sistema, y no el propio modelo. Las herramientas de edición deben proporcionar recursos para visualizar, organizar y gestionar conjuntos de modelos.

Cuando se dibuje un sistema o un subsistema en UML:

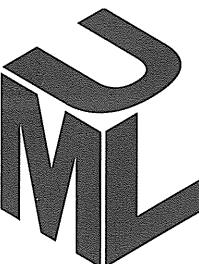
- Hay que utilizarlo como punto de partida de todos los artefactos asociados con él.
- Hay que mostrar sólo la agregación básica entre el sistema y sus subsistemas; normalmente, los detalles de sus conexiones se dejarán para los diagramas de más bajo nivel.



Parte 7
CIERRE



LENGUAJE
UNIFICADO DE
MODELADO



Capítulo 33
UTILIZACIÓN DE UML

En este capítulo

- Transición a UML.
- Adónde ir a continuación.

Los problemas simples son fáciles de modelar con UML. Los problemas difíciles también son fáciles de modelar, especialmente después de haber adquirido soltura con el lenguaje.

Leer sobre cómo utilizar UML es una cosa, pero sólo se puede llegar a dominar el lenguaje *utilizándolo*. Según la formación de cada persona, hay varias formas de afrontar el uso de UML por primera vez. Conforme se gana más experiencia, se llegan a entender y apreciar sus aspectos más sutiles.

Si algo puede pensarse, UML puede modelarlo.

Transición a UML

El 80 por ciento de la mayoría de los problemas puede modelarse con el 20 por ciento de UML. Los elementos estructurales básicos, tales como clases, atributos, operaciones, casos de uso y paquetes, junto a las relaciones estructurales básicas, como la dependencia, la generalización y la asociación, son suficientes para crear modelos estáticos para muchos tipos de dominios de problemas. Si a esta lista se añaden los elementos de comportamiento básicos, tales como las máquinas de estados simples y las interacciones, se pueden modelar muchos aspectos útiles de la dinámica de un sistema. Sólo habrá que utilizar las características más avanzadas de UML cuando se empiece a modelar las cosas que aparecen en situaciones más complejas, como cuando se modela la concurrencia y la distribución.

Un buen punto de partida para comenzar a utilizar UML es modelar algunas de las abstracciones o comportamientos básicos de sistemas existentes. Hay que desarrollar un modelo conceptual de UML, de forma que se disponga de un marco alrededor del cual pueda ampliarse la comprensión del lenguaje. Posteriormente, se irá comprendiendo mejor cómo encajan las partes más avanzadas de UML. Conforme se acometan problemas más complejos, se puede profundizar en algunas características específicas de UML estudiando las técnicas comunes de modelado de este libro.

Si uno es nuevo en el campo de la orientación a objetos:

- Hay que empezar sintiéndose cómodo con la idea de abstracción. Los ejercicios en equipo con tarjetas CRC y el análisis basado en casos de uso son formas excelentes de desarrollar las habilidades de identificación de abstracciones claras.
- Puede modelarse una parte estática sencilla de un problema mediante clases, dependencias, generalizaciones y asociaciones, para familiarizarse con la visualización de sociedades de abstracciones.
- Puede modelarse una parte dinámica de un problema mediante diagramas de secuencia o comunicación. Un buen punto de partida es construir un modelo de la interacción del usuario con el sistema, lo que proporcionará un beneficio inmediato, al ayudar a razonar a través de algunos de los casos de uso más importantes del sistema.

Si uno es nuevo en el modelado:

- Hay que empezar tomando una parte de algún sistema ya construido (preferiblemente implementado en algún lenguaje de programación orientado a objetos, como Java o C++) y construir un modelo UML de sus clases y sus relaciones.
- Utilizando UML, hay que tratar de capturar algunos detalles de construcciones particulares del lenguaje de programación o mecanismos usados en ese sistema, que se tienen en la cabeza pero no pueden expresarse directamente en el código.
- Especialmente si se tiene una aplicación no trivial, hay que tratar de reconstruir un modelo de su arquitectura mediante los componentes (incluyendo los subsistemas) para representar sus principales elementos estructurales. Hay que utilizar los paquetes para organizar el propio modelo.
- Después de haberse familiarizado con el vocabulario de UML y antes de empezar a teclear código en el siguiente proyecto, en primer lugar,

hay que construir un modelo UML de la parte elegida del sistema. Hay que pensar en la estructura o comportamiento que se ha especificado, y sólo entonces, cuando uno esté contento con el tamaño, la forma y la semántica, se puede utilizar ese modelo como un marco para la implementación.

Si ya se tiene experiencia con algún otro método orientado a objetos:

- Hay que tomar el lenguaje de modelado que se está utilizando y construir una correspondencia de sus elementos con los elementos de UML. En la mayoría de los casos se puede encontrar una correspondencia uno a uno y la mayoría de los cambios son puramente cosméticos.
- Puede pensarse en algún problema de modelado complicado que sea difícil o imposible de modelar con el lenguaje actual de modelado. Hay que buscar algunas de las características avanzadas de UML que podrían permitir abordar ese problema con mayor claridad o simplicidad.

Si uno es un usuario avanzado:

- Hay que asegurarse de haber desarrollado, en primer lugar, un modelo conceptual de UML. Puede perderse la armonía de los conceptos si se profundiza en las partes más sofisticadas del lenguaje sin haber comprendido primero su vocabulario más básico.
- Hay que prestar una atención especial a las características de UML que modelan la estructura interna, la colaboración, la concurrencia, la distribución y los patrones (cuestiones que a menudo conllevan una semántica sutil y compleja).
- Hay que considerar también los mecanismos de extensibilidad de UML y ver cómo se puede particularizar UML para incluir directamente el vocabulario del dominio en cuestión. Hay que resistir la tentación de llegar al extremo de producir un modelo UML que no reconocería nadie, excepto otros usuarios avanzados.

Adónde ir a continuación

Este libro es parte de una serie de libros que, en conjunto, pueden ayudar a aprender cómo aplicar UML. Además de este libro, que cumple la función de una guía del usuario de UML, están los siguientes:

- James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*, Addison-Wesley, 2005. Este libro proporciona una referencia global de la sintaxis y la semántica de UML¹.
 - Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*, Addison-Wesley, 1999. Este libro presenta un proceso de desarrollo recomendado para utilizar con UML².

Para aprender más acerca del modelado de los principales creadores de UML, pueden verse las siguientes referencias:

- Michael Blaha, James Rumbaugh, *Object-Oriented Modeling and Design with UML, Second Edition*. Prentice-Hall, 2005.
 - Grady Booch, *Object-Oriented Analysis and Design with Applications, Second Edition*. Addison-Wesley, 1993³.
 - Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

La última información sobre el Proceso Unificado de Rational puede encontrarse en:

- Philippe Kruchten, *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley, 2004.

La última información sobre UML está en el sitio web del OMG en www.omg.org, donde puede encontrarse la última versión del estándar de UML.

Hay muchos otros libros que describen UML y varios métodos de desarrollo, además de la gran cantidad de libros que describen la práctica de la ingeniería del software en general.

¹ N. del T. Existe una versión en español de la primera edición, *El lenguaje unificado de modelado. Manual de referencia*.

² N. del T. Existe una versión en español, *El proceso unificado de desarrollo de software*.

³ N. del T. Existe una versión en español, *Análisis y diseño orientado a objetos con aplicaciones*.



Apéndice A

NOTACIÓN UML

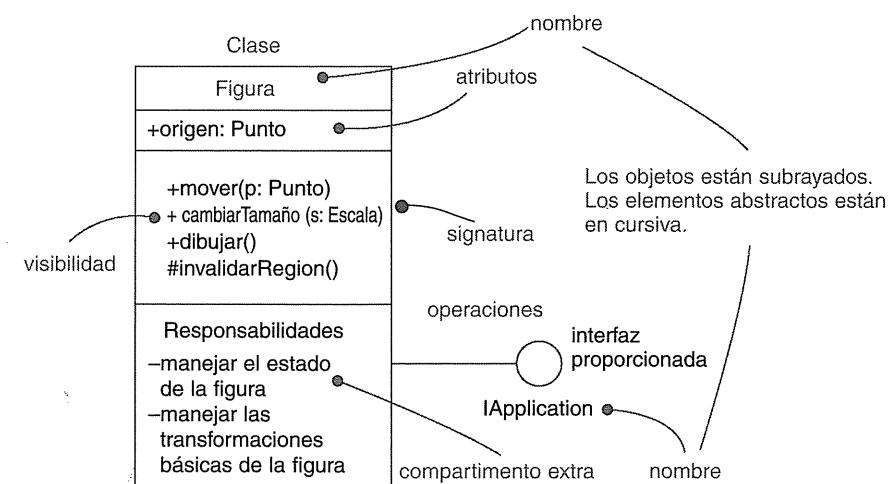
UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software. Como lenguaje, UML tiene una sintaxis y una semántica bien definidas. La parte más visible de la sintaxis de UML es su notación gráfica.

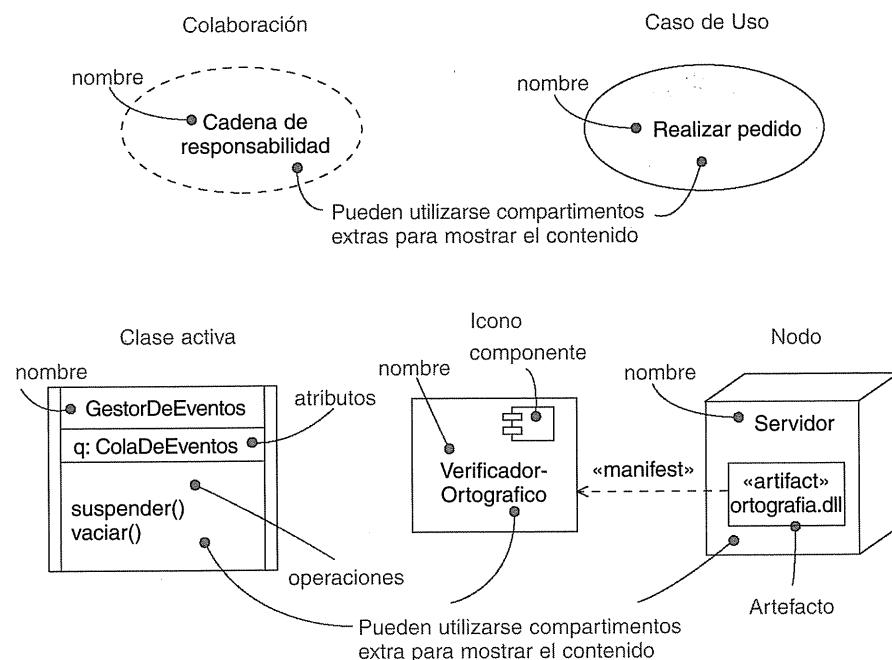
Este apéndice resume los elementos de la notación UML.

