

MANUAL DE UML



MANUAL DE UML

PAUL KIMMEL

Traducción

José Hernán Pérez Castellanos

Traductor profesional



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

Director Editorial: Fernando Castellanos Rodríguez
Editor de desarrollo: Cristina Tapia Montes de Oca
Supervisor de producción: Jacqueline Brieño Álvarez
Diagramación: By Color Soluciones Gráficas

MANUAL DE UML

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin autorización escrita del editor.



DERECHOS RESERVADOS © 2008 respecto a la primera edición en español por
McGRAW-HILL INTERAMERICANA EDITORES, S.A. de C.V.

A Subsidiary of The McGraw-Hill Companies, Inc.

Corporativo Punta Santa Fe
Prolongación Paseo de la Reforma 1015 Torre A
Piso 17, Col. Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. núm. 736

ISBN 970-10-5899-2

Translated from the 1st English edition of

UML DEMYSTIFIED

By: Paul Kimmel

Copyright © MMVI by The McGraw-Hill Companies, Inc. All rights reserved.

ISBN: 0-07-226182-X

1234567890

09765432108

Impreso en México

Printed in Mexico

*A la memoria de mi hermana Jennifer Anne
a quien sólo se le concedieron 35 años.*



ACERCA DEL AUTOR

Paul Kimmel es arquitecto en jefe y uno de los fundadores de Software Conceptions, Inc. Ha estado diseñando e implementando software orientado a objetos desde 1990, tiene más de 12 años de experiencia con los lenguajes de modelado, y fue uno de los primeros en adoptar el Unified Modeling Language. Paul ha ayudado a diseñar e implementar soluciones con el uso del UML para algunas de las más grandes corporaciones del mundo, desde bancos internacionales, empresas multinacionales de telecomunicaciones, empresas de logística y embarque, oficinas del Departamento de Defensa hasta grupos gubernamentales, nacionales e internacionales.

CONTENIDO BREVE



CAPÍTULO 1	Una imagen vale más que mil líneas de código	1
CAPÍTULO 2	El principio con casos de uso	17
CAPÍTULO 3	Diagramación de características como procesos	47
CAPÍTULO 4	Comportamientos con diagramas de interacción	81
CAPÍTULO 5	¿Cuáles son las cosas que describen mi problema?	101
CAPÍTULO 6	Cómo se relacionan las clases	131
CAPÍTULO 7	Uso de los diagramas de esquemas de estado	157
CAPÍTULO 8	Modelado de componentes	175
CAPÍTULO 9	Ajuste y finalización	185
CAPÍTULO 10	Visualización de su topología de despliegue	197
APÉNDICE A	Examen final	209
	Bibliografía seleccionada	225
	Índice	227



CONTENIDO

	Reconocimientos	xv
	Introducción	xvii
CAPÍTULO 1	Una imagen vale más que mil líneas de código	1
	Comprensión de los modelos	2
	Comprensión del UML	3
	La evolución del diseño de software	3
	Si nadie está modelando, ¿por qué debe hacerlo usted?	5
	Modelado y el futuro del desarrollo de software	5
	Herramientas para modelado	5
	Uso de los modelos	6
	Creación de diagramas	7
	Revisión de los tipos de diagramas	7
	Hallar la línea final	12
	¿Cuántos diagramas debo crear?	12
	¿Cuán grande debe ser un diagrama?	13
	¿Cuánto texto debe complementar mis modelos?	13
	Obtenga una segunda opinión	13



	Contraste de los lenguajes de modelado con el proceso	14
	Examen	14
	Respuestas	16
CAPÍTULO 2	El principio con casos de uso	17
	Cómo hacer el caso para los casos de uso	18
	Establecimiento de prioridad de las capacidades	19
	Comunicación con los no tecnófilos	20
	Uso de los símbolos de los casos de uso	21
	Símbolos de actores	21
	Casos de uso	21
	Conectores	22
	Casos de uso de inclusión y de extensión	25
	Anotaciones en los diagramas de casos de uso	27
	Creación de los diagramas de casos de uso	32
	¿Cuántos diagramas son suficientes?	34
	Ejemplos de diagramas de casos de uso	34
	Diseño controlado con casos de uso	43
	Examen	44
	Respuestas	46
CAPÍTULO 3	Diagramación de características como procesos	47
	Elaboración de las características como procesos	48
	Un viaje hacia el código	48
	Comprensión de los usos de los diagramas de actividades	49
	Uso de los símbolos de los diagramas de actividades	51
	Nodo inicial	52
	Flujo de control	52
	Acciones	56



Nodos de decisión y de fusión	62
Bifurcaciones y uniones de transición	63
Partición de la responsabilidad con carriles	63
Indicación de las señales cronometradas	67
Configuración de los parámetros de entrada	70
Forma de mostrar las excepciones en los diagramas de actividades	70
Terminación de los diagramas de actividades	71
Creación de los diagramas de actividades	72
Reingeniería del proceso	73
Reingeniería de una subactividad	74
Saber cuándo renunciar	77
Examen	77
Respuestas	79
CAPÍTULO 4	
Comportamientos con diagramas de interacción	81
Elementos de los diagramas de secuencia	82
Uso de las líneas de vida de objetos	83
Activación de una línea de vida	84
Envío de mensajes	85
Adición de restricciones y notas	87
Uso de marcos de interacción	87
Comprensión de lo que nos dicen las secuencias	91
Descubrimiento de objetos y mensajes	92
Elementos de los diagramas de colaboración (o comunicación)	94
Igualación del diseño con el código	96
Examen	97
Respuestas	99



CAPÍTULO 5	¿Cuáles son las cosas que describen mi problema?	101
	Elementos de los diagramas básicos de clase	102
	Comprensión de las clases y los objetos	103
	Modelado de relaciones en los diagramas de clases	112
	Estereotipado de las clases	117
	Uso de paquetes	118
	Uso de notas y comentarios	118
	Restricciones	118
	Modelado de primitivos	120
	Modelado de enumeraciones	121
	Indicación de espacios de nombres	122
	Cómo saber qué clases necesita	123
	Uso de un enfoque ingenuo	124
	Descubra otros beneficios del análisis de dominios	124
	Examen	128
	Respuestas	130
CAPÍTULO 6	Cómo se relacionan las clases	131
	Modelado de la herencia	132
	Uso de la herencia simple	132
	Uso de la herencia múltiple	135
	Modelado de la herencia de interfaces	139
	Boceto de diagrama	139
	Uso de la realización	140
	Descripción de la agregación y la composición	143
	Asociaciones y las clases asociaciones	145
	Examen de las relaciones de dependencia	150
	Adición de detalles a las clases	153



	Examen	153
	Respuestas	155
CAPÍTULO 7	Uso de los diagramas de esquemas de estado	157
	Elementos de un diagrama de estado	158
	Examen de los símbolos de estado	159
	Examen de las transiciones	164
	Creación de máquinas de estado de comportamiento	166
	Creación de máquinas de estado de protocolo	167
	Implementación de diagramas de estado	168
	Examen	172
	Respuestas	174
CAPÍTULO 8	Modelado de componentes	175
	Introducción del diseño basado en componentes	177
	Diseño componentes-interfaz	177
	Diseño a partir de las clases	177
	Modelado de un componente	178
	Especificación de las interfaces proporcionadas y requeridas	179
	Examen de los estilos de modelado de componentes	180
	Trazado de los diagramas de componentes para consumidores	180
	Trazado de los diagramas de componentes para productores	182
	Examen	183
	Respuestas	184
CAPÍTULO 9	Ajuste y finalización	185
	Modelado de los hacer y los no hacer	186
	No tenga esperando a los programadores	187



	Trabaje de una macrovista hacia una microvista	187
	Documento en forma económica	187
	Encuentre un editor	188
	Sea selectivo acerca de los diagramas que elige crear	188
	No dependa de la generación del código	188
	Modele y estructure disminuyendo el riesgo	188
	Si es obvio, no lo modele	189
	Haga hincapié en la especialización	189
	Uso de patrones de estado conocidos	189
	Refactorización de su modelo	192
	Modo de agregar documentación de soporte	192
	Validación de su modelo	193
	Examen	193
	Respuestas	195
CAPÍTULO 10	Visualización de su topología de despliegue	197
	Modelado de nodos	198
	Manera de mostrar artefactos en nodos	201
	Adición de trayectorias de comunicación	204
	Examen	206
	Respuestas	207
APÉNDICE A	Examen final	209
	Respuestas	223
	Bibliografía seleccionada	225
	Índice	227

RECONOCIMIENTOS



Bien entrada mi segunda década como escritor, tengo que agradecer a Wendy Rinaldi, de McGraw-Hill/Osborne, junto con Alexander McDonald y a mi agente David Fugate, de Waterside, por esta oportunidad de escribir lo que creo que el lector encontrará como un libro informativo, entretenido y fácil de seguir sobre el Unified Modeling Language.

También quiero manifestar mi agradecimiento a Eric Cotter, de Portland, Oregon, al ofrecerse para proporcionar la edición técnica para el Manual de UML. Eric realizó un trabajo excelente al hallar mis equivocaciones y omisiones, así como al mejorar las explicaciones.

Doy las gracias a mis anfitriones en el Ministry of Transportation Ontario, de St. Catharines, Ontario; colaborar con ustedes en el CIMS fue un proceso agradable y el examen de mis modelos y diseños con ustedes proporcionó una excelente base para este libro. Gracias también a Novica Kovacevic, Jennifer Fang, Rod, Marco Sánchez, Chris Chartrand, Sergey Khudoyarov, Dalibor Skacic, Michael Lam, Howard Bertrand y David He de Microsoft; fue un placer trabajar con y aprender de todos ustedes.

En 2004, junto con Bill Maas, Paul Emery, Sainney Drammeh, Bunmi Akin-yemichu y Ryan Doom, se formó el área Greater Lansing de .NET Users Group (glugnet.org) y me gustaría mandar un saludo a todos los grandes miembros y promotores de [glugnet](http://glugnet.org). Nos reunimos el tercer jueves de cada mes a las 6:00 P.M., en el bello campus de la Michigan State University. Gracias a la MSU por permitir el uso de sus excelentes instalaciones en el Engineering Building y el Anthony Hall.

Mientras estaba trabajando en Ontario, mi sustento me fue graciosamente suministrado en Prudhommes, en Vineland, Ontario, en las salidas 55 y 57, y en el Honest Lawyer, en St. Catharines, Ontario, Canadá. Gracias a Lis, Jen, Cheriton, Everett, Kathryn y Kim por los alimentos y la bebida para adultos, así como al personal del Honest Lawyer, por el acceso inalámbrico.

Por último, pero no porque sean los menos importantes, tengo una deuda de gratitud con mi esposa Lori y mis cuatro hijos, Trevor, Douglas, Alex y Noah, que representan el papel de mis más importantes admiradores y partidarios. Una familia es la más grande de las bendiciones. (También me gustaría presentar al miembro más reciente de nuestra familia, Leda, un eficiente laboratorio de chocolate, quien espera con paciencia a mis pies como un sutil recuerdo para empujarme de regreso a la computadora e ir a hacer algo más una que otra vez.)

INTRODUCCIÓN



A menudo, los nuevos inventos nacen sin necesidad y se documentan sobre servilletas mucho antes, si acaso, de que se proporcione una definición autorizada y formal. El Unified Modeling Language (UML) es precisamente uno de esos ejemplos. Los aspectos individuales de lo que al final se convirtió en el UML los definieron Ivar Jacobson, James Rumbaugh y Grady Booch, sin necesidad, mucho antes de que sus colaboraciones individuales se consolidaran en una sola definición.

Existe un problema mixto con las especificaciones formales y estándar. En general, para que un cuerpo augusto de científicos ratifique algo debe estar definido sin ambigüedad y con rigor. Si busca la definición del UML, encontrará metamodelos que describen hasta el más mínimo detalle lo que es y lo que no es. El efecto es muy semejante a leer informes del congreso: extensos, áridos, tediosos y con un poquito de jugo ocasional. Piense en las definiciones formales, en comparación con las aplicaciones prácticas, como esto: existen reglas rigurosas específicas que definen algo tan sencillo como el álgebra, pero usted no necesita conocerlas, aun cuando realizamos álgebra sencilla o nos apoyamos en ella en tareas cotidianas, como bombear gasolina. Por ejemplo, precio por litro multiplicado por el número de litros = precio total. Con una simple sustitución de texto por carácter, podemos crear ecuaciones aritméticas, $p * g = t$, que empiezan por parecerse a esas confusas ecuaciones de la escuela, pero que las hacen convenientes, desde el punto de vista rotacional, para determinar cualquier cantidad de ella. Lo que quiero decir es que incluso las personas que se identificarían como desafiadas por las matemáticas las aplican todos los días para fines prácticos, sin siquiera pensar que lo que están haciendo es resolver problemas matemáticos.

Ése es el objetivo de este libro. Hay definiciones formales y rigurosas del UML y existen por buenas razones, pero usted no necesita conocerlas para usar este lenguaje de una manera práctica. Los lingüistas del UML deben conocerlo en lo más íntimo, para definir con rigor, precisamente como los profesores de un idioma conocen la

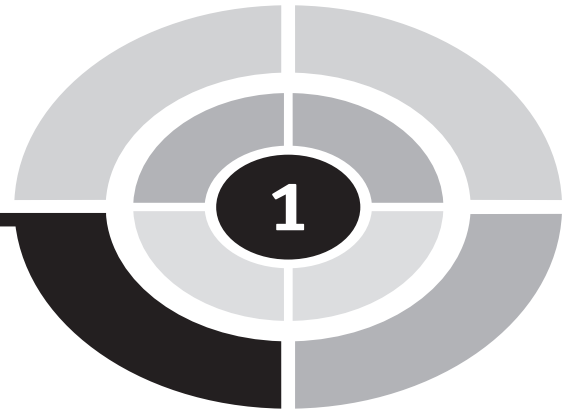


gramática hasta lo más profundo para poder enseñarlo, pero usted no necesita ser un profesor de su idioma para comunicarse con eficacia. Esto es verdad también para el UML; no necesita conocer todos los detalles acerca de él para usarlo con eficacia.