Elementos

Elementos estructurales

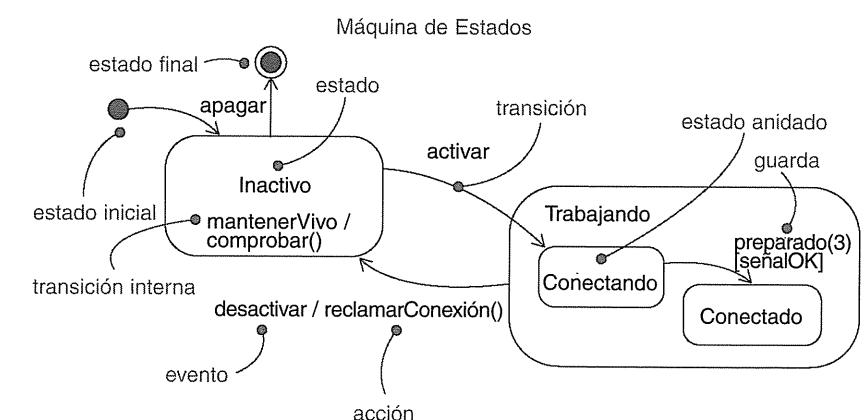
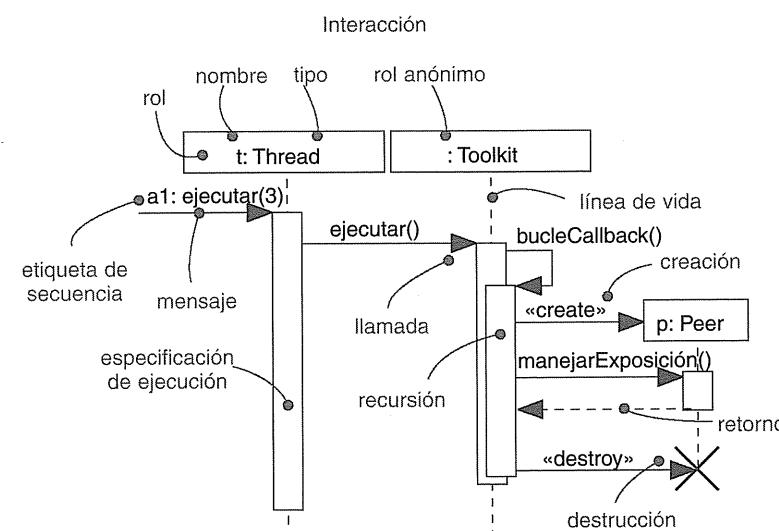
Los elementos estructurales son los nombres de los modelos UML. Éstos incluyen clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos.





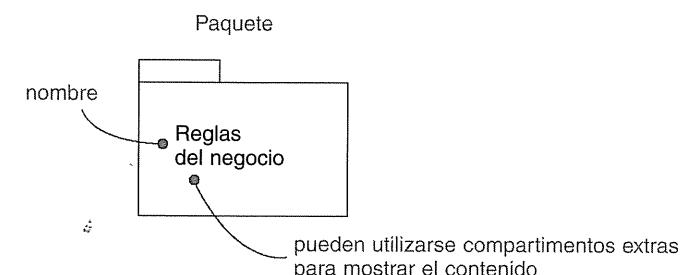
Elementos de comportamiento

Los elementos de comportamiento son las partes dinámicas de los modelos de UML. Éstos incluyen diagramas de interacción y máquinas de estados.



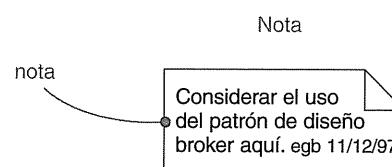
Elementos de agrupación

Los elementos de agrupación son las partes organizativas de los modelos de UML. Éstos incluyen los paquetes.



Elementos de anotación

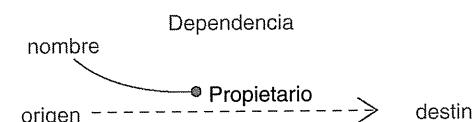
Los elementos de anotación son las partes explicativas de los modelos de UML. Éstos incluyen las notas.



Relaciones

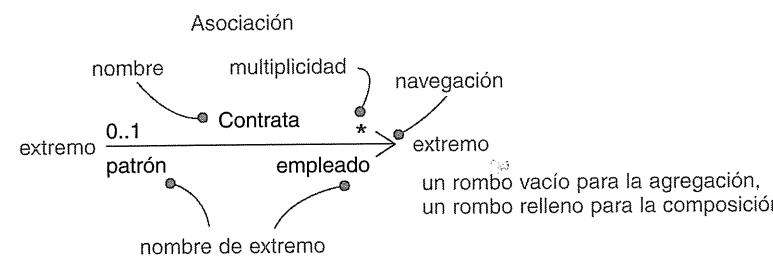
Dependencia

Una dependencia es una relación semántica entre dos elementos, en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (el elemento dependiente).



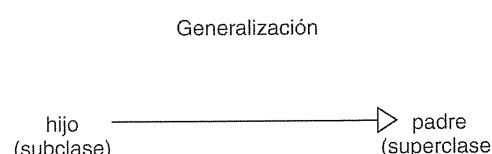
Asociación

Una asociación es una relación estructural que describe un conjunto de enlaces; un enlace es una conexión entre objetos.



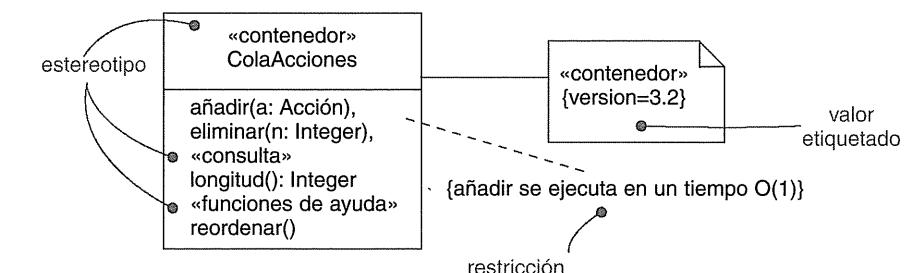
Generalización

La generalización es una relación de especialización/generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre).



Extensibilidad

UML proporciona tres mecanismos para extender la sintaxis y la semántica del lenguaje: estereotipos (que representan nuevos elementos de modelado), valores etiquetados (que representan nuevos atributos de modelado) y restricciones (que representan nueva semántica de modelado).



Diagramas

Un diagrama es la representación gráfica de un conjunto de elementos, representado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Un diagrama es una proyección de un sistema. UML incluye trece de estos diagramas.

1. Diagrama de clases
 2. Diagrama de objetos
 3. Diagrama de componentes
 4. Diagrama de estructura compuesta
- Un diagrama estructural que muestra un conjunto de clases, interfaces, colaboraciones y sus relaciones.
- Un diagrama estructural que muestra un conjunto de objetos y sus relaciones.
- Un diagrama estructural que muestra las interfaces externas, incluyendo los puertos, y la composición interna de un componente.
- Un diagrama estructural que muestra las interfaces externas y la composición interna de una clase estructurada. En este libro, hemos combinado el tratamiento de los diagramas de estructura compuesta con los diagramas de componentes.

5. Diagrama de casos de uso Un diagrama de comportamiento que muestra un conjunto de casos de uso, actores y sus relaciones.
6. Diagrama de secuencia Un diagrama de comportamiento que muestra una interacción, destacando la ordenación temporal de los mensajes.
7. Diagrama de comunicación Un diagrama de comportamiento que muestra una interacción, destacando la organización estructural de los objetos que envían y reciben mensajes.
8. Diagrama de estados Un diagrama de comportamiento que muestra una máquina de estados, destacando el comportamiento dirigido por eventos de un objeto.
9. Diagrama de actividades Un diagrama de comportamiento que muestra un proceso computacional, destacando el flujo de control a través de las actividades.
10. Diagrama de despliegue Un diagrama estructural que muestra las relaciones entre un conjunto de nodos, artefactos, clases manifestadas y componentes. En este libro, también hemos especializado el modelado de artefactos como un diagrama de artefactos.
11. Diagrama de paquetes Un diagrama estructural que muestra la organización del modelo en paquetes.
12. Diagrama de tiempo Un diagrama de comportamiento que muestra una interacción con mensajes en tiempos específicos. Estos diagramas no se abordan en este libro.
13. Diagrama de vista global de interacciones Un diagrama de comportamiento que combina aspectos de los diagramas de actividades y de los diagramas de secuencia. No se abordan en este libro.

También se permiten tipos de diagramas mezclados: no hay una separación estricta entre elementos del modelo.



Apéndice B EL PROCESO UNIFICADO DE RATIONAL

Un proceso es un conjunto de pasos parcialmente ordenados para alcanzar un objetivo. En la ingeniería del software, el objetivo es entregar un producto software que satisfaga las necesidades del negocio, de forma eficiente y predecible.

UML es bastante independiente del proceso, lo que significa que se puede utilizar con diferentes procesos de ingeniería del software. El Proceso Unificado de Rational es uno de esos enfoques de ciclo de vida que se adapta especialmente bien a UML. El objetivo del Proceso Unificado de Rational es permitir la producción de un software de la mayor calidad que satisfaga las necesidades de los usuarios finales, dentro de planificaciones y presupuestos predecibles. El Proceso Unificado de Rational captura algunas de las mejores prácticas de desarrollo de software, de una forma que es adaptable a un amplio rango de proyectos y organizaciones. El Proceso Unificado de Rational proporciona un enfoque disciplinado acerca de cómo asignar tareas y responsabilidades dentro de una organización de desarrollo de software, mientras que al mismo tiempo permite que el equipo se adapte a las siempre cambiantes necesidades del proyecto.

Este apéndice resume los elementos del Proceso Unificado de Rational.

Características del proceso

El Proceso Unificado de Rational es un proceso *iterativo*. Para los sistemas simples, parece perfectamente factible definir de forma secuencial el problema completo, diseñar la solución completa, construir el software y, a continuación, hacer pruebas con el producto final. Sin embargo, dadas la complejidad y sofisticación que demandan los sistemas actuales, este enfoque lineal al desarrollo de sistemas no es realista. Un enfoque iterativo propone una comprensión incremental del problema a través de refinamientos sucesivos y un crecimiento

incremental de una solución efectiva a través de varios ciclos. Como parte del enfoque iterativo se encuentra la flexibilidad para acomodarse a nuevos requisitos o a cambios tácticos en los objetivos del negocio. También permite que el proyecto identifique y resuelva los riesgos lo antes posible.

Las actividades del Proceso Unificado de Rational promueven la creación y el mantenimiento de *modelos* más que documentos sobre papel. Los modelos (especialmente aquellos especificados mediante UML) proporcionan representaciones ricas, desde el punto de vista semántico, del sistema software que se está desarrollando. Estos modelos pueden verse de muchas formas, y la información representada puede ser capturada instantáneamente y controlada electrónicamente. La razón subyacente al interés que pone el Proceso Unificado de Rational en los modelos, antes que en los documentos sobre papel, es minimizar la sobrecarga asociada con la generación y el mantenimiento de los documentos y maximizar el contenido de información relevante.

El desarrollo bajo el Proceso Unificado de Rational está *centrado en la arquitectura*. El proceso se centra en establecer al principio una arquitectura software que guía el desarrollo del sistema. Tener una arquitectura robusta facilita el desarrollo en paralelo, minimiza la repetición de trabajos e incrementa la probabilidad de reutilización de componentes y el mantenimiento posterior del sistema. Este diseño arquitectónico sirve como una sólida base sobre la cual se puede planificar y manejar el desarrollo de software basado en componentes.

Las actividades de desarrollo bajo el Proceso Unificado de Rational están *dirigidas por los casos de uso*. El Proceso Unificado de Rational pone un gran énfasis en la construcción de sistemas basada en una amplia comprensión de cómo se utilizará el sistema que se entregue. Los conceptos de los casos de uso y los escenarios se utilizan para guiar el flujo de procesos desde la captura de los requisitos hasta las pruebas, y para proporcionar caminos que se pueden reproducir durante el desarrollo del sistema.

El Proceso Unificado de Rational soporta las *técnicas orientadas a objetos*. Los modelos del Proceso Unificado de Rational se basan en los conceptos de objeto y clase y las relaciones entre ellos, y utilizan UML como la notación común.

El Proceso Unificado de Rational es un proceso *configurable*. Aunque un único proceso no es adecuado para todas las organizaciones de desarrollo de software, el Proceso Unificado de Rational es adaptable y puede configurarse para cubrir las necesidades de proyectos que van desde pequeños equipos de desarrollo de software hasta grandes empresas de desarrollo. El Proceso Unificado

de Rational se basa en una arquitectura de procesos simple y clara, que proporciona un marco común a toda una familia de procesos y que, además, puede variarse para acomodarse a distintas situaciones. Dentro del propio Proceso Unificado de Rational se encuentran las guías sobre cómo configurar el proceso para adaptarse a las necesidades de una organización.

El Proceso Unificado de Rational promueve un *control de calidad* y una *gestión del riesgo* objetivos y continuos. La evaluación de la calidad va contenida en el proceso, en todas las actividades, e implicando a todos los participantes, mediante medidas y criterios objetivos. La calidad no se trata como algo *a posteriori* o como una actividad separada. La gestión del riesgo va contenida en el proceso, de manera que los riesgos para el éxito del proyecto se identifican y se acometen al principio del proceso de desarrollo, cuando todavía hay tiempo de reaccionar.

Fases e iteraciones

Una *fase* es el intervalo de tiempo entre dos hitos importantes del proceso durante la cual se cumple un conjunto bien definido de objetivos, se completan artefactos y se toman las decisiones sobre si pasar a la siguiente fase. Como ilustra la Figura B.1, el Proceso Unificado de Rational consta de las cuatro fases siguientes:

- | | |
|-----------------|--------------------------------------------------------------------------------------------|
| 1. Concepción | Establecer la visión, el alcance y el plan inicial del proyecto. |
| 2. Elaboración | Diseñar, implementar y probar una arquitectura correcta, y completar el plan del proyecto. |
| 3. Construcción | Construir la primera versión operativa del sistema. |
| 4. Transición | Entregar el sistema a sus usuarios finales. |

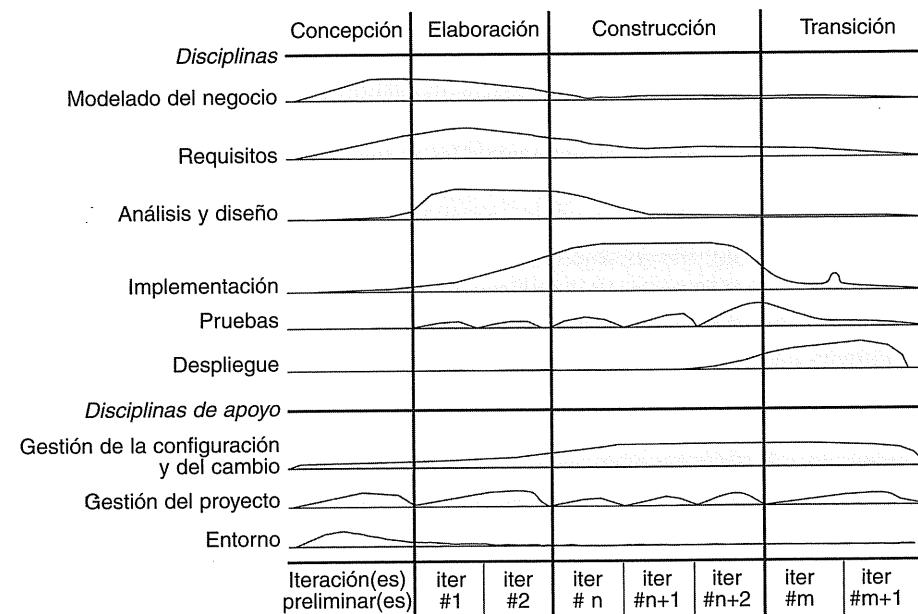


Figura B.1. El ciclo de vida del desarrollo de software.

Las fases de concepción y elaboración se centran más en las actividades creativas e ingenieriles del ciclo de vida del desarrollo, mientras que la construcción y la transición se centran más en las actividades de producción.

Dentro de cada fase hay varias iteraciones. Una *iteración* representa un ciclo de desarrollo completo, desde la captura de requisitos en el análisis hasta la implementación y pruebas, que produce como resultado una versión ejecutable. La versión no tiene por qué incluir un conjunto de características dispuestas para su publicación comercial. Su objetivo es proporcionar una base sólida para la evaluación y las pruebas, así como unos cimientos uniformes para el siguiente ciclo de desarrollo.

Cada fase e iteración se centra en disminuir algún riesgo y concluye con un hito bien definido. La revisión de hitos es el momento adecuado para evaluar cómo se están satisfaciendo los objetivos y si el proyecto necesita ser reestructurado de alguna forma para continuar.

Fases

Concepción. Durante la fase de concepción, se establece la visión del sistema y se delimita el alcance del proyecto. Esto incluye la oportunidad del negocio, los requisitos de alto nivel y el plan inicial del proyecto. El plan del proyecto

incluye los criterios de éxito, la evaluación del riesgo, estimaciones de recursos que se necesitarán y un plan de fases que muestre la planificación de los hitos principales. Durante la concepción, es frecuente crear un prototipo ejecutable que sirva como prueba de los conceptos.

Esta fase suele implicar a un puñado de personas.

Al final de la fase de concepción se examinan los objetivos del ciclo de vida del proyecto y se decide si proceder con el desarrollo del sistema.

Elaboración. Los objetivos de la fase de elaboración son analizar el dominio del problema, establecer una base arquitectónica correcta, desarrollar el plan del proyecto y eliminar los elementos de más alto riesgo del proyecto. Las decisiones arquitectónicas deben tomarse con una comprensión del sistema global. Esto implica que se deben describir la mayoría de los requisitos del sistema. Para verificar la arquitectura, se implementa un sistema que demuestre las distintas posibilidades de la arquitectura y ejecute los casos de uso significativos.

Esta fase implica al arquitecto del sistema y al gestor del proyecto como participantes clave, así como a analistas, desarrolladores, probadores y otros. Normalmente, la elaboración implica a más gente que la concepción, y requiere más tiempo.

Al final de la fase de elaboración se examinan el alcance y los objetivos detallados del sistema, la elección de la arquitectura y la resolución de los riesgos más grandes, y se decide si se debe pasar a la construcción.

Construcción. Durante la fase de construcción, se desarrolla de forma iterativa e incremental un producto completo que está preparado para pasar a la comunidad de usuarios. Esto implica describir los requisitos restantes y los criterios de aceptación, refinando el diseño y completando la implementación y las pruebas del software.

Esta fase involucra al arquitecto del sistema, el gestor del proyecto y a los jefes del equipo de construcción, así como a todo el personal de desarrollo y pruebas.

Al final de la fase de construcción se decide si el software, los lugares donde se instalará y los usuarios están todos preparados para instalar la primera versión operativa del sistema.

Transición. Durante la fase de transición, el software se despliega en la comunidad de usuarios. Hay que tener en cuenta que hemos estado involucrados con

los usuarios a través de todo el proyecto mediante demostraciones, talleres y las versiones preliminares. Una vez que el sistema está en manos de los usuarios finales, a menudo aparecen cuestiones que requieren un desarrollo adicional para ajustar el sistema, corregir algunos problemas no detectados o finalizar algunas características que habían sido pospuestas. Esta fase comienza normalmente con una versión beta del sistema, que luego será reemplazada con el sistema de producción.

Los miembros del equipo clave para esta fase incluyen al gestor del proyecto, los probadores, los especialistas de distribución, y el personal de ventas y comercialización. No debe olvidarse que el trabajo preparativo de la versión externa, de marqueting y ventas comenzó mucho antes en el proyecto.

Al final de la fase de transición se decide si se han satisfecho los objetivos del ciclo de vida del proyecto, y se determina si se debería empezar otro ciclo de desarrollo. Éste es también un punto en el que se asimilan las lecciones aprendidas en el proyecto para mejorar el proceso de desarrollo, que será aplicado al próximo proyecto.

Iteraciones

Cada fase en el Proceso Unificado de Rational puede descomponerse en iteraciones. Una iteración es un ciclo completo de desarrollo que produce una versión (interna o externa) de un producto ejecutable, que constituye un subconjunto del producto final en desarrollo, que luego se irá incrementando de iteración en iteración hasta convertirse en el sistema final. Cada iteración pasa a través de varias disciplinas, aunque con diferente énfasis en cada una de ellas dependiendo de la fase. Durante la concepción, el interés se orienta hacia la captura de requisitos. Durante la elaboración, el interés se dirige al análisis, el diseño y la implementación de la arquitectura. Durante la construcción, las actividades centrales son el diseño detallado, la implementación y las pruebas, y la transición se centra en el despliegue. Las pruebas son importantes a través de todas las fases.

Ciclos de desarrollo

El paso a través de las cuatro fases principales constituye un ciclo de desarrollo, y produce una generación de software. La primera pasada a través de las cuatro fases se denomina ciclo de desarrollo inicial. A menos que acabe la vida del producto, un producto existente evolucionará a la siguiente generación

repitiendo la misma secuencia de concepción, elaboración, construcción y transición. Ésta es la evolución del sistema, así que los ciclos de desarrollo después del ciclo inicial son los ciclos de evolución.

Disciplinas

El Proceso Unificado de Rational consta de nueve disciplinas.

- | | |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| 1. Modelado del negocio | Describe la estructura y la dinámica de la organización del cliente. |
| 2. Requisitos | Extrae los requisitos utilizando diferentes métodos. |
| 3. Análisis y diseño | Describe las diferentes vistas arquitectónicas. |
| 4. Implementación | Tiene en cuenta el desarrollo de software, las pruebas unitarias y la integración. |
| 5. Pruebas | Describe los scripts, la ejecución de pruebas y las métricas para rastreo de defectos. |
| 6. Despliegue | Incluye la facturación de los materiales, notas de edición, formación y otros aspectos relacionados con la entrega de la aplicación. |
| 7. Gestión de configuraciones | Controla los cambios y mantiene la integridad de los artefactos de un proyecto y de las actividades de gestión. |
| 8. Gestión del Proyecto | Describe varias estrategias de trabajo en un proceso iterativo. |
| 9. Entorno | Cubre la infraestructura necesaria para desarrollar un sistema. |

Dentro de cada disciplina hay un conjunto de artefactos y actividades relacionados. Un *artefacto* es algún documento, informe o ejecutable que se produce, se manipula o se consume. Una *actividad* describe las tareas (pasos de concepción, realización y revisión) que llevan a cabo los trabajadores para crear o modificar los artefactos, junto con las técnicas y guías para ejecutar las tareas, incluyendo posiblemente el uso de herramientas para ayudar a automatizar algunas de ellas.