UML *DesMitificado* está escrito de manera sencilla y está diseñado para hacer que este lenguaje sea práctico, así como una herramienta eficaz para comunicar análisis y diseño de software.

Hay muchos libros sobre proceso, y el UML no define un proceso. Sin embargo, este libro está organizado de tal manera que si usted crea los tipos de modelos según se necesita, en el orden en el que aparecen en él, entonces puede contar con un inicio práctico de un proceso susceptible de usarse.

UML *DesMitificado* es un libro de tamaño modesto, pero es una recopilación de más de una docena de años de experiencia práctica trabajando con algunas de las mayores y mejor conocidas empresas del mundo, así como con muchas bien conocidas y no tan grandes empresas. El UML descrito en este libro es pragmático, práctico y aplicable, ya sea que usted se encuentre estructurando aplicaciones pequeñas, medianas o muy grandes. En pocas palabras, UML *DesMitificado* deja la pelusa y el rigor de torre de marfil a otros textos y le dice a usted lo que necesita saber para usar con éxito el UML al describir software.



CAPÍTULO

Una imagen vale más que mil líneas de código

Las imágenes de pequeñas personas formadas por palillos representan la forma de comunicación más antigua registrada en la historia humana. Algo de este arte rupestre se remonta a épocas tan antiguas como hace 75,000 años. Lo que resulta bastante extraño es que nos encontramos al principio del moderno siglo XXI y todavía estamos usando pequeñas figuras de línea para transmitir información. Eso es correcto; un pequeño hombre formado por palillos que llamamos Esaw es el carácter central en uno de los lenguajes más recientes desarrollado por los humanos (figura 1-1).





Figura 1-1 Esaw, a quien se menciona como actor en el UML.

El lenguaje acerca del cual estoy hablando se llama *Unified Modeling Language* (Lenguaje unificado de modelado), o UML. El UML es un lenguaje tanto como Pascal, C# (C sharp), el alemán, el inglés y el latín; y el UML posiblemente es uno de los lenguajes más recientes inventados por la humanidad, alrededor de 1997.

Como sucede con otros lenguajes, el UML fue inventado por necesidad. Es más, como con muchos lenguajes, en el UML se usan símbolos para transmitir significado. Sin embargo, a diferencia de los lenguajes orgánicos, como el inglés y el alemán, que evolucionan con el transcurso del tiempo a partir del uso común y la adaptación, el UML fue inventado por científicos, lo cual, por desgracia, es un problema. Los científicos son muy inteligentes pero con frecuencia no son muy buenos para explicar las cosas a aquellos menos científicos. Aquí es en donde intervengo.

En este capítulo, revisaremos el origen y la evolución del UML; también hablaremos acerca de cómo crear imágenes usando el UML, cuántas imágenes crear y qué tipos de ellas, qué deben transmitir esas imágenes y, lo más importante, cuándo suspender el dibujo de imágenes y empezar a escribir código.

Comprensión de los modelos

Un *modelo* es una colección de imágenes y texto que representa algo; para nuestros fines, software. (Los modelos no tienen que representar software, pero ahora reduciremos nuestro ámbito a los modelos de software.) Un modelo es para el software lo que un plano azul es para una casa.

Los modelos son valiosos por muchas razones específicas; en gran parte, constan de imágenes e, incluso, las imágenes simples pueden transmitir más información que una gran cantidad de texto; por ejemplo, código. Esto resulta coherente con el viejo adagio un tanto modificado de que una imagen expresa un millar de líneas de código. Los modelos son valiosos porque es más fácil dibujar algunas imágenes sencillas que escribir código o incluso texto que describan lo mismo. Los modelos son valiosos porque es más barato, rápido y fácil cambiar modelos que cambiar código. La verdad simple es que barato, rápido, fácil y flexible es lo que usted quiere cuando está resolviendo problemas.

Desafortunadamente, si cada uno usa imágenes diferentes para dar a entender lo mismo, entonces las imágenes se agregan a la confusión, en lugar de mitigarla. Aquí es en donde entra el UML.

Comprensión del UML

El UML es una definición oficial de un lenguaje pictórico con símbolos y relaciones comunes que tienen un significado común. Si todos los participantes hablan UML, entonces las imágenes tienen el mismo significado para todos aquellos que las observen. Por lo tanto, aprender UML es esencial para ser capaz de usar imágenes para experimentar barata, flexible y rápidamente con las soluciones.

Es importante reiterar aquí que es más rápido, más barato y más fácil resolver problemas con imágenes que con código. La única barrera para obtener beneficios del modelado es aprender el lenguaje del mismo.

El UML es un lenguaje precisamente como lo son el inglés o el afrikaans. El UML comprende símbolos y una gramática que define la manera en que se pueden usar estos símbolos. Aprenda los símbolos y la gramática, y sus imágenes serán comprensibles para todo aquel que reconozca estos símbolos y conozca la gramática.

Aunque, ¿por qué el UML? Usted podría usar cualesquiera símbolos y reglas con el fin de crear su propio lenguaje de modelado, pero el truco estaría en hacer que otros también lo usaran. Si sus aspiraciones son inventar un mejor lenguaje de modelado, entonces no me corresponde detenerlo. Debe saber que el UML se considera un estándar y que lo que este lenguaje es o no es lo define un consorcio de empresas que constituyen el Object Management Group (OMG, Grupo de Administración de Objetos). La especificación del UML está definida y ha sido publicada por el OMG en *www.omg.org*.

La evolución del diseño de software

Si siente que ha llegado tarde a la fiesta del UML, no se inquiete; en realidad, ha llegado temprano. La verdad es que el UML ha llegado tarde a la fiesta de desarrollo del software. Trabajo en todo Estados Unidos y converso con una gran cantidad de gente en muchas empresas muy grandes de software, y el UML y el modelado apenas están empezando a ponerse de moda. Esto queda ejemplificado de la mejor manera en las propias palabras de Bill Gates después de su famosa “semana de reflexión” en 2004, en donde se informa que habló acerca de la importancia creciente del análisis y diseño formales (léase UML) en el futuro. Este sentimiento lo apoya también la muy reciente compra de Visio, que incluye las capacidades de modelado de UML, por parte de Microsoft.

El UML representa una formalización del análisis y el diseño, y la formalización siempre parece llegar tarde. Considere los fabricantes de automóviles del siglo pasado. Al principio del siglo pasado, todos los fabricantes de coches en Flint, Michigan, estaban convirtiendo los carruajes ligeros tirados por un solo caballo en automóviles. Esto ocurrió mucho antes de que las grandes universidades, como la Michigan State University (MSU), graduaran ingenieros mecánicos capacitados para construir automóviles y herramientas de software, como programas para diseño con ayuda de computadora (CAD) que son especialmente buenos en el dibujo de artículos complejos, como las partes de los automóviles. La evolución de la ingeniería formalizada de los automóviles es consecuente con la evolución de la ingeniería formalizada del software.

Hace alrededor de 5000 años, los chinos crearon una de las primeras computadoras: el ábaco. Hace cerca de 150 años, Charles Babbage inventó una máquina mecánica de cálculo. En 1940, Alan Turing definió la máquina Turing de cálculo, y Presper Eckert y John Mauchly inventaron la Eniac. Después de las máquinas de cálculo, vinieron las tarjetas perforadas y el análisis y diseño estructurados de Grace Hopper para apoyar el desarrollo de Cobol. En la década de 1960, se inventó Smalltalk, un lenguaje orientado a objetos, y en 1986, Bjarne Stroustrup inventó lo que ahora se conoce como C++. No fue sino hasta alrededor de este mismo periodo —la década de 1980— cuando hombres muy inteligentes, como Ivar Jacobson, James Rumbaugh y Grady Booch, empezaron a definir los elementos del análisis y diseño modernos de software, lo que ahora llamamos el UML.

A finales de la década de 1980 y principios de la de 1990, las guerras sobre la notación del modelado estaban plenamente entabladas, con diferentes facciones apoyando a Jacobson, Rumbaugh o Booch. Recuerde, no fue sino hasta 1980 cuando la persona promedio pudo comprar y poseer una computadora personal (PC), y hacer algo útil con ella. Jacobson, Rumbaugh y Booch, cada uno por su lado, usaron símbolos y reglas diferentes para crear sus modelos. Finalmente, Rumbaugh y Booch empezaron a colaborar en relación con los elementos de sus respectivos lenguajes de modelado, y Jacobson se les unió en Rational Software.

A mediados de la década de 1990, se fusionaron los elementos de modelado de Rumbaugh [Object Modeling Technique (OMT, técnica de modelado de objetos)], Booch (método de Booch) y Jacobson (Objectory and Use Cases, cajas objetos y de usos) —a Rumbaugh, Jacobson y Booch se les mencionaba como “los tres amigos”— para formar el *proceso unificado de modelado*. Poco tiempo después, se eliminó *proceso* de la especificación del modelado y nació el UML. Esto ocurrió hace muy poco tiempo, apenas en 1997. La especificación UML 2.0 se estabilizó en octubre de 2004. Es correcto, ahora sólo estamos en la versión 2.

Esto lleva a la pregunta: ¿precisamente cuántas empresas están usando el UML y, en realidad, diseñando software con modelos? La respuesta es todavía muy pocas. Trabajo en toda Norteamérica y personalmente conozco ejecutivos en algunas empresas de software con mucho éxito, y cuando les pregunto si estructuran el software con UML, la respuesta es, casi siempre, no.

Si nadie está modelando, ¿por qué debe hacerlo usted?

Una persona racional podría preguntar: ¿por qué entonces, si Bill Gates está ganando miles de millones escribiendo software sin hacer un hincapié significativo en el modelado formal, debo preocuparme acerca del UML? La respuesta es que casi el 80% de todos los proyectos de software fallan. Estos proyectos sobrepasan sus presupuestos, no proporcionan las características que los clientes necesitan o desean o, lo que es peor, nunca se entregan.

La tendencia actual es llevar al exterior el desarrollo del software, hacia las naciones en desarrollo o del tercer mundo. La idea básica es que si los ingenieros estadounidenses especializados en software están fallando, entonces si se paga una quinta parte a un desarrollador euroasiático de software esto permitirá a las empresas intentar tener éxito con una frecuencia cinco veces mayor. ¿Qué están hallando estas empresas que están llevando el desarrollo hacia el exterior? Están descubriendo que Estados Unidos tiene algunos de los mejores talentos y recursos disponibles, y que la mano de obra barata en lugares alejados sólo introduce problemas adicionales y tampoco es garantía de éxito. La respuesta real que se necesita consumir más tiempo en el análisis y el diseño del software, y esto significa modelos.

Modelado y el futuro del desarrollo de software

Un énfasis creciente en el análisis y diseño formales no significa el fin del crecimiento de la industria del software; significa que los días del salvaje, salvaje oeste de las décadas de 1980 y 1990 llegarán al momento en que terminen; pero todavía está el salvaje, salvaje oeste de los hackers, allí en la tierra del software, y estará por algún tiempo.

Lo que un énfasis creciente en el análisis y diseño del software significa precisamente ahora es que los profesionales capacitados en UML tienen una oportunidad única para capitalizar este interés creciente en este lenguaje; también significa que, de manera gradual, menos proyectos fallarán, la calidad del software mejorará y se esperará que más ingenieros en software aprendan el UML.

Herramientas para modelado

Hasta hace muy poco, el modelado ha sido un cautivo en una torre de marfil rodeada por una guarnición impenetrable de científicos armados con metamodelos y herramientas para modelar ridículamente caras. El costo de una licencia para una herramienta popular para modelar estaba en los miles de dólares, lo que significó que el profesional promedio debía gastar por una aplicación para modelar tanto como lo que gastó por toda una computadora. Esto es ridículo.

Las herramientas para modelar pueden ser muy útiles, pero es posible modelar sobre trozos de papel. Por fortuna, usted no necesita ir tan lejos. La ame o la odie, Microsoft es muy buena para bajar el costo del software. Si tiene una copia de MSDN, entonces tiene una herramienta casi gratuita para modelar: Visio. Ésta es una buena herramienta, capaz de producir de manera competente modelos UML de alta calidad, y no le destrozará su presupuesto.¹

Para mantenernos en el tema de este libro —desmitificar UML—, en lugar de hacer saltar la banca en Together o Rose, usaremos el Visio de precio adecuado. Si el lector quiere usar Rose XDE, Together o algún otro producto, sea bienvenido para hacerlo, pero después de leer este libro verá que puede usar Visio y crear modelos profesionales, y ahorrarse cientos o incluso miles de dólares.

Uso de los modelos

Los modelos consisten en diagramas o imágenes. Lo que se intenta con los modelos es que sean más baratos para producir y experimentar que con el código. Sin embargo, si usted trabaja arduamente sobre qué modelos trazar, cuándo suspender el dibujo y empezar a codificar, o en si sus modelos son perfectos o no, entonces con lentitud observará reducirse el costo y el valor en tiempo de los modelos.

Puede usar texto llano para describir un sistema, pero se puede transmitir más información con imágenes. Podría seguir con ahínco la máxima de la eXtreme Programming (XP, programación extrema) y codificar, volviendo a descomponer en factores conforme avanza, pero los detalles de las líneas de código son mucho más complejos que las imágenes, y los programadores se adhieren al código pero no a las imágenes. (Yo no comprendo por completo la psicología de esta adhesión al código, pero en realidad existe. Sólo trate de criticar en forma constructiva el código de alguien más y observe cómo se deteriora la conversación con rapidez hasta llegar al insulto.) Esto significa que una vez que se escribe el código, es muy difícil obtener la aceptación de su codificador o de un administrador para hacerle modificaciones, en especial si el código se percibe para trabajar. Inversamente, la gente trabajará con mucho gusto de manera informal con los modelos y aceptará sugerencias.