A algunos de estos flujos de trabajo del proceso se les asocian importantes conexiones entre los artefactos. Por ejemplo, el modelo de casos de uso que se genera durante la captura de requisitos *es realizado por* el modelo de diseño de la disciplina de análisis y diseño, *es implementado por* el modelo de implementación de la disciplina de implementación, y *es verificado por* el modelo de pruebas de la disciplina de pruebas.

Artefactos

Cada actividad del Proceso Unificado de Rational lleva algunos artefactos asociados, bien sean requeridos como entradas, bien sean generados como salidas. Algunos artefactos se utilizan como entradas directas en las actividades siguientes, se mantienen como recursos de referencia en el proyecto, o se generan en algún formato específico, en forma de entregables definidos en el contrato.

Modelos

El modelado se discute en el Capítulo 1.

Los modelos son el tipo de artefacto más importante en el Proceso Unificado de Rational. Un modelo es una simplificación de la realidad, creada para comprender mejor el sistema que se está creando. En el Proceso Unificado de Rational, hay varios modelos que en conjunto cubren todas las decisiones importantes implicadas en la visualización, especificación, construcción y documentación de un sistema con gran cantidad de software.

- | | |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| 1. Modelo de casos de uso del negocio | Establece una abstracción de la organización. |
| 2. Modelo de análisis del negocio | Establece el contexto del sistema. |
| 3. Modelo de casos de uso | Establece los requisitos funcionales del sistema. |
| 4. Modelo de análisis (opcional) | Establece un diseño conceptual. |
| 5. Modelo de diseño | Establece el vocabulario del problema y de su solución. |
| 6. Modelo de datos (opcional) | Establece la representación de los datos para las bases de datos y otros repositorios. |
| 7. Modelo de despliegue | Establece la topología hardware sobre la cual se ejecutará el sistema, además de los mecanismos de sincronización y concurrencia. |

La arquitectura se discute en el Capítulo 2.

8. Modelo de implementación

Establece cuáles son las partes que se utilizarán para ensamblar y hacer disponible el sistema físico.

Una vista es una proyección de un modelo. En el Proceso Unificado de Rational, la arquitectura de un sistema se captura en forma de cinco vistas que interactúan entre sí: la vista de diseño, la vista de interacción, la vista de despliegue, la vista de implementación y la vista de casos de uso.

Otros artefactos

Los artefactos del Proceso Unificado de Rational se clasifican en artefactos de gestión y artefactos técnicos. Los artefactos técnicos del Proceso Unificado de Rational pueden dividirse en cinco conjuntos principales.

- | | |
|----------------------------------|-----------------------------------------------------------------------|
| 1. Conjunto de requisitos | Describe qué debe hacer el sistema. |
| 2. Conjunto de análisis y diseño | Describe cómo se va a construir el sistema. |
| 3. Conjunto de pruebas | Describe el método por el que se validará y se verificará el sistema. |
| 4. Conjunto de implementación | Describe el ensamblado de los componentes software desarrollados. |
| 5. Conjunto de despliegue | Proporciona todos los datos para la configuración entregable. |

Conjunto de requisitos. Este conjunto agrupa toda la información que describe lo que debe hacer el sistema. Esto puede comprender un modelo de casos de uso, un modelo de requisitos no funcionales, un modelo del dominio, un modelo de análisis y otras formas de expresión de las necesidades del usuario, incluyendo pero no limitándose a maquetas, prototipos de la interfaz, restricciones legales, etcétera.

Conjunto de diseño. Este conjunto agrupa información que describe cómo se va a construir el sistema y captura las decisiones acerca de cómo se va realizar, teniendo en cuenta las restricciones de tiempo, presupuesto, aplicaciones existentes, reutilización, objetivos de calidad y demás consideraciones. Esto puede implicar un modelo de diseño, un modelo de pruebas y otras formas de expresión de la naturaleza del sistema, incluyendo, pero no limitándose a, prototipos y arquitecturas ejecutables.

Conjunto de pruebas. Este conjunto agrupa información sobre las pruebas del sistema, incluyendo guiones, casos de prueba, métricas para el seguimiento de defectos y criterios de aceptación.

Conjunto de implementación. Este conjunto agrupa toda la información acerca de los elementos software que comprende el sistema, incluyendo, pero no limitándose a, código fuente en diferentes lenguajes de programación, archivos de configuración, archivos de datos, componentes software, etcétera, junto con la información que describe cómo ensamblar el sistema.

Conjunto de despliegue. Este conjunto agrupa toda la información acerca de la forma en que se empaqueta actualmente el software, se distribuye, se instala y se ejecuta en el entorno destino.

GLOSARIO

abstracción características esenciales de una entidad que la distinguen de otros tipos de entidades. Una abstracción define una frontera desde la perspectiva del observador.

acción computación ejecutable que produce un cambio de estado en el sistema o que devuelve un valor.

acción asíncrona solicitud en la que el objeto emisor no espera a que se produzcan los resultados.

actividad comportamiento expresado como un conjunto de acciones conectadas por flujos de datos y de control.

actor conjunto coherente de roles que representan los usuarios de los casos de uso cuando interactúan con éstos.

adorno detalle de la especificación de un elemento que se añade a su notación gráfica básica.

agregación forma especial de asociación que especifica una relación todo-parte entre el agregado (el todo) y un componente (la parte).

agregado clase que representa el “todo” en una relación de agregación.

ámbito contexto que da significado a un nombre.

argumento valor específico que corresponde a un parámetro.

arquitectura conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales (y sus interfaces) de los que se compone el sistema, junto con su comportamiento tal y como se especifica en las colaboraciones entre esos elementos, la composición de esos elementos estructurales y de comportamiento en subsistemas cada vez mayores y el estilo arquitectónico que dirige esta organización (esos elementos y sus interfaces, sus colaboraciones y su composición). La arquitectura del software no sólo tiene que ver con la estructura y el comportamiento, sino también con las restricciones y los

compromisos entre uso, funcionalidad, rendimiento, flexibilidad, reutilización, comprensibilidad, economía y tecnología, y con intereses estéticos.

artefacto pieza discreta de información que es utilizada o producida por un proceso de desarrollo de software o un sistema existente.

asociación relación estructural que describe un conjunto de enlaces, donde un enlace es una conexión entre objetos; relación semántica entre dos o más clasificadores que implica la conexión entre sus instancias.

asociación binaria asociación entre dos clases.

asociación n-aria asociación entre tres o más clases.

atributo propiedad con nombre de un clasificador que describe un rango de valores que pueden contener las instancias de la propiedad.

booleano enumeración cuyos valores son *verdadero* y *falso*.

cadena secuencia de caracteres de texto.

clasificador atributo de una asociación cuyos valores partitionan el conjunto de objetos relacionado con un objeto a través de una asociación.

calle partición sobre un diagrama de actividades para organizar las responsabilidades de las acciones.

característica propiedad, tal como una operación o un atributo, que se encapsula dentro de otra entidad, como una interfaz, una clase o un tipo de datos.

característica de comportamiento característica dinámica de un elemento, tal como una operación.

característica estructural característica estática de un elemento.

cardinalidad número de elementos en un conjunto.

caso de uso descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable, de valor para un actor.

centrado en la arquitectura en el contexto del ciclo de desarrollo de software, proceso que se centra en el desarrollo y establecimiento de una arquitectura software desde el principio, y utiliza la arquitectura del sistema como el principal artefacto para conceptualizar, construir, manejar y hacer evolucionar el sistema en desarrollo.

clase descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.

clase activa clase cuyas instancias son objetos activos.

clase abstracta clase que no puede ser instanciada directamente.

clase asociación elemento de modelado que tiene propiedades tanto de clase como de asociación. Una clase asociación puede verse como una asociación que tiene propiedades de clase, o una clase que tiene propiedades de asociación.

clase concreta clase que puede ser instanciada directamente.

clasificación dinámica variación semántica de la generalización en la que un objeto puede cambiar de tipo o de rol.

clasificación estática variación semántica de la generalización en la que un objeto no puede cambiar de tipo ni de rol.

clasificación múltiple variación semántica de la generalización en la que un objeto puede pertenecer directamente a más de una clase.

clasificador mecanismo que describe características estructurales y de comportamiento. Los clasificadores incluyen a las clases, interfaces, tipos de datos, señales, componentes, nodos, casos de uso y subsistemas.

cliente clasificador que requiere un servicio de otro clasificador.

colaboración sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos; especificación de cómo se realiza un elemento, tal como un caso de uso o una operación, por un conjunto de clasificadores y asociaciones que representan roles específicos y que se utilizan de una forma específica.

comentario anotación asociada a un elemento o a una colección de elementos.

componente parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la realización de dicho conjunto.

comportamiento especificación de una computación ejecutable.

composición forma de agregación con fuerte pertenencia y un tiempo de vida coincidente entre las partes y el todo; las partes con una multiplicidad no fija pueden ser creadas después del propio compuesto, pero una vez creadas viven y mueren con él; tales partes también pueden ser eliminadas explícitamente antes de la eliminación del compuesto.

compuesto clase relacionada con una o más clases a través de una relación de composición.

concepción primera fase del ciclo de vida del desarrollo, en el cual se lleva la idea inicial del desarrollo hasta el punto de estar suficientemente bien fundada para justificar la entrada en la fase de elaboración.

concurrencia ocurrencia de dos o más lugares de ejecución durante el mismo intervalo de tiempo. La concurrencia se puede conseguir entrelazando o ejecutando simultáneamente dos o más hilos.

condición de guarda condición que se debe satisfacer para permitir que se dispare una transición asociada.

construcción tercera fase del ciclo de vida del desarrollo de software, en la cual el software se lleva desde una base arquitectónica ejecutable básica hasta el punto en el que está listo para ser transferido a la comunidad de usuarios.

contenedor objeto que existe para contener otros objetos y que proporciona operaciones para acceder o iterar sobre los elementos que contiene.

contexto conjunto de elementos relacionados para un objetivo particular, como especificar una operación.

delegación capacidad de un objeto de enviar un mensaje a otro objeto en respuesta a un mensaje.

dependencia relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (el elemento dependiente).

diagrama representación gráfica de un conjunto de elementos, representado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones).

diagrama de actividades diagrama que muestra el flujo de control y datos entre actividades. Los diagramas de actividades cubren la vista dinámica de un sistema.

diagrama de casos de uso diagrama que muestra un conjunto de casos de uso y actores y sus relaciones; los diagramas de casos de uso cubren la vista de casos de uso estática de un sistema.

diagrama de clases diagrama que muestra un conjunto de clases, interfaces y colaboraciones y sus relaciones; los diagramas de clases cubren la vista de diseño estática de un sistema; diagrama que muestra una colección de elementos declarativos (estáticos).

diagrama de componentes diagrama que muestra la organización y las dependencias entre un conjunto de componentes; los diagramas de componentes cubren la vista de implementación estática de un sistema.

diagrama de comunicación diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben señales; diagrama que muestra interacciones organizadas alrededor de instancias y los enlaces de unas a otras.

diagrama de despliegue diagrama que muestra la configuración en tiempo de ejecución de los nodos de procesamiento y los componentes que residen en ellos; un diagrama de despliegue cubre la vista de despliegue estática de un sistema.

diagrama de estados diagrama que muestra una máquina de estados; los diagramas de estados cubren la vista dinámica de un sistema.

diagrama de interacción diagrama que muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden enviarse entre ellos; los diagramas de interacción cubren la vista dinámica de un sistema; término genérico que se aplica a varios tipos de diagramas que resaltan las interacciones entre objetos, incluyendo los diagramas de comunicación y los diagramas de secuencia. Los diagramas de actividades están relacionados, pero son semánticamente distintos.

diagrama de objetos diagrama que muestra un conjunto de objetos y sus relaciones en un momento dado; los diagramas de objetos cubren la vista de diseño estática o la vista de procesos estática de un sistema.

diagrama de secuencia diagrama de interacción que resalta la ordenación temporal de los mensajes.

dirigido por casos de uso en el contexto del ciclo de vida del desarrollo de software, proceso en el que se utilizan los casos de uso como artefactos principales para establecer el comportamiento deseado del sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para comunicación entre los usuarios del proyecto.

dirigido por el riesgo en el contexto del ciclo de vida del desarrollo de software, proceso en el que cada nueva versión se ocupa principalmente de acometer y reducir los riesgos más significativos para el éxito del proyecto.

disparar ejecutar una transición de estado.

dominio área de conocimiento o actividad que se caracteriza por un conjunto de conceptos y una terminología que entienden los profesionales de esa área.

ejecución puesta en marcha de un modelo dinámico.

elaboración segunda fase del ciclo de vida del desarrollo de software, en la cual se definen la visión del producto y su arquitectura.

elemento constituyente atómico de un modelo.

elemento derivado elemento del modelo que se puede calcular a partir de otro elemento, pero que se muestra por claridad o que se incluye por criterios de diseño, aunque no añade información semántica.

elemento parametrizado descriptor de un elemento con uno o más parámetros no ligados.

emisor objeto que envía una instancia de un mensaje a un objeto receptor.

enlace conexión semántica entre objetos; instancia de una asociación.

enumeración lista de valores con nombre utilizada como rango de valores de un tipo de atributo particular.

envío paso de una instancia de un mensaje desde un objeto emisor a un objeto receptor.

escenario secuencia específica de acciones que ilustra un comportamiento.

espacio de nombres ámbito en el que se pueden definir y utilizar nombres; dentro de un espacio de nombres, cada nombre denota a un elemento único.

especificación descripción textual de la sintaxis y la semántica de un bloque de construcción específico; descripción declarativa de lo que algo es o hace.

estado condición o situación en la vida de un objeto durante la cual satisface una condición, realiza alguna actividad o espera algún evento.

estado compuesto estado que consta de subestados concurrentes o subestados disjuntos.

estereotipo extensión del vocabulario de UML que permite crear nuevos bloques de construcción derivados a partir de los existentes pero específicos a un problema concreto.

estímulo operación o señal.

evento especificación de un acontecimiento significativo ubicado en el tiempo y en el espacio; en el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede disparar una transición de estado.

evento de tiempo evento que denota el tiempo transcurrido desde que se entró en el estado actual.

exportar en el contexto de los paquetes, hacer visible un elemento fuera del espacio de nombres en el que está contenido.

expresión cadena de texto que al evaluarse produce un valor de un tipo particular.

expresión booleana expresión que al evaluarse produce un valor booleano.

expresión de tiempo expresión que al evaluarse produce un valor absoluto o relativo del tiempo.

expresión de tipo expresión que al evaluarse produce una referencia a uno o más tipos.

extremo de asociación extremo de una asociación, que la conecta con un clasificador.

extremo de enlace instancia de un extremo de asociación.

fase intervalo de tiempo entre dos hitos importantes del proceso de desarrollo durante el cual se satisface un conjunto bien definido de objetivos, se completan artefactos y se toman las decisiones sobre si pasar a la siguiente fase.

foco de control símbolo sobre un diagrama de secuencia que muestra el período de tiempo durante el cual un objeto ejecuta una acción directamente o a través de una operación subordinada.

framework patrón arquitectónico que proporciona una plantilla extensible para las aplicaciones dentro de un dominio.

generalización relación de especialización/generalización, en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre).

herencia mecanismo por el que elementos más específicos incorporan la estructura y comportamiento de elementos más generales.

herencia de implementación el hecho de heredar la implementación de un elemento por parte de otro elemento más específico; también incluye la herencia de la interfaz.

herencia de interfaz el hecho de heredar la interfaz de un elemento por parte de otro elemento más específico; no incluye la herencia de implementación.

herencia múltiple variación semántica de la generalización en la que un hijo puede tener más de un padre.

herencia simple variación semántica de la generalización en la que un hijo sólo puede tener un padre.

hijo subclase u otro elemento especializado.

hilos flujo de control ligero que se puede ejecutar concurrentemente con otros hilos en el mismo proceso.

implementación realización concreta del contrato declarado por una interfaz; definición de cómo se construye o se computa algo.

importación en el contexto de los paquetes, dependencia que muestra el paquete a cuyas clases se puede hacer referencia dentro de un paquete dado (inclu-

- yendo a los paquetes incluidos recursivamente en él) sin proporcionar un nombre calificado.
- incompletitud** modelado de un elemento, sin considerar algunas de sus partes.
- inconsistencia** modelado de un elemento sin garantizar la integridad del modelo.
- incremental** en el contexto del ciclo de vida del desarrollo de software, proceso que implica la continua integración de la arquitectura del sistema para producir versiones, donde cada nueva versión incluye mejoras incrementales sobre las otras.
- ingeniería directa** proceso de transformar un modelo en código a través de una correspondencia con un lenguaje de implementación específico.
- ingeniería inversa** proceso de transformación de código en un modelo a través de una correspondencia con un lenguaje de implementación específico.
- invocación síncrona** solicitud en la que el objeto emisor se espera a recibir los resultados.
- instancia** manifestación concreta de una abstracción; entidad a la que se puede aplicar un conjunto de operaciones y que tiene un estado que almacena el efecto de las operaciones.
- integridad** relación correcta y consistente de unas cosas con otras.
- interacción** comportamiento que comprende un conjunto de mensajes que se intercambian entre un conjunto de objetos, dentro de un contexto particular, para lograr un objetivo.
- interfaz** colección de operaciones que se utiliza para especificar un servicio de una clase o un componente.
- iteración** conjunto bien definido de actividades, con un plan base y unos criterios de evaluación, que produce una versión, ya sea interna o externa.
- iterativo** en el contexto del ciclo de vida del desarrollo de software, proceso que implica la gestión de un flujo de versiones ejecutables.
- jerarquía de contención** una jerarquía de espacios de nombres que consta de elementos y las relaciones de agregación existentes entre ellos.
- Lenguaje de Restricción de Objetos (Object Constraint Language, OCL)** lenguaje formal que se utiliza para expresar restricciones libres de efectos laterales.
- ligadura** creación de un elemento a partir de una plantilla cuando se proporcionan argumentos para los parámetros de la plantilla.

- línea de vida de un objeto** línea en un diagrama de secuencia que representa la existencia de un objeto a lo largo de un período de tiempo.
- localización** asignación de un artefacto a un nodo.
- máquina de estados** comportamiento que especifica la secuencia de estados por los que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con la respuesta a esos eventos.
- marca de tiempo** indicación del momento en el que ocurre un evento.
- mecanismo** patrón de diseño que se aplica a una sociedad de clases.
- mecanismo de extensibilidad** cada uno de los tres mecanismos (estereotipos, valores etiquetados y restricciones) que permiten extender UML de forma controlada.
- mensaje** especificación de una comunicación entre objetos que transmite información con la expectativa de que se desencadenará actividad; la recepción de la instancia de un mensaje se considera normalmente una instancia de un evento.
- metaclase** clase cuyas instancias son clases.
- método** implementación de una operación.
- modelo** simplificación de la realidad, creada para comprender mejor el sistema que se está creando; abstracción semánticamente cerrada de un sistema.
- multiplicidad** especificación del rango de cardinalidades permisible que puede asumir un conjunto.
- nivel de abstracción** posición dentro de una jerarquía de abstracciones que abarca desde los niveles altos de abstracción (muy abstracto) hasta los niveles bajos de abstracción (muy concreto).
- nodo** elemento físico que existe en tiempo de ejecución y que representa un recurso computacional, que normalmente tiene algo de memoria y, a menudo, capacidad de procesamiento.
- nombre** cómo se llama a un elemento, relación o diagrama; cadena de texto que se utiliza para identificar un elemento.
- nota** símbolo gráfico para representar restricciones o comentarios asociados a un elemento o a una colección de elementos.
- objeto** manifestación concreta de una abstracción; entidad con unos límites bien definidos e identidad que encapsula estado y comportamiento; instancia de una clase.

objeto activo objeto que tiene un proceso o hilo y puede iniciar actividad de control.

objeto persistente objeto que existe después de que el proceso o hilo que lo creó deja de existir.

objeto transitorio objeto que existe sólo durante la ejecución del hilo o del proceso que lo creó.

ocurrencia instancia de un evento, incluyendo una ubicación en el tiempo y en el espacio y un contexto. Una ocurrencia puede disparar una transición en una máquina de estados.

omisión modelado de un elemento ocultando algunas de sus partes para simplificar la vista.

operación implementación de un servicio que puede ser requerido a cualquier objeto de la clase para que muestre un comportamiento.

padre superclase u otro elemento más general.

paquete contenedor de propósito general para organizar elementos en grupos.

parámetro especificación de una variable que puede cambiarse, pasarse o ser devuelta.

parámetro formal parámetro.

parámetro real argumento de una función o procedimiento.

patrón solución común a un problema común en un contexto determinado.

plantilla elemento parametrizado.

poscondición restricción que debe ser cierta al finalizar la ejecución de una operación.

precondición restricción que debe ser cierta cuando se invoca una operación.

proceso flujo de control pesado que puede ejecutarse concurrentemente con otros procesos.

producto artefacto del desarrollo, como los modelos, el código, la documentación y los planes de trabajo.

proyección correspondencia de un conjunto hacia un subconjunto de él.

propiedad valor con nombre que denota una característica de un elemento.

proveedor tipo, clase o componente que proporciona servicios que pueden ser invocados por otros.

pseudoestado nodo en una máquina de estados que tiene la forma de un estado pero que no se comporta como tal; los pseudoestados incluyen los nodos inicial, final y de historia.

realización relación semántica entre clasificadores, en la cual un clasificador especifica un contrato que otro clasificador se compromete a llevar a cabo.

recepción manejo de una instancia de un mensaje lanzada desde un objeto emisor.

receptor objeto al que se le envía un mensaje.

refinamiento relación que representa una especificación más completa de algo que ya ha sido especificado a cierto nivel de detalle.

relación conexión semántica entre elementos.

requisito característica, propiedad o comportamiento deseado de un sistema.

responsabilidad contrato u obligación de un tipo o una clase.