Por último, debido a que en los modelos se usan símbolos sencillos, más personas interesadas pueden participar en el diseño del sistema. Muestre a un usuario final un centenar de líneas de código y escuchará el chillar de los grillos; muestre a ese usuario final un diagrama de actividades, y esa misma persona le dirá si ha captado usted la esencia de cómo se realiza correctamente esa tarea.

¹ Microsoft tiene un nuevo programa que le permite comprar MSDN Universal, el cual incluye Visio, por 375 dólares. Éste es un valor especialmente bueno.

Creación de diagramas

La primera regla de la creación de modelos es que el código y el texto consumen tiempo, y no queremos pasar una gran cantidad de tiempo creando documentos de texto que nadie leerá. Lo que sí queremos hacer es captar con exactitud las partes importantes del problema y una solución. Desafortunadamente, ésta no es una prescripción para el número o la diversidad de diagramas que necesitamos crear y no indica cuánto detalle necesitamos agregar a esos diagramas.

Hacia el final de este capítulo, en la sección “Hallar la línea final”, hablaré más acerca de cómo se sabe que se ha completado el modelado. En este momento, hablemos acerca de los tipos de diagramas que tal vez queramos crear.

Revisión de los tipos de diagramas

Existen varios tipos de diagramas que usted puede crear. Revisaré con rapidez los tipos de diagramas que puede crear y los tipos de información que se pretende transmitir con cada uno de estos diagramas.

Diagramas de casos de uso

Los *diagramas de casos de uso* son el equivalente del arte rupestre moderno. Los símbolos principales de un caso de uso son el *actor* (nuestro amigo Esaw) y el *óvalo del caso de uso* (figura 1-2).

Los diagramas de casos de uso son responsables principalmente de documentar los macrorrequisitos del sistema. Piense en los diagramas de casos de uso como la lista de las capacidades que debe proporcionar el sistema.

Diagramas de actividades

Un *diagrama de actividades* es la versión UML de un diagrama de flujo. Los diagramas de actividades se usan para analizar los procesos y, si es necesario, volver a realizar la ingeniería de los procesos (figura 1-3).

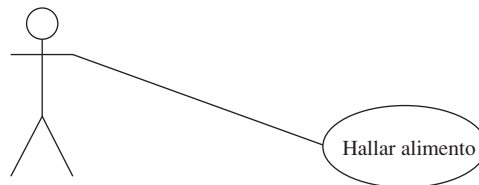


Figura 1-2 El caso de uso “Hallar alimento”.

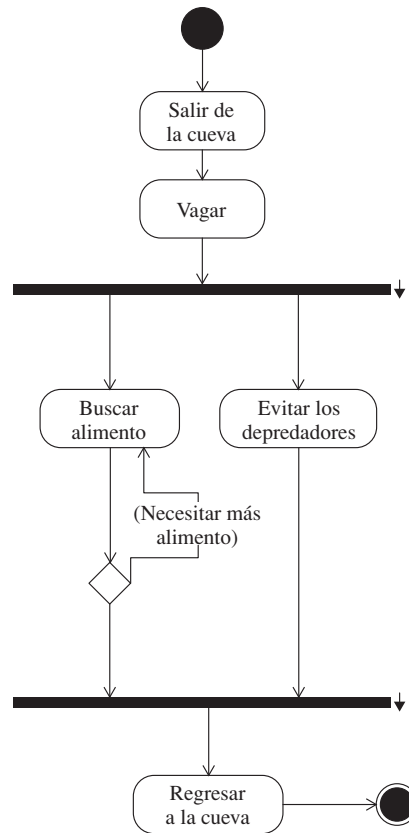


Figura 1-3 Un diagrama de actividades en el que se muestra la manera en que Esaw camina para hallar alimento.

Un diagrama de actividades es una herramienta excelente para analizar problemas que, al final, el sistema deberá resolver. Como una herramienta de análisis, no queremos empezar resolviendo el problema en un nivel técnico mediante la asignación de clases, pero podemos usar los diagramas de actividades para entender el problema e incluso refinar los procesos que comprenden el problema.

Diagramas de clases

Los *diagramas de clases* se usan para mostrar las clases de un sistema y las relaciones entre ellas (figura 1-4). Una sola clase puede mostrarse en más de un diagrama de clases y no es necesario mostrar todas las clases en un solo diagrama monolítico de clases. El mayor valor es mostrar las clases y sus relaciones desde varias perspectivas, de una manera que ayudará a transmitir la comprensión más útil.

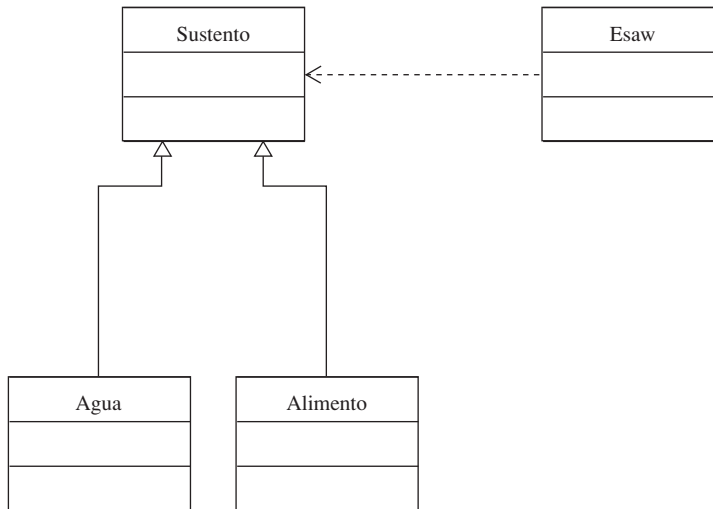


Figura 1-4 Un diagrama sencillo de clases, quizás uno de muchos, que transmite una faceta del sistema que se está diseñando.

Los diagramas de clases muestran una vista estática del sistema; no describen los comportamientos o cómo interactúan los ejemplos de las clases. Para describir los comportamientos y las interacciones entre los objetos de un sistema, podemos revisar los *diagramas de interacción*.

Diagramas de interacción

Existen dos tipos de diagramas de interacción: la *secuencia* y la *colaboración*. Ambos transmiten la misma información, empleando una perspectiva un poco diferente. Los diagramas de secuencia muestran las clases a lo largo de la parte superior y los mensajes enviados entre esas clases, modelando un solo flujo a través de los objetos del sistema. Los diagramas de colaboración usan las mismas clases y mensajes, pero organizados en una disposición espacial. La figura 1-5 muestra un ejemplo sencillo de diagrama de secuencia, y la 1-6 transmite la misma información con el uso de un diagrama de colaboración.

Un diagrama de secuencia implica un ordenamiento en el tiempo al seguir la secuencia de mensajes desde arriba a la izquierda hasta abajo a la derecha. Debido a que en el diagrama de colaboración no se indica en forma visual un ordenamiento en el tiempo, numeramos los mensajes para indicar el orden en el cual se presentan.

Algunas herramientas convertirán de manera automática los diagramas de interacción entre secuencia y colaboración, pero no es necesario crear los dos tipos de diagramas. En general, se percibe que un diagrama de secuencia es más fácil de leer y más común.

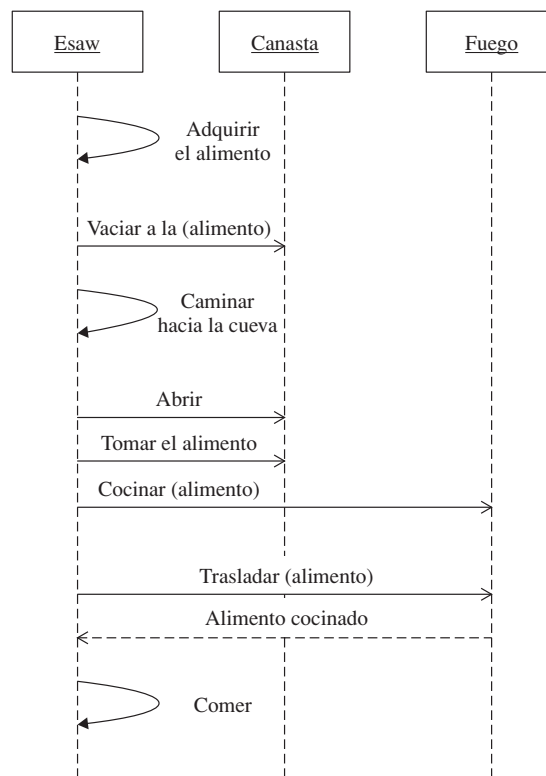


Figura 1-5 Diagrama sencillo de secuencia en el que se demuestra cómo se recoge y prepara el alimento.

Diagramas de estado

Mientras que los diagramas de interacción muestran los objetos y los mensajes que se pasan entre ellos, un *diagrama de estado* muestra el estado cambiante de un solo objeto, conforme éste pasa por un sistema. Si continuamos con nuestro ejemplo, entonces nos enfocaremos sobre Esaw y cómo está cambiando su estado a medida que busca con afán el alimento, lo encuentra y lo consume (figura 1-7).

RECUERDE Desmitificado: *el UML es un lenguaje. Como programar o hablar idiomas, si no se usan con frecuencia, se pueden olvidar un poco. Es perfectamente aceptable mejorar un idioma particular. La meta del modelado es captar la esencia del mismo y diseñar con pericia y, finalmente, con tanta exactitud como sea posible, sin quedarse atascado decidiendo acerca de los elementos del lenguaje. Desafortunadamente, las herramientas del UML no son tan exactas como los compiladores en la descripción de los errores del lenguaje.*

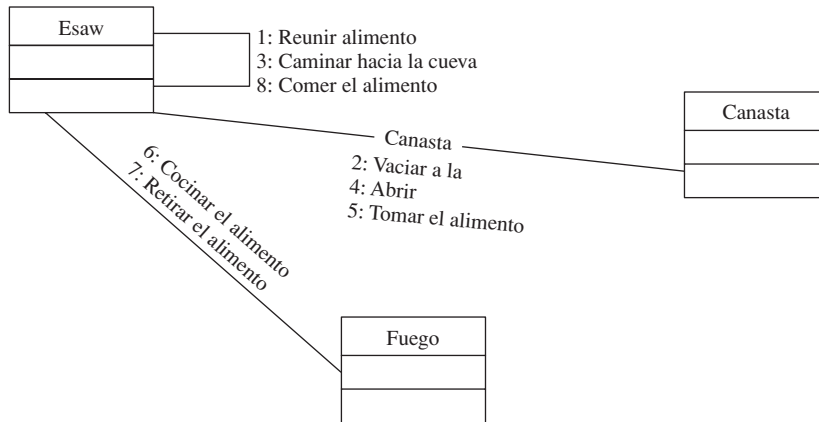


Figura 1-6 Diagrama de colaboración que transmite el mismo comportamiento de adquisición y consumo.

Diagramas de componentes

El UML define varios tipos de modelos, incluyendo modelos para análisis, para diseño y para implementación. Sin embargo, nada hay que le fuerce a crear o mantener tres modelos para una aplicación. Un ejemplo de un diagrama que podría encontrar en un modelo de implementación es de componentes. En un *diagrama de componentes*, éstos se muestran —piense en subsistemas— en el producto final.

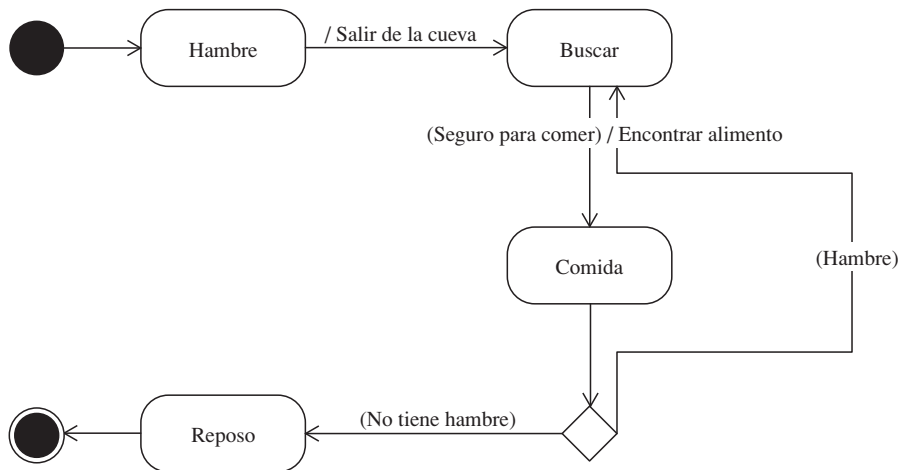


Figura 1-7 Diagrama de estado (o *esquema de estado*) que muestra el estado progresivo conforme Esaw busca con afán el alimento y come.

Cubriré los diagramas de despliegue más adelante en este libro, pero por ahora, aplazaré la cita de un ejemplo. En general, un diagrama de componentes es un poco semejante a uno de clases, con símbolos de componentes.

Otros diagramas

Hay otros tipos o variaciones de diagramas que podemos crear. Por ejemplo, un *diagrama de topología del despliegue* le mostrará cómo se verá desplegado su sistema. Lo común es que un diagrama de este tipo contenga símbolos que representen cosas, como servidores web, servidores de bases de datos y varios dispositivos diversos, así como software que constituye la solución de usted. Este tipo de diagrama es más común cuando usted está estructurando sistemas distribuidos en n hileras.

Más adelante, en este libro, le mostraré ejemplos de algunos de estos diagramas. Recuerde que, en el modelado, la clave consiste en modelar aspectos interesantes de su sistema que ayuden a aclarar elementos que puedan no ser obvios, en oposición a modelarlo todo.