restricción extensión de la semántica de un elemento de UML, que permite añadir nuevas reglas o modificar las existentes.

restricción de tiempo declaración semántica sobre un valor absoluto o relativo de un período de tiempo o duración.

rol participante estructural en un contexto particular.

señal especificación de un estímulo asíncrono que es transmitido entre instancias.

signatura nombre y parámetros de una operación.

sistema conjunto de elementos organizados para lograr un propósito específico y que se describe por un conjunto de modelos, posiblemente desde diferentes puntos de vista. Un sistema se suele descomponer en un conjunto de subsistemas.

subclase en una relación de generalización, el hijo, que es la especialización de otra clase.

subestado estado que forma parte de un estado compuesto.

subestado ortogonal subestado ortogonal que puede darse simultáneamente con otros subestados contenidos en el mismo estado compuesto.

subestado no ortogonal subestado que no puede darse simultáneamente con otros subestados contenidos dentro del mismo estado compuesto.

subsistema agrupación de elementos, algunos de los cuales constituyen una especificación del comportamiento ofrecido por los otros elementos.

superclase en una relación de generalización, el padre, que es la generalización de otra clase.

tarea flujo de ejecución único a través de un programa, un modelo dinámico o alguna otra representación de un flujo de control; hilo o proceso.

tiempo valor que representa un momento absoluto o relativo.

tipo relación entre un elemento y su clasificación.

tipo de datos tipo cuyos valores no tienen identidad. Los tipos de datos incluyen tipos primitivos predefinidos (tales como números y cadenas de caracteres), así como tipos enumerados (tales como los booleanos).

tipo primitivo tipo básico, tal como un entero o una cadena de caracteres.

transición cuarta fase del ciclo de vida del desarrollo de software, en la que el software se pone en manos de la comunidad de usuarios; relación entre dos estados que indica que un objeto en el primer estado realizará ciertas acciones y pasará al segundo estado cuando ocurra un evento específico y se satisfagan ciertas condiciones.

traza dependencia que indica una relación de proceso o histórica entre dos elementos que representan el mismo concepto, sin reglas para derivar el uno del otro.

UML Lenguaje Unificado de Modelado, un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software.

unidad de distribución conjunto de objetos o componentes que se colocan en un nodo como un grupo.

uso dependencia en la que un elemento (el cliente) requiere la presencia de otro elemento (el proveedor) para un funcionamiento o una implementación correcta.

valor elemento del dominio de un tipo.

valor etiquetado extensión de las propiedades de un elemento de UML, que permite crear nueva información en la especificación de ese elemento.

versión conjunto relativamente completo y consistente de artefactos entregado a un usuario externo o interno; la entrega de ese conjunto.

visibilidad modo en que puede ser visto y utilizado un nombre por otros.

vista proyección de un modelo, que se ve desde una perspectiva o un punto de vista dado, y que omite entidades que no son relevantes desde esa perspectiva.

vista de casos de uso vista de la arquitectura de un sistema que incluye los casos de uso que describen el comportamiento del sistema tal y como es percibido por sus usuarios finales, los analistas y los encargados de las pruebas.

vista de despliegue vista de la arquitectura de un sistema que incluye los nodos que forman la topología hardware sobre la cual se ejecuta el sistema; una vista de despliegue cubre la distribución, entrega e instalación de las partes que configuran el sistema físico.

vista de diseño vista de la arquitectura de un sistema que incluye las clases, interfaces y colaboraciones que forman el vocabulario del problema y su solución; una vista de diseño se ocupa de los requisitos funcionales de un sistema.

vista de implementación vista de la arquitectura de un sistema que incluye los componentes que se utilizan para ensamblar y hacer disponible el sistema físico; una vista de implementación abarca la gestión de configuraciones de las versiones del sistema, formada por componentes relativamente independientes que se pueden ensamblar de varias formas para producir un sistema ejecutable.

vista de interacción vista de la arquitectura de un sistema que incluye los objetos, hilos y procesos que conforman la concurrencia y los mecanismos de sincronización del sistema, el conjunto de actividades y el flujo de mensajes, control y datos entre ellas. La vista de interacción también abarca el rendimiento, la escalabilidad y la capacidad de procesamiento del sistema.

vista dinámica aspecto de un sistema que destaca su comportamiento.

vista estática aspecto de un sistema que destaca su estructura.

ÍNDICE

A

- ABMC 445
abstracción 58, 106, 108, 187-190, 201, 351
definición 497
nivel de 505
access 147, 179
acción 24, 232, 243, 293-295, 325, 376
asíncrona 451, 497
definición 328, 376, 497
acción de entrada 330, 335-336
acción de salida 330
acción síncrona
definición 497
Actives 449
actividad 25, 291-293, 334, 374
definición 293, 328, 376, 493, 497
actividades-do 337
actor 24, 243, 245, 255-256, 259, 261, 314, 353, 467
definición 247, 497
símbolo 248
Ada 132, 352
adorno 31, 81, 85, 107
definición, 497
agregación 27, 54, 72, 78, 120, 154, 231
composición 154
compuesta 73
definición 72, 497
símbolo 72
y atributos 54
agregación compuesta 73, 78, 176
agregado
definición 497
alcance 30, 135
alcance estático 132
algoritmo 225, 307-308, 325
alt 277
análisis 59, 244, 493
analistas 37
andamiaje 101
API 220-221, 440
aplicación 24, 43
applet 42, 47-48, 58
arcos 485
archivo 24, 58, 392, 394-395, 437, 440
armonía 408
arquitecto 98
arquitectura 11, 16, 18, 36, 127, 351, 407, 423, 425, 439, 456, 488
definición 36, 497
documentación 18-19
estilo 424
flexibilidad 408
integración continua de la 38
modelado 99, 470
patrón 424
vistas 36, 98, 148, 157, 159, 181, 183, 283, 303, 377, 399, 408, 466, 468, 470, 493, 495

artefacto 15-16, 18-19, 23, 47, 387, 437, 439, 469, 494
 definición 389, 493, 498
 nombre 389
 símbolo 23, 388
 técnico 495
B
 artefactos 37
 asíncrono 234, 315-316, 328-329, 355
 asociación 27, 65, 97, 115, 129, 143, 146, 170, 187, 189, 192, 193, 229-231, 248, 261, 354, 403, 439, 455, 484
 agregación 72
 binaria 498
 calificación 153
 composición 154
 definición 69, 151, 498
 dirección 70
 instancia de 187
 multiplicidad 71, 135
 nombre 70
 n-aria 70
 navegación 70, 152
 rol 70-71
 símbolo 27, 70
 visibilidad 152
 asociación binaria
 definición 498
 asociación n-aria 70
 definición 498
 atributo de clase 132, 193
 atributos 20, 54, 127, 132-133, 146, 153-154, 156, 192, 237, 275, 467
 alcance 135
 características 55
 definición 54, 498
 estáticos 132
 multiplicidad 134-135
 nombre 54
 organización 56
 símbolo 54
 visibilidad 135

y la agregación 54
 y la composición 155
 y los valores etiquetados 87
B
 balanceo de la carga 265
 base de datos 118, 443, 445-447
 física 147
 lógica 116, 118, 147, 445
 orientada a objetos 18, 440
 relacional 18
 replicación 447
 biblioteca 24, 391-393, 437
 biblioteca de clases 429
 biblioteca dinámica 391
 bind 146
 bloque 173, 409, 411
 Booleano
 definición 498
 broadcasting 318
 bucle 277
 buzón 355
C
 C 41, 64
 C++ 18, 34, 64, 132, 233, 352, 440
 calificación 153
 calificado 246-247
 calificador
 definición 498
 calle 299
 definición 498
 cambio 317
 cambio de estado 315
 camino 230, 281, 287,
 capacidad de procesamiento 37
 característica
 de comportamiento 129, 498

definición 498
 estructural 129, 498
 cardinalidad
 definición 498
 caso de uso 22, 37, 77, 108, 127, 130, 143-144, 147, 158, 176, 187-189, 226-228, 243-245, 249, 259, 261, 283, 286, 304, 307, 325-326, 329, 347, 409, 413, 416-417, 433, 461, 467, 481, 488
 bien estructurado 257
 definición 246, 498
 instancia 187
 nombre 247
 representación 257
 símbolo 22, 246
 casos de uso 128
 centrado en la arquitectura 38, 488
 definición 497
 ciclo 447-448
 ciclo de desarrollo 492
 ciclo de vida 38-39, 490
 ciclo de vida del desarrollo de software 490
 clase 21, 37, 42, 97, 101, 113, 115, 128, 157, 162, 177, 188, 226-229, 245, 259, 283-284, 286, 303, 307, 325-326, 347, 354, 359, 377, 389, 399, 402, 409, 413, 433, 465, 481
 abstracta 76, 133, 167, 189, 320, 499
 atributos 54
 base 68
 bien estructurado 51, 64
 características avanzadas 51, 127
 concreta 133, 138, 499
 definición 53, 499
 hoja 69, 134
 jerarquía 133
 nombre 53

omisión 56
 operación 55-56
 pasiva 359
 raíz 68
 representación 64
 semántica 58, 140-141
 símbolo 20, 53
 singleton 134
 plantilla 138, 322
 utilidad 134
 clase activa 22, 37, 102, 193, 283, 353-355, 359, 418, 481
 bien estructurada 362
 definición 353, 498,
 representación 363
 símbolo 22, 353
 clase asociación 73, 155
 definición 499
 clasificación
 dinámica 499
 estática 499
 múltiple 499
 clasificador 20, 127, 129, 131, 190, 254, 410, 413, 467
 bien estructurado 142
 definición 128, 499
 representación 142
 cliente 459
 definición 499
 cliente ligero 458
 cliente pesado 459
 código 454
 código fuente 18, 395-396,
 colaboración 21, 37, 46, 97, 101, 108, 110, 113, 116-117, 143, 158, 167-168, 176, 225, 227, 250, 283, 304, 307, 347, 361, 399, 426, 431, 433, 467, 481
 bien estructurada 420
 definición 409, 499
 estructura 410
 nombre 410

representación 421
organización 413
símbolo 21, 409
color 87
COM+ 440, 443
comentario 26, 90
definición 499
comillas francesas 84, 86
compartimento extra 85, 253, 389
compilación 395
complete 150
componente 21, 97, 101-102, 108, 127-129, 143, 157, 167, 176, 188-189, 207-208, 226, 228, 283, 307, 326, 366, 368, 370, 402, 409, 413, 433, 444, 465, 481
e interfaces 209
definición 499
distribución 405
introspección 168
migración 368
notación 216
y nodos 401
componente de despliegue 391
componente de ejecución 391
componente producto del trabajo 391
comportamiento 21, 37, 99, 103, 134, 162, 225, 243-244, 254, 259-260, 315, 325, 327, 329, 368, 412, 454, 467, 470
definición 499
composición 154, 143, 468
definición 499
y atributos 154
compromisos hardware/software 457
compuesto
definición 499
computación 294-295
comunicación 288
comunicación entre procesos 361
con guardas 138, 357, 360

concepción 39, 489-491
definición 499
concurrencia 136, 138, 167, 297, 300, 332, 346, 352, 354
definición 500
condición 282
guarda 333
condición de guarda 277, 331-333
definición 500
conectar las partes 216
conector 28, 209, 216
conector de delegación 217
conexión 403, 484
configurable 488
conjunto de análisis y diseño 495
conjunto de despliegue 495
conjunto de diseño 495
conjunto de implementación 495
conjunto de prueba 495
conjunto de requisitos 495
constante 368
construcción 39, 489-491
definición 500
consulta 137
contenedor
definición 500
contexto 262-263, 304
definición 500
contrato 33, 163-165
control de calidad 489
control de versiones 395
CORBA 163, 391, 440
corrutina 298
creación 232, 237, 315, 328,
crear 232
crecimiento incremental 487-488