Hallar la línea final

La parte más difícil del modelado es que es tan nuevo que los modelos UML están sujetos a algo de las mismas guerras de los lenguajes que sufrieron los proyectos orientados a objetos durante la última década. Le aliento a evitar estas guerras de lenguajes, ya que principalmente son ejercicios académicos improductivos. Si se encuentra colgado acerca de si algo es bueno o no en UML, entonces se está dirigiendo hacia la parálisis del análisis (y del diseño).

La meta es ser tan exacto como sea posible en una cantidad razonable de tiempo. El software mal diseñado es suficientemente malo, pero ningún software es casi siempre peor. Con el fin de determinar si ha concluido con un diagrama o modelo particular, haga la pregunta: *¿el diagrama o modelo transmite lo que entiendo, lo que quiero dar a entender y mi intención?* Es decir, ¿el diagrama o modelo es suficientemente bueno? La exactitud es importante porque los demás necesitan leer sus modelos, y los errores idiomáticos significan que esos modelos serán más difíciles de leer para los demás.

¿Cuántos diagramas debo crear?

No existe respuesta específica. Una pregunta mejor es: *¿debo crear todo tipo de diagramas?* La respuesta a esta pregunta es no. Un refinamiento de esta respuesta es que resulta útil crear diagramas que resuelvan los problemas delicados de análisis y diseño, así como diagramas que la gente realmente leerá.

¿Cuán grande debe ser un diagrama?

Determinar cuán grande necesita ser un modelo es otra buena cuestión para decidir. Si un modelo dado es demasiado grande, entonces puede aumentar la confusión. Intente crear modelos detallados, pero no demasiado. Como con la programación, la creación de modelos UML requiere práctica.

Solicite retroalimentación de diferentes grupos cuya opinión sea importante. Si los usuarios finales piensan que un diagrama de análisis capta de manera adecuada y correcta el problema, entonces siga adelante. Si los programadores pueden leer una secuencia y deducir cómo implementar esa secuencia, entonces siga adelante. Siempre puede agregar detalles, si debe hacerlo.

¿Cuánto texto debe complementar mis modelos?

Una idea fundamental del uso de imágenes para modelar, en lugar de texto muy confuso, es que las imágenes transmiten más significado en menos espacio y son más fáciles de manipular. Si agrega demasiado texto —restricciones, notas o documentos largos—, entonces está anulando la finalidad de esta notación pictórica más concisa.

El mejor lugar para el texto es el caso de uso. Un buen texto descriptivo en cada caso de uso puede aclarar con precisión qué característica apoya ese caso. En el capítulo 2 demostraré algunas buenas descripciones de los casos de uso.

Se recibirá bien que agregue cualquier texto aclaratorio que necesite, pero la regla general para el texto es análoga a la dada para los comentarios en código: sólo comente cosas que estén razonablemente sujetas a interpretación.

Por último, trate de documentar todo en su herramienta de modelado, en oposición a un documento separado. Si encuentra que usted necesita o el cliente requiere un panorama arquitectónico general escrito, aplase esto hasta después de que el software se haya producido.

Obtenga una segunda opinión

Si se encuentra atascado en un diagrama particular, obtenga una segunda opinión. Con frecuencia, dejar un diagrama a un lado durante un par de horas u obtener una segunda opinión le ayudará a resolver aspectos acerca de un modelo. Puede ser que halle que el usuario final de ese modelo entenderá lo que usted quiere decir o proporcionará más información que aclare la confusión, o bien un segundo par de ojos puede suministrar una respuesta lista. Un elemento crítico de todo software de desarrollo es construir cierta inercia y captar los macroconceptos, o grandes conceptos, sin quedarse atascado o mantener esperando a los usuarios.

Contraste de los lenguajes de modelado con el proceso

En realidad, el UML empezó su vida como Unified Process (Proceso unificado). Los inventores se dieron cuenta con rapidez de que los lenguajes de programación no determinan el proceso, ni debieran hacerlo los lenguajes de modelado. Por tanto, proceso y lenguaje se dividieron.

Existen muchos libros sobre procesos. No pienso que un proceso represente el mejor ajuste para todos los proyectos, pero quizás uno de los procesos más flexibles es el *Rational Unified Process* (Proceso unificado racional). Mi enfoque en este libro es sobre el UML, no sobre cualquier proceso particular. Estaré sugiriendo los tipos de modelos por crear y lo que le dicen a usted, pero le aliento a que examine los procesos de desarrollo por usted mismo. Considere examinar el Rational Unified Process (RUP), el proceso Agile, eXtreme Programming (XP) e, incluso, Microsoft's Services Oriented Architecture (SOA, arquitectura orientada a servicios de Microsoft). [SOA es más que un procedimiento arquitectónico en el que se usan elementos como XML Web Services (Servicios web XML), pero ofrece algunas buenas técnicas.]

No soy un experto en todos los procesos, pero enseguida doy un resumen que le dará un punto de partida. El RUP es un aparador de actividades centradas en el UML, que define macrofases iterativas en pequeñas cascadas, incluyendo iniciación, elaboración, construcción y transición. XP es hackeo constructivo. En general, la idea se basa en estructurar sobre lo que usted comprenda, esperar que las cosas cambien y usar técnicas como la programación de redes composición en factores y para apoyar los cambios a medida que usted aumenta su comprensión. El SOA de Microsoft depende de tecnologías como COM+, Remoting y los XML Web Services así como de una separación de las responsabilidades por medio de los servicios. Agile es una nueva metodología que no entiendo por completo, pero que en el libro del Dr. Boehm, *Balancing Agility and Discipline*, se le compara con XP, y sospecho que, desde el punto de vista conceptual, reside en alguna parte entre RUP y XP.

Es importante tener presente que muchas personas o entidades que le ofrecen un proceso pueden estar intentando venderle algo, y algunas muy buenas ideas tienen que venir de cada una de estas partes.

Examen

1. ¿Qué significa el acrónimo UML?
 - a. Uniform Model Language
 - b. Unified Modeling Language

- c. Unitarian Mock-Up Language
 - d. Unified Molding Language
2. El UML sólo se usa para modelar software.
 - a. Verdadero
 - b. Falso
3. ¿Cuál es el nombre del proceso más íntimamente asociado con el UML?
 - a. El proceso de modelado
 - b. El Rational Unified Process
 - c. eXtreme Programming
 - d. Los métodos Agile
4. ¿Cuál es el nombre del cuerpo de normas que define el UML?
 - a. Unified Modeling Group
 - b. Object Modeling Group
 - c. Object Management Group
 - d. Los cuatro amigos
5. Los diagramas de caso de uso se usan para captar las macrodescripciones de un sistema.
 - a. Verdadero
 - b. Falso
6. Los diagramas de secuencia son diferentes de los de colaboración (elijá todo lo que sea aplicable).
 - a. Los diagramas de secuencia son diagramas de interacción; los diagramas de colaboración no lo son.
 - b. Los diagramas de secuencia representan un ordenamiento en el tiempo, y los de colaboración representan clases y mensajes, pero no se implica el ordenamiento en el tiempo.
 - c. El orden en el tiempo se indica numerando los diagramas de secuencia.
 - d. Ninguna de las anteriores.
7. Un diagrama de clases es una visión dinámica de las clases de un sistema.
 - a. Verdadero
 - b. Falso
8. Un buen modelo UML contendrá por lo menos un diagrama de cada tipo.
 - a. Verdadero
 - b. Falso

9. ¿Cuál es el apodo del grupo de científicos más notablemente asociados con el UML?
 - a. La pandilla de los cuatro
 - b. Los tres mosqueteros
 - c. Los tres amigos
 - d. El dúo dinámico
10. Los diagramas de secuencia son buenos para mostrar el estado de un objeto a través de muchos casos de uso.
 - a. Verdadero
 - b. Falso

Respuestas

1. b
2. b
3. b
4. c
5. a
6. b
7. b
8. b
9. c
10. b

El principio con casos de uso

El Unified Modeling Language (UML) soporta el análisis y diseño orientados a objetos proporcionándole una manera de captar los resultados del análisis y el diseño. En general, iniciamos con la comprensión de nuestro problema; es decir, el análisis. Un tipo excelente de modelo para captar el análisis es el diagrama de casos de uso.

La finalidad de un *caso de uso* es describir la manera en que se usará un sistema: describir sus finalidades esenciales. La finalidad de los *diagramas* de casos de uso es captar en forma visual las finalidades esenciales.

Un caso de uso bien escrito y bien representado en diagrama es una de las clasificaciones de modelos individuales más importantes que usted puede crear. Esto es así porque expresar con claridad, conocer y organizar los objetivos es singularmente importante para alcanzarlos con éxito. Existe un viejo proverbio que dice: “Un viaje de mil millas empieza con un paso”, y existe un proverbio un poco menos antiguo que dice: “Si no sabe hacia adónde va, entonces el viaje nunca terminará.”

En este capítulo, hablaré acerca de una primera parte significativa de ese viaje —la creación de casos de uso— que cubrirá

- Los símbolos usados para crear los diagramas de casos de uso
- Cómo crear los diagramas de casos de uso
- Cuántos diagramas de casos de uso crear
- Cuánto incluir en un diagrama de casos de uso
- El nivel de detalle a incluir en un diagrama de casos de uso
- Cómo expresar las relaciones entre los casos de uso individuales
- La cantidad y el estilo de texto que es útil para hacer anotaciones en los diagramas de casos de uso
- De manera significativa, cómo establecer las prioridades de los casos de uso

Cómo hacer el caso para las casos de uso

Los diagramas de casos de uso parecen muy fáciles; constan de figuras de línea, líneas y óvalos. La figura de palillos se llama *actor* y representa a alguien o algo que actúa sobre el sistema. En el desarrollo de software, los actores son personas u otro software que actúa sobre el sistema. Las líneas son punteadas o continuas, con varias flechas o sin ellas, que indican la relación entre el actor y los óvalos. Estos últimos son los casos de uso y, en el diagrama de casos de uso, los óvalos tienen algún texto que proporciona una descripción básica. La figura 2-1 es un ejemplo sencillo de un diagrama de casos de uso.

Durante mucho tiempo los diagramas de casos de uso me fastidiaron porque parecían demasiado sencillos para tener algún valor. Un niño de tres o cuatro años con un crayón y un pedazo de papel podría reproducir estas figuras de línea; sin embargo, su sencillez es decepcionante.

Que un diagrama de casos de uso sea fácil de crear es un elogio implícito para el UML. Hallar los casos de uso correctos y registrar sus responsabilidades en forma correcta es la decepción. Hallar los casos de uso correctos y describirlos de manera adecuada es el proceso crítico que impide que los listos ingenieros de software pasen por alto necesidades

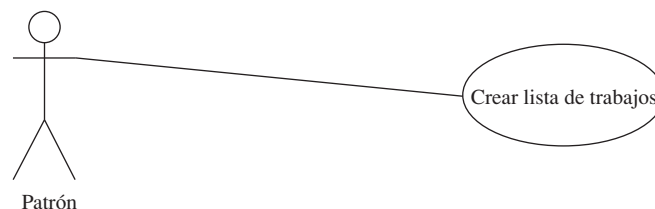


Figura 2-1 Un diagrama de casos de uso muy sencillo.

críticas y que inventen de manera innecesaria. En pocas palabras, los diagramas de casos de uso constituyen un macrorregistro de lo que usted quiere estructurar.

En el párrafo anterior, usé el prefijo *macro*. Macro en este contexto sencillamente significa “grande”. Los grandes objetivos, o macroobjetivos, son los que se mencionan como los argumentos, o razones, poderosos de la empresa para hacer algo. En los diagramas de casos de uso se captan los objetivos grandes, poderosos. En el texto de esos casos se captan los detalles de apoyo.

Esto es lo que se me escapaba en las imágenes de las figuras de línea de los diagramas de casos de uso; perdía de vista que, sencillamente, al registrar lo que el sistema hará y lo que no hará, registramos y especificamos el alcance de lo que se está creando; también perdía de vista que el texto que acompaña los diagramas de casos de uso rellena los espacios en blanco entre los macroúsos y los microúsos, en donde *micro* significa usos “menores, de apoyo”.

Además de registrar los usos primarios y secundarios, los diagramas de casos de uso nos proporcionan en forma implícita varias oportunidades significativas para administrar el desarrollo, a lo cual entraré con más detalle a medida que avance el capítulo.

Establecimiento de prioridad de las capacidades

¿Alguna vez ha escrito una lista de cosas por hacer? Una *lista de cosas* por hacer es una lista de cosas que usted debe hacer o desea hacer. El acto de escribir la lista es un punto de partida. En esencia, los casos de uso son listas de cosas por hacer. Una vez que ha captado los casos de uso, ha articulado lo que el sistema hará, y puede usar la lista para dar prioridades a nuestras tareas. Tanto enunciar como organizar los objetivos son primeras tareas muy críticas.

El valor de establecer prioridades para las capacidades de un sistema es que el software es fluido. Permítame ilustrar, por medio de un ejemplo, lo que quiero decir. Es posible crear, guardar, abrir e imprimir un documento de texto tanto con Notepad (Bloc de notas) como con Word de Microsoft, pero la diferencia en el número de líneas de código y el número de características entre estos dos programas es tremenda. Al establecer prioridades de los usos, con frecuencia tenemos la oportunidad de hacer, con ventaja, malabarismos con las características, el presupuesto y el programa.

Suponga, por ejemplo, que mis objetivos primarios son ser capaz de crear, guardar, abrir e imprimir un documento de texto. Suponga además que mis objetivos secundarios son guardar el documento como texto llano, HyperText Markup Language (HTML, lenguaje de marcado de hipertexto), y como texto enriquecido, es decir, formateo especial. Establecer prioridades de las capacidades significa que podría elegir enfocarme hacia los usos primarios —crear, guardar, abrir e imprimir—, pero aplazar el soporte de HTML y texto enriquecido. (Las características en el software por lo común se aplazan hacia versiones posteriores, debido a las restricciones reales mencionadas con anterioridad, incluyendo tiempo, presupuesto y un cambio en el entorno de la empresa.)