D

delegación
definición 500

dependencia 26, 65, 84, 97, 115, 139, 143, 165, 170, 179, 194, 252-253, 261, 287, 315, 354, 402, 439, 455, 469, 484
definición 67, 145, 500
nombre 68
símbolo 26, 68
depurador 194
derive 146
despliegue 493,
destrucción 232, 237, 315
diagrama 20, 27-28, 97, 99, 176, 274, 485
bien estructurado 111
color 87
creación 110
definición 500
estructural 101-102
representación 111
diagrama de actividades 29, 37, 100, 104, 140, 190, 259, 271, 291, 293, 303-304, 325, 373, 377, 418, 454, 486
definición 104, 293, 500
representación 310
símbolo 273
diagrama de ártefactos 30, 100-102, 450
bien estructurado 451
definición 438
representación 451
diagrama de casos de uso 28, 37, 100, 103, 259-260, 271, 291, 373, 486
bien estructurado 268
definición 103, 261, 500
representación 269
diagrama de clases 28, 37, 43, 45, 100-101, 113, 146, 170, 190, 197-198, 226, 326, 409, 432, 454-455, 485
bien estructurado 124

definición 102, 115, 500
representación 124
símbolo 115
diagrama de componentes 29, 47, 100-102, 113, 115, 201, 437, 444, 454, 485
definición 500
símbolo 438
diagrama de comportamiento 103
diagrama de comunicación 103, 271, 280
definición 104, 273, 500
diagrama de despliegue 29, 37, 100-101, 113, 115, 195, 200-201, 368, 437, 453, 455, 459, 463, 486
bien estructurado 464
definición 102, 455, 501
representación 464
símbolo 455
diagrama de estados 28, 37, 100, 103, 259, 271, 291, 325, 373, 377, 454, 486
bien estructurado 382
definición 104, 375, 501
representación 383
símbolo 375
diagrama de estructura compuesta 100-102,
diagrama de estructura interna 37
diagrama de flujo 291, 293, 308, 419
diagrama de Gantt 292, 374
diagrama de interacción 29, 37, 103, 170, 190, 198, 238, 249, 271-272, 283, 286-287, 292-293, 353, 359-360, 410, 412, 415, 448
bien estructurado 289
definición 103, 273, 501
representación 290
diagrama de objetos 28, 37, 100-101, 190, 197-198, 226, 229, 231, 326, 447, 462, 485

bien estructurado 204
definición 102, 199, 504
representación 191
símbolo 199
diagrama de paquetes 30
diagrama de Pert 292, 374
diagrama de secuencia 46, 100, 103, 237, 249, 259, 271, 274, 283, 291, 373, 432, 454, 486
definición 103, 273, 501
notación 277
símbolo 273
diagrama de tiempos 30
diagrama de visión global de interacciones 30
diagrama estructural 101
dicotomía clase/instancia 369
dicotomía clase/objeto 33, 146, 187
dinámico 103, 190, 225-226, 229, 259, 265, 271-272, 283, 291-292, 303-304, 325, 373, 407, 477
dirección 70, 135
dirigido por casos de uso 38, 488
definición 501
dirigido por riesgos 38
definición 501
disciplina 493
discriminante 73
diseño 18, 493
disjoint 151
disparador 313, 316, 331-332
polimórfico 333
disparador de evento 331-334
disparar
definición 501
dispositivo 403, 454, 457
distribución 37, 118, 209, 361, 365, 370, 405-406, 456
distribución de responsabilidades 60, 359
división 297-298

DLL 391
documento 24, 391, 393, 437, 488
dominio
definición 501
E
eclipse 168
efecto 325, 332, 334
entrada 335
salida 335
Eiffel 132
ejecución 30, 353
definición 501
ejecutable 391-392, 437-439, 442
elaboración 39, 489-491
definición 501
elemento 20, 189, 226, 438, 481, 485
de agrupación 25, 483
definición, 501
de comportamiento 24, 477, 482
de notación 26, 483
estructural 477, 481
elemento abstracto 76, 133, 167, 189, 229, 320
definición 497
elemento derivado
definición 501
elemento estructural 20, 101, 477, 481
elemento no software 51, 62
elemento parametrizado
definición 502
else 296
emisor
definición 502
encargados de las pruebas 37
enlace 71, 73, 157, 189, 193, 200, 216, 231, 281-282, 287, 484
definición 229, 502

enlace transitorio 216
entorno 493
entrega 37
entregable 494
enumeración 64
definición 502
envío 148, 232, 294, 315, 334
definición 502
equilibrio en el diseño 408
escalabilidad 37
escenario 117, 144, 229, 249-250, 292, 304, 415, 488
definición 502
escritor 358
esencia 188
espacio 24, 295, 365, 378
bien estructurado 372
representación 372
espacio de nombres 176
definición 502
especialización 76, 484
especificación 31
definición 502
esquema 116, 118-119, 440, 445
estado 24, 103, 187, 192, 197, 231-232, 237, 275, 300, 327, 329, 374
compuesto 338, 376, 502
definición 328, 375, 502
destino 332
final 331
historia 341
inicial 331
origen 332
partes 330
pseudo 507
símbolo 24, 300
simple 376
subestado 338
estático 99, 101, 132, 138, 190, 193, 226, 229, 313, 326, 407, 454, 456, 477
expresión 294, 297
booleana 502

definición 502
de tiempo 502
tipo 503
expresión de tiempo 365
 definición 332, 502
extend 147, 253
extensibilidad 145, 485
extremo de asociación
 definición 503
extremo de enlace
 definición 503

F

fase 38-39, 490
 definición 489, 503
física 389, 426
flexibilidad 408
flujo 107
 flujo de control 227, 235, 238, 281,
 283-287, 291, 297-298, 300,
 325, 351, 353-354, 358, 359,
 360, 374
 anidado 235
 independiente 352
 múltiple
 plano 235
 por organización 240
 por tiempo 239
 procedimental 235
 sincronización 357
flujo de objetos 300
flujo de trabajo 292, 304-305, 377
foco de control 275
 definición 503
fotograma 232
framework, 26, 45, 162, 168, 419,
 423-424, 428, 433
 definición 425, 503
friend 146
frontera hardware/software 454

G

generación de software 492
generalización 26-27, 43, 65, 68, 75,
 86, 91, 97, 115, 133, 143, 170,
 247, 251, 261, 320, 354, 439,
 484
 definición 68, 149, 503
 nombre 69
 símbolo 27, 68-69
gestión de la configuración 37, 395,
 443, 493
gestión de proyectos 493
gestión de riesgos 489
global 230
guarda 237

H

Hardware 51, 58, 62, 399-400, 453-
 454, 457-458
Harel, David 376
herencia 75, 136
 definición 503
 implementación 503
 interfaz 504
 jerarquía 150
 mixin 150
 múltiple 69, 76, 150, 503
 simple 69, 75, 149, 503
hijo 27, 44, 66, 68, 86, 134, 149,
 484
 definición 503
hilo 24, 138, 193, 234, 236, 297, 318,
 351, 353-354, 371
 definición 503
historia 200, 272, 292
hoja 40, 69, 134-135,
¡Hola, mundo! 41
huérfano 190

I

Icono 87
IDL 163, 440
implementación 33, 469, 492
 definición 503
import 147, 179
 definición 503
in 137
include 147-148, 252, 254
incomplete 150
 definición 504
inconsistencia
 definición 504
incremental
 definición 504
ingeniería de ida y vuelta 18
ingeniería del software
 objetivo 487
ingeniería directa 18, 43, 99, 121-
 122, 128, 144, 266-267, 288-
 289, 309, 381, 448-449, 462
 definición 504
ingeniería inversa 18, 43-44, 99, 123,
 128, 144, 203, 266-268, 288-
 289, 309, 382, 448-449, 462-463
 definición 504
inout 137
instalación 37
instanceOf 146, 194
instancia 46, 129, 132-135, 170-171,
 187, 198, 200, 226, 228-229,
 250, 318, 325-326, 354, 377,
 412, 455, 467
 anónima 188, 191
 bien estructurada 196
 concreta 187, 195
 definición 189, 504
 directa 133, 189
 indirecta 189
 prototípica 187, 189
 representación 195

J

J2EE 12
Java 18, 34, 42-47, 48, 58, 121, 123,
 132, 168, 352, 381, 388-389,
 391, 440, 443

jerarquía 133-134, 136
contención 469, 504

L

LAN 461
lector 358
Lenguaje de Restricción de Objetos 89, 93, 140, 141, 167
definición 504
ligadura
definición 504
Línea de separación 166, 168
Línea de vida 237, 274
definición 505
Línea de vida de un objeto
definición 505
local 231
localización 361, 366, 367-368, 447
definición 505
lógico 390, 470
lugar de ejecución 354
llamada 232-233, 294, 313, 315-316, 319, 355, 361

M

Maine 424
manifestación 188-189
máquina de estados 24, 148, 167, 192, 313, 315-317, 319, 325, 327, 329, 355, 361, 373, 399, 467, 482
bien estructurada 349
definición 328, 375, 505
representación 350
máquina de Mealy 378-379, 383
máquina de Moore 378-379, 383
marca de tiempo 237, 285, 287, 365
definición 367, 505

símbolo 366
materializar 359
mecanismo 117, 201-202, 358, 408, 419-420, 423-424, 426-427
de extensibilidad 48, 81, 505
definición 425, 505
mejores prácticas 487
mensaje 24, 103, 107, 134, 225-226, 229-230, 232-235, 237-238, 271-272, 274, 315, 352, 355-356, 367-368
asíncrono 234
con guarda 237
definición 227, 505
símbolo 24
síncrono 232
mensajes
tipos de 233
metaclass 140
definición 505
metamodelo 86
método 136, 165, 315
definición 505
método de desarrollo de software 15, 38-39
migración 368
mixin 150
modelado 16-17, 51, 97, 121, 465, 477, 480
algorítmico 12
arquitectónico 385
bello 407
casa 4, 51, 66, 98, 101, 113, 127, 144, 161, 174, 187, 244, 292, 326, 352, 423, 437, 466
caseta de perro 4, 5-7, 174, 226, 352, 388, 465
de la arquitectura 470
del comportamiento 99, 225, 311
edificio 174, 226, 352, 374, 387-388, 466

estructural 99
estructuras de datos 200
estructural avanzado 127
feo 407 (OJO, figura horrible)
físico 399
hardware 62
importancia del 4
lógico 400
mundo real 351
no software 399
objetivos del 7
orientado a objetos 10, 11-12
principios 8
sistema no software 19
software 5
tiempo y espacio 365
modelado de casos de uso del negocio 494
modelado del comportamiento 223
avanzado 311
modelado del negocio 493
modelado estructural 49
avanzado 125
modelo 16, 26, 148, 409, 468, 488, 494-495
bien estructurado 473
bien formado 31, 233
definición 6, 99 465, 505
ejecución 18
simulación 18
modelo de análisis 494
modelo de análisis del negocio 494
modelo de casos de uso 494
modelo de datos 494
modelo de despliegue 494
modelo de diseño 494
modelo de implementación 494
modificación 237
multicasting 318
multiplicidad 71, 127, 134-135, 231, 460
definición 505