No tener tiempo suficiente y quedarse sin dinero son problemas directos. Los desarrolladores de software son rutinariamente optimistas, se distraen en salidas por la tangente y pasan más tiempo en reuniones que en la planeación, y estas cosas gravan un presupuesto. Sin embargo, tomemos un momento para examinar un cambio en el entorno de la empresa. Si nuestras necesidades originales fueron HTML, texto llano y texto enriquecido y hemos estado estructurando nuestro software en los últimos cinco años, resultaría perfectamente plausible que un cliente dijera, a la mitad del curso del desarrollo, que guardar un documento como eXtensible Markup Language (XML, lenguaje ampliable de marcado) sería más valioso que como texto enriquecido. De este modo, debido a un clima tecnológico en evolución, a media corriente un cliente podría restablecer las prioridades y demandar XML como más importante que el texto enriquecido. Si no hubiéramos documentado nuestras necesidades primarias y secundarias, entonces podría ser un reto muy grande determinar los trueques deseables, como cambiar el texto enriquecido por el XML. Debido a que registramos con claridad los casos deseables de usos, podemos establecer prioridades y hacer trueques valiosos, si es necesario.

Comunicación con los no tecnófilos

Otra cosa que no perdí de vista acerca de los casos de uso es que su mera sencillez los hace un medio fácil de transmisión para comunicarse con no tecnófilos. A estas personas las llamamos *usuarios o clientes*.

Los programadores que usan el hemisferio izquierdo de su cerebro en general detestan a los usuarios. La idea básica es que si uno no puede leer el código, entonces es tonto o, por lo menos, más tonto que aquellos que sí pueden. El UML y los casos de uso cubren la brecha entre los programadores que usan el hemisferio izquierdo del cerebro y los usuarios no tecnófilos.

Una figura de palillos, una línea y un óvalo son suficientemente simplistas, cuando se combinan con algún texto, para que todos los participantes puedan entender el significado. El resultado es que los usuarios y clientes pueden observar los dibujos y leer el texto llano, y determinar si los tecnólogos han, o no, registrado con exactitud y comprendido las características deseables. Esto también significa que los administradores —quienes pueden no haber escrito código en 10 años— y las direcciones técnicas pueden examinar el producto final y, por inspección, garantizar que la inventiva desenfadada no es la causa de los programas no cumplidos ni de las características ausentes. Demostrando esta disonancia al continuar con mi primer ejemplo, suponga que, de cualquier manera, se implementa el soporte de texto enriquecido porque el programador sabe cómo almacenar y recuperar ese tipo de texto. No obstante, debido a que el XML es más reciente y el programador tiene menos experiencia en trabajar con él, la característica de escritura en XML se aplaza sin malicia. Un administrador proactivo puede descubrir las necesidades de un cliente, según se captan mediante los casos de uso, y apropiarse de salidas por la tangente improductivas.

Debido a que los casos de uso son visuales y sencillos, los usuarios y clientes pueden suministrar retroalimentación, y las personas que constituyen el puente entre los clientes y los programadores, como los administradores, pueden determinar si las características que en realidad se estructuraron reflejan con exactitud los deseos de los usuarios.

Uso de los símbolos de los casos de uso

Los diagramas básicos de casos de uso constan de sólo unos cuantos símbolos: el *actor*, un *conector* y el *óvalo del caso de uso* (figura 2-2). Tomemos unos cuantos minutos para hablar de cómo se usan estos símbolos y qué información transmiten.

Símbolos de actores

La figura de palillos, mencionada como *actor*, representa participantes en los casos de uso. Los actores pueden ser personas o cosas. Si un actor es una persona, entonces, en realidad, nunca se puede representar por medio de un código. Si un actor es otro subsistema, entonces se le puede observar como una clase o subprograma, pero todavía representarse usando el símbolo de actor en los diagramas de casos de uso.

Los actores se descubren como resultado del análisis. Conforme vaya identificando los macrouso del sistema, identificará quiénes son los participantes para esos casos de uso. En principio, registre cada actor a medida que se descubre, agregando un símbolo de actor a su modelo y describiendo cuál es su papel. Nos preocuparemos acerca de la organización y el refinamiento más adelante, en la sección titulada “Creación de los diagramas de casos de uso”.

Casos de uso

El símbolo del caso de uso se utiliza para representar capacidades. Al caso de uso se le da un nombre y una descripción mediante un texto. Este último debe describir cómo inicia y finaliza el caso de uso, e incluye una descripción de la capacidad descrita por el nombre de la misma, así como escenarios de apoyo y requisitos no funcionales. En la sección titulada “Creación de los diagramas de casos de uso”, examinaremos ejemplos

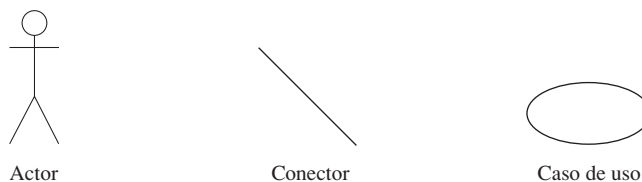


Figura 2-2 Los símbolos básicos de los diagramas incluyen al actor, al conector y al óvalo del caso de uso.

de nombres de casos de uso, y en la sección titulada “Documentación de un caso de uso utilizando un borrador”, proporcionaré un borrador modelo que pueda usar para ayudarse a escribir las descripciones de los casos de uso.

Conectores

Dado que los diagramas de casos de uso tienen múltiples actores y en virtud de que los casos de uso pueden estar asociados con los actores y con otros casos de uso, se utilizan los conectores para indicar la manera en que ambos están asociados. Además, los estilos de conectores pueden cambiar para transmitir más información acerca de la relación entre los actores y los casos de uso. Por último, los conectores pueden tener adornos y anotaciones que suministran incluso más información.

Estilos de líneas para los conectores

Existen tres estilos básicos de líneas para los conectores. Un conector de línea simple se llama *asociación* y se usa para mostrar cuáles actores están relacionados con cuáles casos de uso. Por ejemplo, en la figura 2-1 se mostró que un patrón está asociado con el caso de uso “Crear lista de trabajos”.

Un segundo estilo de conector es una línea punteada con una flecha direccional (figura 2-3). Este estilo de conector se conoce como *dependencia*. La flecha apunta hacia el caso de uso del que depende. Por ejemplo, suponga que a los patrones de *www.motown-jobs.com* se les debe dar acceso para crear una lista de trabajos. Entonces podemos decir que el caso de uso “Crear lista de trabajo” depende de un caso de uso “Entrar”. Ésta es la relación que se ilustra en la figura 2-3.

Un tercer estilo de conector es una línea dirigida con un triángulo hueco, al cual se le conoce como *generalización*. La palabra *generalización* en el UML significa “herencia”. Cuando mostramos una relación de generalización entre dos actores o dos casos de uso, estamos indicando que el actor o el caso de uso “hijos” son un caso del actor o uso básico y algo más. En la figura 2-4, se muestra una relación de generalización entre dos actores y dos casos de uso.



Figura 2-3 El caso de uso “Crear lista de trabajos” depende de que el patrón obtenga acceso.

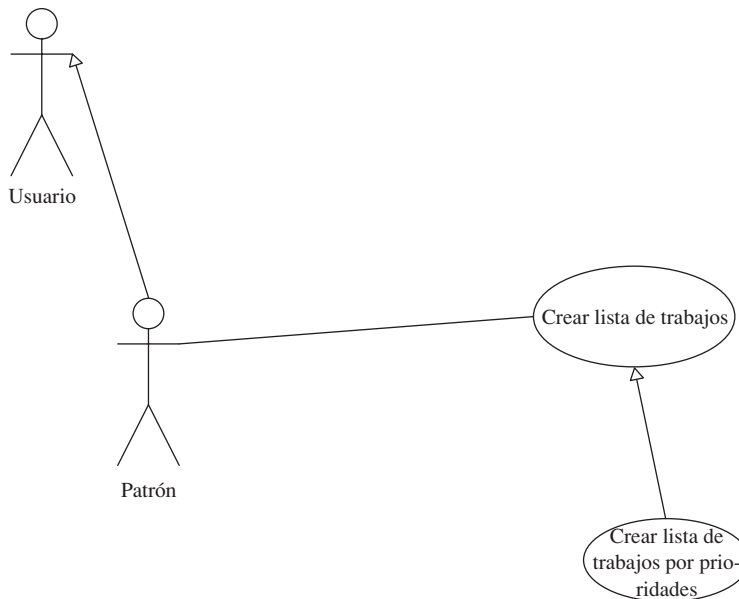


Figura 2-4 Diagrama de casos de uso en el que se muestran dos relaciones de generalización entre dos actores y entre dos casos de uso.

En las relaciones de generalización, la flecha apunta hacia la cosa sobre la cual nos estamos expandiendo. Existen varias maneras en las que usted puede describir esta relación en forma verbal —acerca de la cual usted debe saber—, pero desafortunadamente, todos estos sinónimos pueden conducir a confusión verbal. Los siguientes enunciados describen las relaciones de generalización que se muestran en la figura 2-4:

- El usuario es el objetivo y el patrón es la fuente.
- El patrón es un usuario.
- El usuario es el subtipo y el patrón es el supertipo.
- El patrón se hereda del usuario.
- El usuario es el tipo padre y el patrón es el tipo hijo.
- El patrón generaliza al usuario.

(En esta lista, puede sustituir la frase *Crear lista de trabajos* en todas partes en donde vea la palabra *Usuario*, y sustituir la frase *Crear lista de trabajos por prioridades* en todas las partes en donde vea la palabra *Patrón*, para transmitir la relación entre los dos casos de uso.) El último enunciado, en el cual se usa la palabra *generaliza*, es el más exacto en el contexto del UML, pero vale la pena reconocer que todos los enunciados son equivalentes.

Adornos de los conectores

Los diagramas UML fomentan el uso de menos texto porque las imágenes transmiten una gran cantidad de información a través de una conveniente taquigrafía visual, pero los diagramas UML no se abstienen por completo del texto; por ejemplo, los conectores pueden incluir texto que indique multiplicidad de los puntos extremos y texto que estereotipa el conector.

Manera de mostrar la multiplicidad

En general, los conectores pueden tener notaciones de multiplicidad en cualquiera de sus dos extremos. Las notaciones de multiplicidad indican el conteo posible de cada cosa. Por ejemplo, un asterisco significa muchos; un asterisco próximo a un actor significa que puede haber muchos ejemplos de ese actor. Aun cuando el UML permite hacer anotaciones de esta manera en los conectores de caso de uso, eso no es muy común. Es más probable que usted vea estas marcas de notación de conteo en diagramas del tipo de los de clase, de modo que daré detalles sobre la multiplicidad en el capítulo 3.

Estereotipado de los conectores

Una notación más común en los conectores es el estereotipo. Los estereotipos agregan detalles a la relación entre los elementos en un diagrama de caso de uso. Por ejemplo, en la figura 2-3, introduje el conector de dependencia. Se puede usar un estereotipo para ampliar el significado de este conector.

En la sección titulada “Estilos de líneas para los conectores”, dije que un patrón puede crear una lista de trabajos e ilustré esto con un actor patrón, un caso de uso “Crear lista de trabajos” y un conector de asociación; sin embargo, también dije que el patrón debe obtener acceso. Cuando un caso de uso —“Crear lista de trabajos”— necesita los servicios de otro caso de uso —“Entrar”— entonces se dice que el caso de uso dependiente *incluye* el caso de uso del que depende. (En código, una relación incluir se implementa como reutilización de código.)

Un estereotipo se muestra como texto entre los caracteres « y » (comillas angulares). Por ejemplo, si decimos que “Crear lista de trabajos” incluye a “Entrar”, entonces podemos representar un estereotipo *incluir* colocando una anotación en el conector de dependencia como se muestra en la figura 2-5.



Figura 2-5 Ejemplo de un estereotipo incluir —usado para representar *reutilizar*— en la dependencia entre “Crear lista de trabajos” y “Entrar”.

Incluir y extender son conceptos importantes en los diagramas de caso de uso, de modo que enseguida ampliaré lo relativo a estos temas.

NOTA *Estereotipo es un concepto generalmente útil en el UML. La razón de esto es que es permisible que usted introduzca y defina sus propios estereotipos. De esta manera, puede extender el UML.*

Caso de uso de inclusión y de extensión

Una relación de dependencia entre dos casos de uso significa que, de alguna manera, el caso dependiente necesita al caso del que depende. Dos estereotipos de uso común y predefinidos que refinan las dependencias en los casos de uso son el incluir y el extender. Tomemos un minuto para ampliar nuestros comentarios de introducción sobre incluir, de la sección anterior, e introduzcamos extender.

SUGERENCIA *Visio aplica un estereotipo extender en el conector de generalización para dar a entender herencia. Existen variaciones entre el UML y las herramientas del mismo, porque el UML es un estándar en evolución y la implementación de las herramientas puede ir adelante o atrás de la definición oficial del UML.*

Más sobre los estereotipos incluir

Una dependencia rotulada con el estereotipo incluir significa que, finalmente, el caso de uso dependiente es para volver a usar el caso del que depende. El equipaje que va con el estereotipo incluir es que el caso de uso dependiente necesitará los servicios del caso del que depende y saber algo acerca de la realización de ésta, pero lo opuesto no es cierto. El caso de uso del que se depende es una entidad completa y distinta que no debe depender del caso dependiente. La concesión de acceso es un buen ejemplo. Resulta claro que requerimos que un patrón tenga acceso para crear una lista de trabajos, pero también pudimos obtener acceso por otras razones.

NOTA *En una dependencia incluir entre casos de uso, el caso dependiente también se conoce como el caso de uso básico, y aquella de la que se depende también se conoce como el caso de uso de inclusión. Aunque básico y de inclusión pueden ser términos más precisos, no parece que se empleen de manera común al hablar.*

Poner tanto significado en una pequeña palabra como *incluir* es la razón por la cual el UML puede transmitir una gran cantidad de significado en un diagrama sencillo, pero también es la razón por la cual los modelos UML pueden representar un reto para crearse y leerse. Una estrategia real a la que puede recurrir es agregar una nota en donde no esté

seguro acerca del uso de algún aspecto idiomático del UML (vea, más adelante, “Anotaciones en los diagramas de caso de uso”). Por ejemplo, si quiere describir la relación entre “Crear lista de trabajos” y “Entrar”, pero no está seguro acerca de cuál conector o cuál estereotipo usar, entonces podría usar una asociación simple y una nota asociada al conector que describa en texto llano lo que usted quiere dar a entender. La nota puede actuar como un recordatorio para, más adelante, regresar y buscar el UML preciso.

Uso de los estereotipos extender

El estereotipo extender se usa para agregar más detalle a una dependencia, lo cual significa que estamos agregando más capacidades (como ejemplo, vea la figura 2-6). Como se muestra en la figura, decimos que “Registrar las listas vistas” extiende (y depende de) “Ver lista”.

NOTA En una relación extendida, la flecha apunta hacia el caso de uso básico y el otro extremo se conoce como el caso de uso de extensión.

En la sección anterior, no permitiríamos que un patrón creara una lista de trabajos sin registrarse, pero en el caso del registro del caso de uso entrar es indiferente para la reutilización del caso. En esta sección, a el caso de uso ver lista no le importa que la estén registrando; en otras palabras, la característica registrar necesitará saber acerca de la característica ver lista, pero no en sentido contrario.

Una perspectiva valiosa aquí es quién podría estar interesado en el registro. Es evidente que al “Solicitante de trabajo” tal vez no le interese cuántas veces se ha visto la lista, pero un patrón previsor podría estar interesado en cuánto tráfico está generando su lista. Pasemos ahora por un momento a un dominio diferente. Suponga que el “Solicitante de trabajo” fuera el comprador de una casa y que la lista lo fuera de residencias. Ahora tanto el comprador como el vendedor podrían estar interesados en el número de veces que se ha visto la propiedad. Una casa que ha estado en el mercado durante meses puede tener

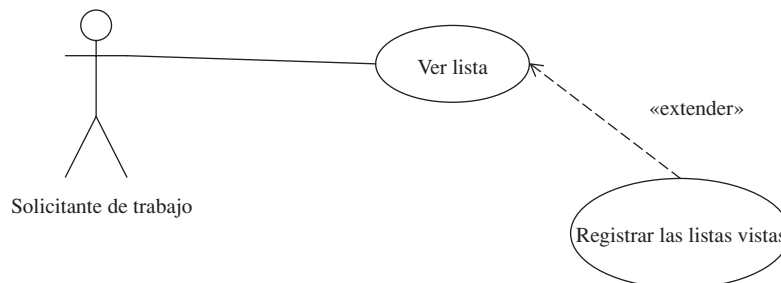


Figura 2-6 Seguir el rastro del número de veces que se ve una lista de trabajos es una extensión de “Ver lista”, como se describe mediante la dependencia y el estereotipo extender.

problemas. Sin embargo, en los dos escenarios, la lista es lo más importante y el número de veces que se ha visto es secundario. Esto ilustra la noción de caso de uso de extensión como parecidas a las características y, desde una perspectiva de mercadeo, las extensiones podrían ser elementos que estén separados en un paquete opcional de características.

SUGERENCIA *Considere la alternativa, ya que se relaciona con un caso de uso de extensión. Los casos de uso de extensión son características secundarias naturales. Si su proyecto tiene un programa apretado, lleve hasta el final los casos de uso de extensión, y, si su tiempo se agota, entonces posponga los casos de uso de extensión para una versión posterior.*

Incluir y extender parecen algo semejantes, pero la mejor manera de tenerlos en orden es recordar que “la relación incluir es para volver a aplicar el comportamiento modelado por otro caso de uso, en tanto que la relación extender es para agregar partes a caso de uso existentes así como para modelar servicios opcionales del sistema” (Övergaard y Palmkvist, 2005, p. 79).

Anotaciones en los diagramas de casos de uso

Considere el trabajo de un estenógrafo en un juicio. Los estenógrafos usan esas graciosas máquinas estenográficas de escribir que producen una suerte de majaderías taquigráficas. Podemos suponer con seguridad que si una máquina de escribir común o un procesador de palabras pudiera aceptar una entrada suficientemente rápida como para mantenerse al ritmo del habla natural, entonces el estenógrafo nunca se hubiera inventado.

Los estenógrafos producen una taquigrafía que es más condensada que el discurso al hablar. El UML es como la taquigrafía para el código y el texto, y las herramientas de modelado del UML son semejantes a los estenógrafos. La idea es que los modelos se puedan crear más rápido que el código o más rápido que escribir descripciones en forma de texto. Dicho eso, a veces no hay un buen sustituto para el texto.

Si se encuentra en el predicamento de que sólo el texto parece resolver —o no está seguro del UML—, entonces siga adelante y agregue texto. Puede agregar texto mediante la documentación de sus modelos con características de la mayoría de las herramientas de modelado, agregando referencias URL a los documentos más verbosos o agregando notas directamente en los propios diagramas. Sin embargo, si agrega demasiado texto, entonces de manera natural, tardará más en completar el modelado y puede ser que se requiera un esfuerzo mayor para entender el significado de cada uno de los diagramas.

Inserción de notas

El UML es una taquigrafía para una gran cantidad de texto y de código, pero si lo necesita, siempre puede agregar texto. Todos los diagramas, incluyendo los casos de uso, permi-

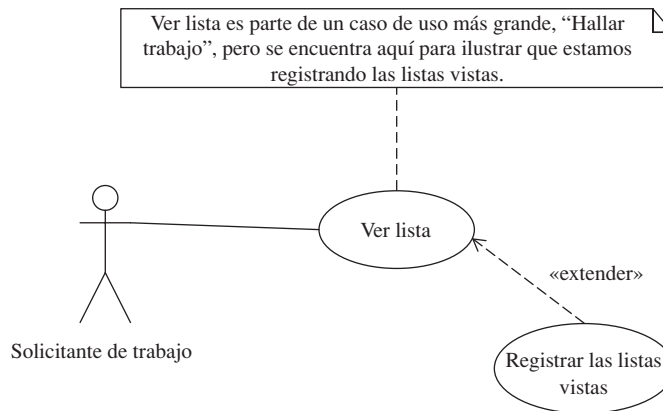


Figura 2-7 Nota que agrega texto llano para aclarar algún aspecto de un diagrama.

ten que se les agreguen anotaciones en forma de texto. Las notas se representan como un trozo de papel con una punta doblada y una línea que une el cuadro de texto al elemento que se le está haciendo la anotación (figura 2-7). Use las notas con moderación, porque pueden abarrotar un diagrama y hacerlo difícil de leer.

Modo de agregar documentación de soporte

Todas las herramientas de modelado que he usado —Together, Rose, Rose XDE, Visio, Poseidon para UML y la de Cayenne Software— permiten la documentación del modelo. Por lo común, esta documentación toma dos formas: texto que se almacena en el modelo y Uniform Resource Locators (URL, localizadores uniformes de recursos), que hacen referencia a documentos externos (figura 2-8). El examen de las características de su herramienta particular le descubrirá estas capacidades.

Más importante es qué tipo de documentación debe usted proporcionar. De manera subjetiva, la respuesta es tan pequeña como aquella con la que pueda ponerse en marcha, pero en general, los diagramas de caso de uso parecen necesitar lo máximo.

Los diagramas de caso de uso son bastante básicos con sus figuras de línea, pero son bastante importantes porque registran las capacidades que tendrá el sistema de usted. Una buena información a incluir con sus diagramas de caso de uso es

- Un párrafo conciso en el que se describa cómo empieza el uso, incluyendo cualesquiera condiciones previas
- Un párrafo corto para cada una de las funciones primarias
- Un párrafo corto para cada una de las funciones secundarias
- Un párrafo corto para cada uno de los escenarios primarios y secundarios, los cuales ayuden a ubicar en un contexto la necesidad de las funciones

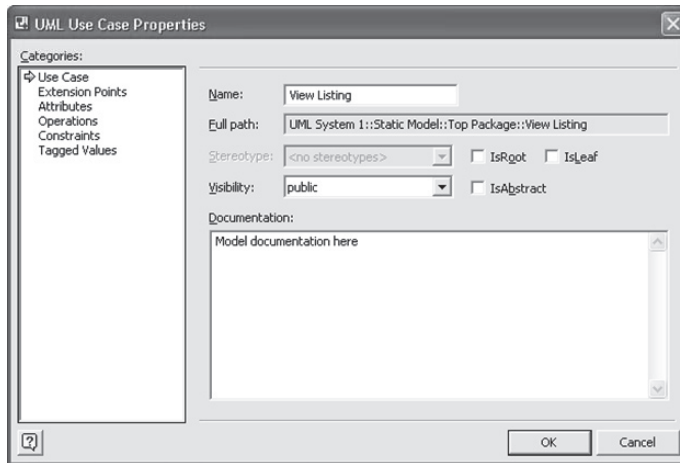


Figura 2-8 Mediante un doble clic sobre un elemento de modelo en Visio, puede añadir documentación que está agregada en el modelo.

- Un párrafo para las necesidades no funcionales
- Puntos de inserción en donde se usen cualesquiera otros casos de uso dependientes
- Un punto de finalización con las condiciones posteriores

Todos estos elementos suenan como una gran cantidad de trabajo, y pueden serlo. Sin embargo, recuerde que los casos de uso son los fundamentos del análisis, y es importante que las documente tan cuidadosa y completamente como pueda. De igual importancia es notar que usé las palabras *conciso* y *corto* de manera intencional. Por corto, quiero dar a entender que es aceptable tener párrafos de una sola oración.

Puede usar cualquier formato que le guste para documentar sus casos de uso. Si se siente cómodo con el formato de borrador, es muy fácil crear un borrador modelo con base en la lista con viñetas que acabo de dar. Una buena práctica es elegir un estilo para su documentación y adherirse a él.

Tomemos un momento para explicar en detalle los elementos —según se describen en la anterior lista con viñetas— de la documentación del caso de uso. Tenga presente que esto no es una ciencia exacta y que su documentación de los casos de uso no necesita ser perfecta.

Documentación de un caso de uso utilizando un borrador

Puede usar texto de forma libre para documentar un caso de uso, pero encuentro que un modelo de borrador sugiere la extensión de la información y actúa como un recordatorio de los elementos necesarios para documentar cada caso en forma adecuada. A continua-

ción se presenta un modelo que incluye una breve descripción y un ejemplo para cada sección. Vale la pena hacer notar que este estilo de documentación no es parte del UML, pero es un elemento útil del modelado.

1. Título

- a. Descripción: Use aquí el nombre del caso de uso, pues facilita mucho el acoplamiento de los diagramas de caso de uso con su documentación respectiva.
- b. Ejemplo: Mantener lista de trabajos.

2. Inicios del caso de uso

- a. Descripción: Describa con brevedad las circunstancias que llevan al caso de uso, incluyendo las condiciones previas. Deje fuera los detalles de la implementación, como “El usuario hace clic en un hipervínculo”, o las referencias a las formas, los controles o los detalles específicos de la implementación.
- b. Ejemplo: Este caso de uso se inicia cuando un patrón, un agente de un patrón o el sistema quiere crear, modificar o eliminar una lista de trabajos.

3. Funciones primarias

- a. Descripción: Los casos de uso no son necesariamente singulares. Por ejemplo, “Administrar la lista de trabajos” es un caso de uso razonable y puede incluir funciones primarias como leer un recipiente o escribir en él. La clave aquí es evitar demasiado pocas o demasiadas funciones primarias. Si necesita una buena medida, podrían ser dos o tres funciones primarias por caso de uso.
- b. Ejemplo: “CLAB la lista de trabajos.” Las funciones primarias de “Mantener la lista de trabajos” son crear, leer, actualizar y borrar la lista de trabajos.

4. Funciones secundarias

- a. Descripción: Las funciones secundarias son como un reparto de apoyo en una pieza teatral. Por ejemplo, dado un caso de uso “Administrar la lista de trabajos”, actualizar, insertar, crear y borrar una lista de trabajos —llamado *CLAB* por crear, leer, actualizar y borrar— son excelentes funciones secundarias, parte de un caso de uso más grande. Si necesita una medida, entonces el doble de funciones secundarias que de primarias es bueno.
- b. Ejemplos:
 - 1) “Hacer caducar la lista de trabajos.” Treinta días después de que la lista de trabajos se pone a disposición para ser vista, se dice que caduca. Una lista que ha caducado no se borra, pero es posible que los usuarios, con excepción del propietario de la lista, ya no puedan verla.
 - 2) “Renovar la lista de trabajos.” Se puede extender una lista por 30 días adicionales mediante el pago de una tarifa adicional.

- 3) “Hacer que una lista de trabajos sea prioritaria.” En cualquier momento, durante la vida de una lista, su propietario puede elegir promoverla hacia lista prioritaria, mediante el pago de una tarifa prorrateada por la parte consumida del periodo de la misma.
- 4) “Registrar las listas vistas.” Cada vez que se ve una lista, se escribirá una entrada de registro, haciendo constar la fecha y la hora en que se vio la lista y el protocolo de Internet (IP) del visitante.
- 5) “Examinar registros de las veces que se ha visto la lista.” En cualquier momento, el propietario puede ver la información registrada en relación con sus listas.
- 6) “Notificación automática de los registros de la vista de la lista.” El propietario de una lista de trabajos puede elegir que se le envíen por correo electrónico los registros de las vistas de esa lista, con un intervalo especificado por él.
- 7) “Pagar por la lista.” Se pide al propietario que pague por cada lista, a menos que ésta se ofrezca como un premio de promoción.