N

navegación 72, 73, 143, 152, 231
.NET 12, 168, 388
nivel de abstracción
definición 505
nodo 23, 37-38, 62, 97, 101-102, 108, 127-129, 176, 187-189, 226, 229, 283, 307, 326, 361, 366, 390, 399-400, 409, 433, 437, 443, 455, 457, 465, 481
bien estructurado 406
definición 400, 505
instancia de 187
nombre 400
representación 406
símbolo 23, 400
y componentes 401
nodo de actividad 294-295
Nombre 30, 190, 232, 246, 330
calificado 190
definición 505
simple 190
nombre calificado 53
nombre de extremo 70
nombre simple 53
nota 26, 42, 68, 81, 84, 265, 285, 294, 361, 418, 483
bien estructurado 94
definición 83, 505
representación 94
símbolo 26, 83
Notre Dame 407

O

objetivo del negocio 488
objeto 24, 52, 101-104, 187, 189, 200, 225, 228-229, 274, 293-294, 346-347, 352, 462, 481, 484

activo 356, 506
anónimo 191
bien estructurado 196
concreto 229
creación 226
definición 189, 505
destrucción 226
estado de 192
huérfano 190
nombre 190
pasivo 356
persistente 506
prototípico 229
reactivo 373, 377-378
representación 195
transitorio 506
ver también instancia
vida 325, 373
objeto activo 138, 297, 329, 346, 356
bien estructurado 362
definición 353, 506
representación 363
objeto pasivo 356
objeto persistente
definición 506
objeto reactivo 373, 377-378
objeto transitorio
definición 506
OCL. Ver Lenguaje de Restricción de Objetos
ODBC 445
omisión
definición 506
operación 24, 42, 55, 127, 132-133, 134, 136, 164, 187, 191, 226-228, 232, 234, 236, 283-284, 286, 293-294, 304, 307, 319, 325, 409, 418, 467
abstracta 76, 134
características 55
definición 55, 506
estática 132-133

hoja 134
nombre 55
organización 56
polimórfica 134
símbolo 55
operación concreta 133
operación concurrente 138
operador de control 276-278
operador de control condicional 277
operador de control de bucle 277
operador de control estructurado 276-277
operador de control opcional 277
operador de control paralelo 277
opt 277
ORB 362
orden temporal 271, 272, 274, 283
ordered 156
organización 240, 283
orientado a objetos 488
out 137
overlapping 151

P

padre 27, 43-44, 66, 68-69, 86, 149, 484
definición 506
página 24, 58
paquete 25, 42, 45, 53-54, 68, 101, 110, 115, 129, 131, 147, 153, 164, 174, 212, 246-247, 251, 299, 393, 400, 402, 410, 414, 428, 433, 441, 455, 460, 461, 468, 483
bien estructurado 184
definición 175, 506
elementos 176
nombre 175
raíz 177
representación 185
símbolo 25, 175

par 277
parámetro 42, 137, 228, 230
definición 497, 506
formal 506
real 506
parte 72, 154, 209, 214, 467
patrón 122, 168, 408, 419, 423
arquitectónico 472
bien estructurado 435
definición 408, 425, 506
diseño 424, 472
representación 435
patrones arquitectónicos 384, 433-434, 472
permit 146
persistencia 116, 440
pizarra 425
plan del proyecto 18
planos de software 15-16, 82
plantilla 138, 146, 426
definición 506
polimórfico 134
polimorfismo 68, 127, 136, 191, 333
posesión 177
postcondición 141, 167, 305, 379
definición 506
precondición 141, 167, 305, 379
definición 506
privado 131, 152-153
procesador 403-404, 454
proceso 24, 138, 193, 234, 236, 297, 318, 351, 353-354, 371, 480, 487
centrado en la arquitectura 488
configurable 488
definición 506
dirigido por casos de uso 488
iterativo 38, 487
orientado a objetos 488
proceso incremental 38

Proceso Unificado de Rational 86, 159, 472, 488-489
producto
definición 506
propiedad 50, 91-92
definición 506
protected 131
prototípico 187, 189
prototipo 18
proveedor
definición 506
proxy 415, 440
proyección 495
definición 506
pruebas 18, 245, 493
pseudoestado 331
definición 507
público 131, 152-153
puerto 108-109
notación 212

R

raíz 68
ramificación 237, 277, 281, 297
símbolo 297
reactivo 106
readonly 136, 157
realidad 494
realización 27, 33, 108, 143, 165-166, 208, 210, 250, 413, 416-418, 439
definición 27, 157, 507
símbolo 27
recepción
definición 507
receptor
definición 507
recursión 354
red 195, 265, 366, 462
red de área extensa 461

red de área local 461
refinamiento 65, 143, 414
definición 507
refinamiento sucesivo 487
refine 147
región de expansión 300
relación 20, 26, 65, 143, 359, 410, 438, 484-485
bien estructurada 160
definición 67, 145, 507
estructural 66
padre/hijo 66
red de 143, 159
redes de 65
reflexiva 69
representación 80, 160
símbolo 67, 145
uso 66
y estereotipos 86
relación es-un-tipo-de 68
relación tiene-un 72
relación todo/parte 72
relaciones de generalización 129
rendezvous 319, 355
rendimiento 37
replicación 447
requisito 18, 39, 90, 244-245, 259, 262, 264-266, 469
definición 507
funcional 37, 245
requisitos 493
responsabilidad 57, 128, 140-141, 286
definición 57, 507
distribución 60-61
símbolo 57
restricción 26, 34, 63, 81, 88, 237, 286, 294, 355, 358, 365, 367, 485
bien estructurada 95
definición 84, 507
representación 94
símbolo 84

temporal 368, 370, 507
restricción de tiempo 370
definición 367, 507
símbolo 368
retorno 136-137, 232, 236, 276, riesgo 448
rol 225, 237-238, 275, 410
definición 163, 507
roles 237-238
RS-232 403

S

San Francisco 423
satisfacción inmediata 41
sd 277
secuencia 234, 245, 281-282, 353
anidada 236
plana 235
procedimentales 326
secuencia procedural 235
secuencial 138, 357-358, 360
self 230
semántica 17, 88, 93, 140-141, 485
señal 24, 127-130, 136, 165-166, 226, 286, 294, 313, 315, 318-319, 325, 329, 355, 361
definición 314, 507
familia de 320, 333
jerarquía de 320
parámetros 315
y clases 315
separación de intereses 64, 162
servidor 459-460
signatura 43, 74, 127, 134, 136-137
bien estructurado 310
definición 507
símbolo 293
sincronización 357
sincronización sin espera (balking) 356

síncrono 232, 316, 318, 355
singleton 134
sintaxis 485
sistema 148, 211, 227, 234, 243, 245, 255, 259, 263, 283-284, 286, 303-304, 325, 347, 399, 408, 465-468
adaptable 440, 447
bien estructurado 423, 473
cliente/servidor 400, 456, 458-460
concurrente 357
con gran cantidad de software 19, 226, 260, 388, 453
contexto 304
de cine doméstico 207
de sistemas 467, 472
definición 99, 467, 507
de tiempo real 365-366
dirigido por eventos 192
distribuido 209, 365-366, 368-369, 400, 443, 456, 460-461
embebido 456-457
frontera 454
frontera hardware/software 454
hardware 453

T

independiente 400
línea de separación 166, 168
patrones en un 425
reactivo 106
representación 473
secuencial 353
símbolo 466
sin conflictos 365
software 453
topología 392, 454
vocabulario 51, 59, 116, 129
Smalltalk 61, 233, 352
sociedad 231, 240, 250, 307, 325, 407
software 51, 453, 457
SQL 445

subclase 68
definición 507
subestado 330-331, 338
concurrente (en glosario concurrencia, 500)
definición 507
no ortogonal 338, 507
ortogonal 343, 507
subestado concurrente 338
subestado secuencial 338
subregión 277
subsistema 26, 102, 115, 127-130, 163, 211, 227, 245, 255, 259, 263, 283-284, 286, 304, 455, 466-468
definición 467, 507
representación 473
subsistema de 99
símbolo 467
sujeto 261
superclase 68
definición 508
supratipo 140, 146
sustitución 210-211, 484

tiempo de envío 367
 tiempo de recepción 367
 tiempo de transmisión 367
 timeout 357
 tipado dinámico 233
 tipado estático 233
 tipo 63, 151
 definición 163, 508
 dinámico 170, 233
 estático 170, 233
 primitivo 63, 508
 tipo de datos 127-130
 definición 508
 todo 72, 154, 468
 topología 195, 454, 459, 461
 trabajo 245
 transición 24, 39-40, 107, 313, 315,
 325, 327, 331, 376, 490, 491-492
 definición 328, 375-376, 508
V
 interna 329, 336
 partes 332
 símbolo 328
 transición de terminación 333
 traza 65, 108, 148, 396, 441, 469
 definición 508

U

UML
 bloques de construcción 19
 definición 508
 diagramas 27
 elementos 20
 equilibrio en 130
 especificación 480
 extender 81
 historia xvi
 mecanismos de extensibilidad 34
 mecanismos en 31, 81
 metamodelo 86
 modelo conceptual 19, 478

notación 481
 objetivos xvii
 propósito 13
 relaciones 26
 reglas 30
 transición a 477
 visión global 16
 y hardware 62
 unidad de distribución 402
 unión 297
 Unís 441
 URL 84
 uso 147
 definición 508
 usuario 38
 usuario final 37

V

valor etiquetado 35, 81, 87, 92, 355,
 365, 485
 bien estructurado 95
 definición 84, 508
 representación 94
 responsabilidad 57
 símbolo 84
 y atributos 87
 variante 245
 versión 18, 37, 38, 439, 443, 469-470
 definición 508
 versión 469-470
 VHDL 62, 400, 454
 vida 325, 328, 346, 349, 373, 378
 vínculos 469
 visibilidad 30, 127, 131, 135, 152, 178
 definición 508
 vista 36, 51 99, 105, 110, 466-469,
 470, 495
 de casos de uso 37, 98, 106,
 471, 495, 509
 definición 99, 508

dinámica 509
 estática 509
 vista de despliegue 37, 99, 106, 369,
 456, 470, 472, 495
 definición 509
 vista de diseño 37, 98, 106, 116, 197,
 200, 471, 495
 definición 509
 vista de implementación 37, 99, 106,
 437, 439, 470-471, 495
 definición 509
 vista de interacción 90, 106, 471,
 495
 vista de procesos 38, 197, 200, 359-
 360, 471
 definición 508, 509

W
 WAN 461
 Web 11, 15, 42, 47, 58

vista física 388
 vista lógica 388
 Visual Basic 18
 vocabulario 51, 59, 116, 129, 143,
 159, 300, 305, 425, 445

X

XML 379