5. Escenarios primarios

- a. Descripción y ejemplo: Un escenario es un relato corto que describe las funciones en un contexto. Por ejemplo, dada una función primaria “Crear lista de trabajos”, podríamos escribir un escenario como éste: “La secretaria del Sr. García está por jubilarse, y él necesita contratar a alguien que la reemplace. Al Sr. García le gustaría una secretaria que mecanografie 100 palabras por minuto, quiera trabajar sólo cuatro horas al día y cobrar 10 dólares por hora. Necesita que la secretaria de reemplazo empiece a trabajar no después del 15 de enero.” Considere por lo menos tantos escenarios primarios como funciones primarias tenga. También considere un par de variaciones del escenario para las funciones importantes. Esto ayudará a que piense acerca de su problema en formas creativas. Hacer una lista de los escenarios en aproximadamente el mismo orden que el de las funciones que describe el escenario es una práctica útil.

6. Escenarios secundarios

- a. Descripción y ejemplo: Los escenarios secundarios son relatos cortos que ponen a las funciones secundarias en un contexto. Considere un escenario secundario al que nos referiremos como “Hacer caducar lista de trabajos”. Demostrado como un escenario, podríamos escribir: “El Sr. García pagó para que la lista se publicara durante 30 días. Después de 30 días, la lista de trabajos se retirará y se notificará al Sr. García por correo electrónico, dándole oportunidad de renovarla.” Podemos organizar los escenarios secundarios en un orden coherente con las funciones secundarias que apoyan.

7. Necesidades no funcionales
 - a. Descripción: Las necesidades no funcionales se encargan de comportamientos implícitos, como con qué rapidez sucede algo o cuántos datos se pueden transmitir.
 - b. Ejemplo: Debe procesarse el pago de un patrón en un periodo no mayor a 60 segundos, en tanto que él o ella, espera.
8. Finalizaciones de los casos de uso
 - a. Descripción: En esta parte se describe lo que significa haber finalizado para el caso de uso.
 - b. Ejemplo: El caso de uso ha finalizado cuando los cambios hechos en la lista de trabajos se han mantenido y se ha hecho el pago.

Cuánta información incluya en la parte escrita de sus casos de uso en realidad es decisión de usted. El UML guarda silencio acerca de este asunto, pero un proceso como el RUP le puede ofrecer alguna guía sobre el contenido, cantidad y estilo de la documentación en texto.

Como nota final, resulta útil registrar ideas acerca de las funciones y escenarios, incluso si finalmente elige descartarlos. Por ejemplo, podríamos agregar una función secundaria que exprese que “El sistema permitirá una renovación semiautomática de una lista de trabajos que están caducando”, apoyada por el escenario “La lista del Sr. García para contratar una nueva secretaria está próxima a caducar. Se notifica por correo electrónico al Sr. García que su lista está próxima a caducar. Al hacer clic sobre un vínculo en el correo, la lista del Sr. García se renueva en forma automática usando la misma facturación e información de pago usadas con la lista original”.

Al registrar y conservar las ideas consideradas, es posible hacer un registro de las ideas que se consideraron, pero puede ser que se lleven a cabo o que jamás se haga. Conservar un registro de las posibilidades impide que usted repita una y otra vez las ideas conforme los miembros del equipo vienen y se van.

Por último, resulta útil insertar referencias a los casos de uso del que se depende. En lugar de, por ejemplo, repetir un caso de uso de inclusión, sencillamente haga una referencia a ese caso en el punto en que se necesite. Por ejemplo, suponga que pagar por una lista de trabajos requiere que un patrón tenga acceso; en lugar de repetir el caso de uso “Entrar”, sencillamente hacemos una referencia a ella en donde se necesite; en este caso, podemos hacer una referencia a “Entrar” cuando hablemos acerca de pagar por la lista de trabajos.

Creación de los diagramas de casos de uso

Como mencioné al principio, los casos de uso son listas de diseños por hacer. Ya que un día feriado siempre está precisamente a la vuelta de la esquina, una buena analogía

comparativa es que definir casos de uso es como escribir una lista de tareas en orden para preparar su casa para una gran visita de parientes. Por ejemplo, podría escribir “Desempolvar la sala”. Entonces decide que su hija de 10 años hizo un buen trabajo la última vez, de modo que le pide a ella que desempolva. En este caso, el nivel de detalle es importante, porque sabe —si alguna vez ha desempolvado— que diferentes tipos de cosas necesitan diferentes tipos de desempolvado: las chucherías pequeñas se pueden desempolvar con un plumero; las mesas para servir café y las de los extremos podrían necesitar Pledge® y un paño limpio y seco, y los ventiladores del techo podrían necesitar la varita y el cepillo de una aspiradora. La clave en este caso es la diferencia entre lo que describimos con un diagrama y lo que escribimos como parte de nuestro caso de uso.

NOTA Podría preguntarse qué tiene que ver desempolvar con los casos de uso y el software. La primera respuesta es que se pueden usar los modelos de caso de uso para cosas que no son software, y la segunda parte es que el software se encuentra en un número cada vez mayor de aparatos. Suponga que estábamos definiendo casos de uso para un robot que limpia casas; entonces nuestras reglas para desempolvar podrían ser útiles. Y si se está preguntando cuán probable podría ser el software para robots, entonces considere la aspiradora Roomba®, un pequeño robot que vaga por un cuarto aspirando los desperdicios y, según su material de mercadeo, incluso sabe cuándo recargarse. Alguien tuvo que definir e implementar esas capacidades.

El caso de uso para desempolvar del párrafo anterior constaría de un actor, “Niña”, un conector de asociación y un caso de uso “Desempolvar la sala” (figura 2-9). No hace falta que el propio diagrama de casos de uso describa todas las microtareas necesarias de las que consta “Desempolvar la sala”. Por ejemplo, “Encontrar Pledge y un paño limpio y seco” es una subtarea necesaria, pero en realidad, no es un caso de uso en y por sí misma. Los casos de uso buenos significan tener que hallar buenos actores y el nivel correcto de detalle, sin hacer confusos los diagramas.

Después de que tenemos el diagrama de caso de uso, podemos agregar información de soporte de la documentación del modelo para nuestro caso de uso. Las funciones primarias incluirían desempolvar las áreas clave, y las funciones secundarias incluirían la preparación, como hacerse de la aspiradora y hallar el Pledge. Los escenarios adecua-

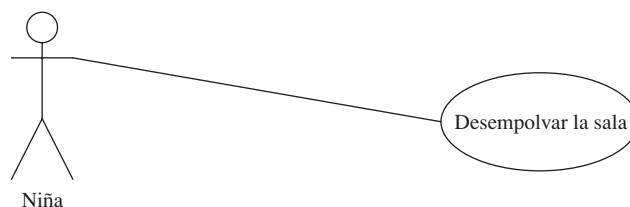


Figura 2-9 Caso de uso para un actor niña y desempolvar una sala.

dos incluirían el manejo de áreas problemas específicas, como desempolvar los marcos de los cuadros y artículos de colección. Los requisitos no funcionales podrían incluir “Terminar de desempolvar antes de que lleguen los abuelos”. No se preocupe acerca de lograr diagramas perfectos ni de la documentación del caso de uso; use el borrador para ayudarse a considerar los detalles y los diagramas de caso de uso para obtener una buena imagen de sus objetivos.

¿Cuántos diagramas son suficientes?

La suficiencia es un problema difícil. Si proporciona demasiados casos de uso, su modelo puede continuar durante meses o incluso años. También puede adentrarse en el mismo problema con la documentación de los casos de uso.

***NOTA** Fui consultor en un proyecto para un departamento grande de la agencia de defensa. Literalmente, la agencia había estado trabajando sobre casos de uso por casi dos años sin tener el fin a la vista. Aparte de que me pareció un proyecto interminable, los expertos del dominio sentían que se estaban captando casos de uso erróneos o que las casos de uso tenían poco o ningún valor práctico y explicativo. Los modelos no estaban logrando la marca. El objetivo es captar las características esenciales del objetivo de usted, y los modelos de casos de uso son una excelente manera de baja tecnología para hacer que se involucren expertos en el dominio que no son técnicos. Pasar por alto el valor de incitación al diálogo de los diagramas de casos de uso es perder la mitad del valor de esos diagramas.*

Una línea de base razonable es que las aplicaciones de mediana complejidad podrían tener entre 20 y 50 buenos casos de uso. Si usted sabe que su problema es moderadamente complejo y tiene cinco casos de uso, entonces puede estar pasando por alto funcionalidad crítica. Por otra parte, si tiene cientos de casos de uso, entonces puede estar subdividiendo macrocajas de uso prácticas en microcajas.

Desafortunadamente, no existen reglas difíciles y rápidas. Definir los casos de uso correctos requiere práctica y buen juicio que se adquiere con el transcurso del tiempo. Para ayudarlo a empezar a adquirir alguna experiencia, en la siguiente subsección se demuestran algunos diagramas reales de casos de uso para www.motown-jobs.com.

Ejemplos de diagramas de casos de uso

Este libro es acerca del UML. La documentación específica de texto no es parte del UML, de modo que limitaré los ejemplos de esta sección a la creación de los diagramas de casos de uso. Puede usar su imaginación y el borrador de la sección titulada “Documentación de un caso de uso usando un borrador” para practicar la escritura de las descripciones de casos de uso.

Motown-jobs.com es un producto de mi empresa, Software Conceptions, Inc. Motown-jobs es un sitio web para poner en contacto personas que buscan trabajo con quienes los ofrecen; es un sitio web como dice.com, monster.com, computerjobs.com o hotjobs.com y está implementado en ASP.NET. Dejando todo esto aparte, Motown-jobs.com se inició como una idea cuyas características se captaron como un grupo de casos de uso. Debido a que estaba estructurando el software para mi empresa, tuve que representar el papel de experto en el dominio, siendo el dominio lo que se requiere para hacer coincidir patrones con empleados. Dado que he estado buscando y hallando clientes para mi empresa durante 15 años, tengo algo de experiencia en esta área.

Hallar los casos de uso puede iniciar con una entrevista con su experto en el dominio, o bien haciendo una lista. Como yo estaba representado el papel de entrevistador y entrevistado, sencillamente empecé con una lista de las cosas que pensaba que Motown-jobs.com necesitaría ofrecer para ser útil. He aquí mi lista:

- Los patrones o los agentes de los patrones querrán publicar información acerca de los trabajos que están ofreciendo.
- Quienes están buscando trabajo pueden querer publicar un currículum vitae que puedan ver los patrones potenciales.
- Los patrones o los agentes de los patrones querrán buscar en forma activa en el sitio web los currículum vitae que se ajusten a las habilidades necesarias para llenar los sitios vacantes en el trabajo.
- Quienes están buscando empleo querrán buscar en los puestos que se encuentran en lista.
- Los patrones o los agentes de los patrones deberán pagar por las listas y por buscar en los currículum vitae, pero publicar currículum vitae o buscar en las listas de trabajos será un servicio gratuito.
- Una fuente adicional de ingresos podría ser publicidad y servicios de estructuración de currículum vitae, de modo que el sitio web podrá vender y tener espacio para publicidad, y ayudar a los solicitantes de trabajo a crear su currículum vitae.

Además de que escribir software es caro y de que también lo son el hardware, el software del servidor y las conexiones de alta velocidad de Internet tanto para comprarlos como para darles mantenimiento, ayudar a las empresas a encontrar empleados es un servicio valioso, o, por lo menos, ésa es la premisa que se encuentra detrás de la estructuración de Motown-jobs.com. Resolver acerca de cuánto cobrar por las listas y para atraer anunciantes son funciones de negocios y de mercadeo, de modo que hablaré acerca de eso en mi lista de casos de uso.

Ahora bien, claro que podría empeñarme en examinar todas las pequeñas tareas de las que consta cada una de las macrotareas —como publicar las vacantes de puestos de trabajo—, pero la lista que tengo es un buen lugar para iniciar. Empecemos por diagramar estas características (figura 2-10).

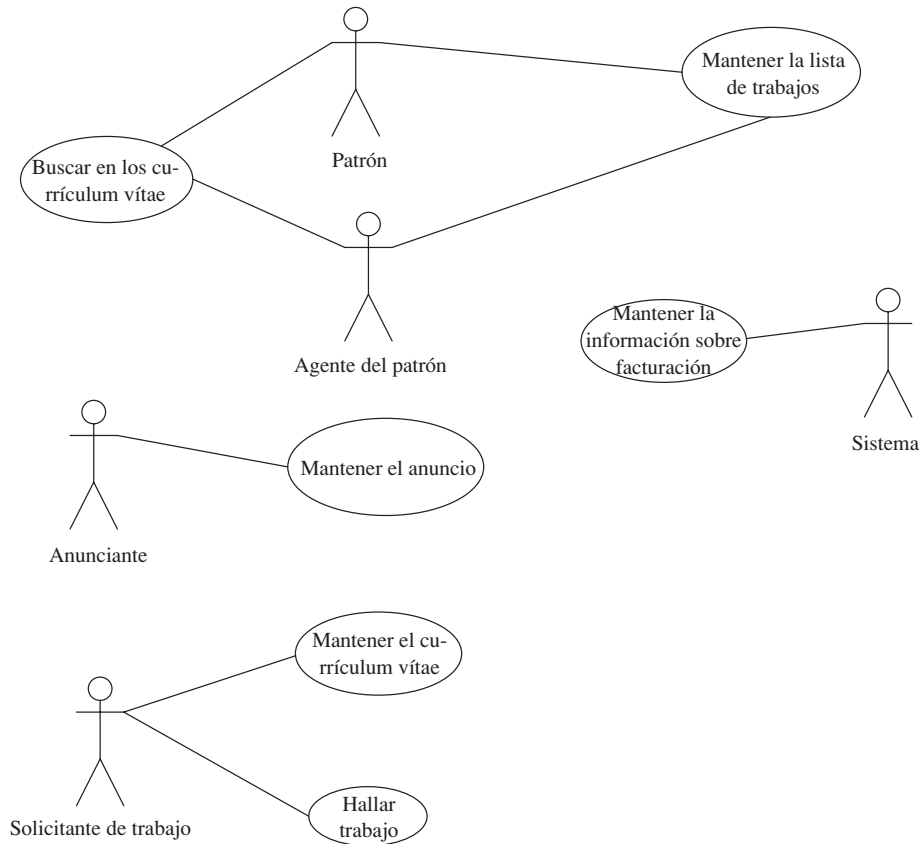


Figura 2-10 Un primer paso en el diagrama de casos de uso para Motown-jobs.com.

Observe en la figura 2-10 que capté mantener trabajos y hallar currículum para la clasificación patrón, mantener anuncios para la clasificación anunciantes, publicar currículum y hallar trabajos para la clasificación solicitantes de empleo y para administrar la facturación para el sistema. Lo siguiente que puedo hacer es preguntar a las partes involucradas si estos casos de uso captan la esencia de las características que necesito.

Como un diagrama de caso de uso, a esto le doy una calificación de C, pero es un inicio. Lo siguiente que puedo hacer es revisar a los actores y a los propios casos de uso en busca de redundancias, simplificaciones o detalles adicionales que se necesitan, y hacer al diagrama los ajustes necesarios.

Definición de los actores

En el diagrama de casos de uso de la figura 2-10, tengo los actores “Patrón” y “Agente del patrón”; no obstante, para todas las intenciones y finalidades, estos dos actores hacen

las mismas cosas en relación con el sistema y lo hacen de la misma manera; por consiguiente, puedo eliminar el “Agente del patrón” y renombrar al “Patrón” como “Propietario de la tarea”; con una descripción sencilla, “Propietario de la tarea”, se capta la idea de que un trabajo en lista “lo posee” una parte responsable. En la figura 2-11, se muestra la revisión en el diagrama de casos de uso.

A continuación, parece bastante obvio que una lista de puestos de trabajo, un currículum vitae y un anuncio son todos clasificaciones de listas, y las personas a quienes pertenecen esos elementos son “Propietarios de listas”. Puede experimentar con estas relaciones usando la generalización. En la figura 2-12, se muestra el diagrama modificado de casos de uso.

En la figura 2-12, se tratan los trabajos, los anuncios y los currículum vitae todos como listas que necesitan mantenerse. También se muestra que el sistema de facturación está asociado con las listas y las búsquedas de los currículum vitae. En ciertos aspectos, la figura 2-12 es una mejora, pero en otros es demasiado ingeniosa. Por ejemplo, describir

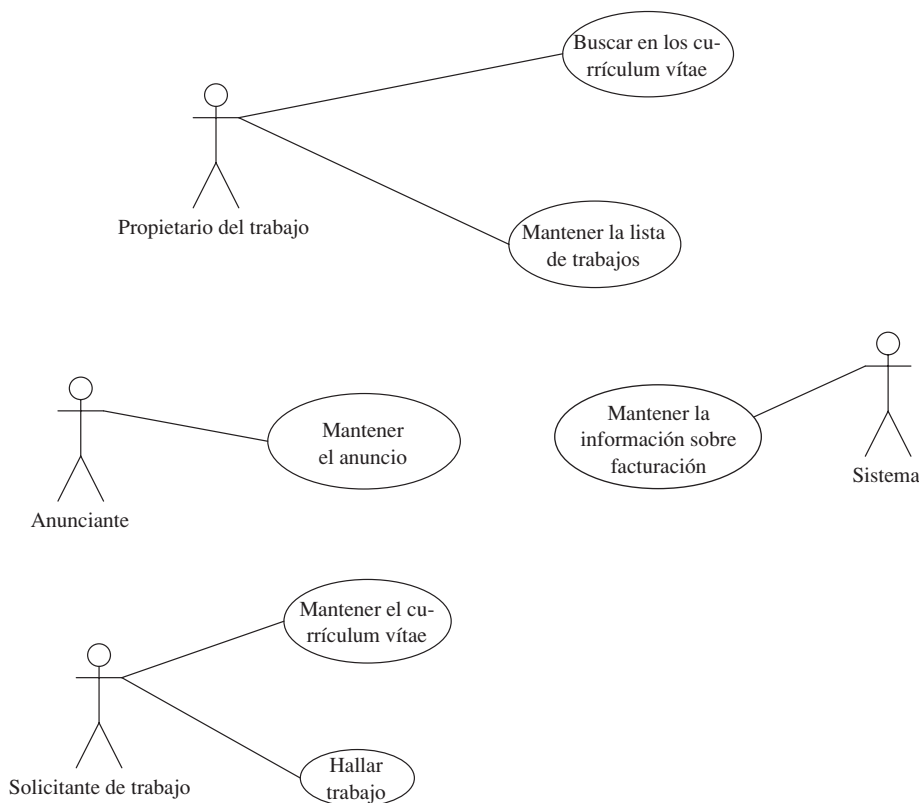


Figura 2-11 “Patrón” y “Agente del patrón” se convierten en un solo actor: “Propietario del trabajo”.

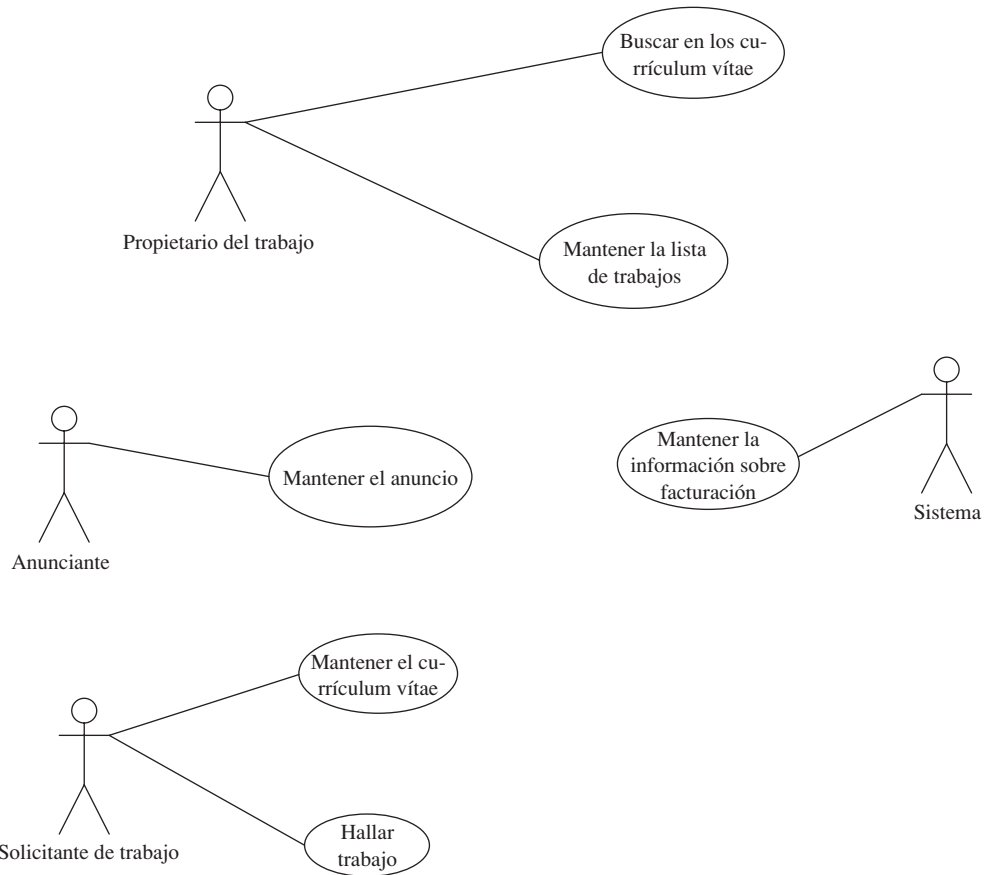


Figura 2-12 Esta figura sugiere que los trabajos, los currículum vitae y los anuncios son todos listas que debe mantener un propietario de lista, así como una asociación entre el sistema de facturación y las listas y las búsquedas de los currículum vitae.

a un solicitante de trabajo como un “Propietario de listas” sugiere que cada solicitante de trabajo posee un currículum vitae en lista. ¿Qué sucede si un solicitante de trabajo no quiere publicar un currículum vitae? Además, dije que publicar un currículum vitae es un servicio gratuito, pero la implicación es que el sistema de facturación trate las listas de currículum vitae como un concepto susceptible de facturación. ¿Significa esto que es susceptible de facturación, pero que el costo es 0 dólares? El diagrama revisado parece un poco más ingenioso y lleva tanto a preguntas como a respuestas. Quizás podría, además, dividir “Listas” en “Listas susceptibles de facturación” y “Listas gratuitas”. Esto podría resolver la cuestión del sistema de facturación, pero ¿qué sucede acerca de los solicitantes de trabajo que no publican currículum vitae? Todavía debo resolver este problema. Por ahora, regreso a los cuatro actores separados, en oposición a los tres tipos de propietarios de listas y el actor sistema (figura 2-13).

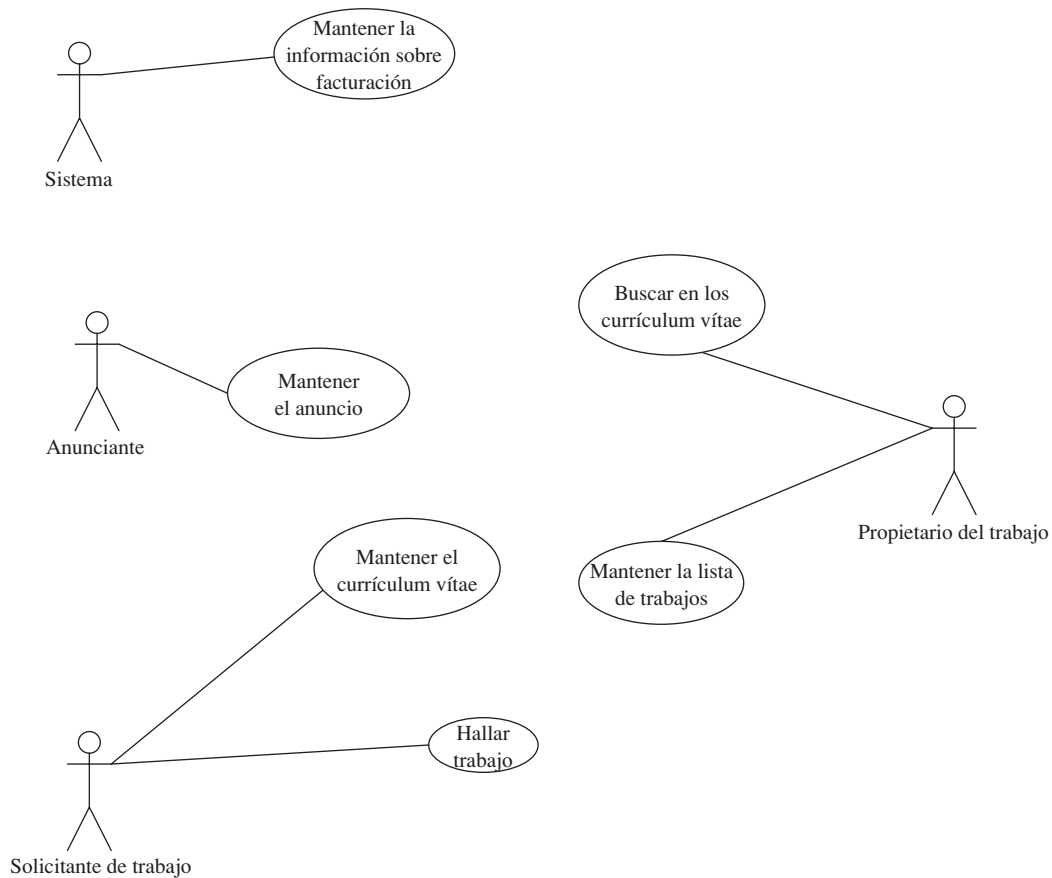


Figura 2-13 Cuatro actores separados no relacionados que participan en casos de uso no relacionados.

Me gusta la forma más sencilla del diagrama de casos de uso de la figura 2-13; está menos abarrotada, es más fácil de seguir y me dice lo que necesito saber acerca de las características del sistema.

División de los casos de uso en diagramas múltiples

Puede elegir tener un diagrama maestro de casos de uso y varios diagramas menores de casos de uso o sólo varios diagramas menores. Usted decide. Los diagramas más sencillos son más fáciles de manejar y seguir, pero puede ser que no muestren cómo están relacionados los casos de uso. En general, prefiero los diagramas sencillos y separados y crear un solo diagrama maestro, si estoy seguro de que al hacerlo obtendré algunos beneficios específicos.

En mi ejemplo de Motown-jobs.com, tengo cuatro facetas significativas; tengo casos de uso relacionados con el solicitante de empleo, casos de uso relacionados con el propietario del trabajo, casos de uso para los anunciantes y el sistema de facturación. Para examinar cada una de estas facetas del sistema, separaré estos casos de uso y los actores que les incumben en diagramas separados y agregaré detalles. En las figuras 2-14 a la 2-17 se muestran los nuevos diagramas.

Al separar “Mantener la información sobre facturación” en un caso de uso separado, tengo espacio para agregar detalles. Por ejemplo, es razonable que el sistema de facturación se interese sólo en lo que es susceptible de facturación y que un actor llamado “Usuario registrado” pueda mantener elementos susceptibles de facturación. Advierta que agregué el caso de uso “Entrar”. Dado que necesito saber cuáles usuarios están para ser facturados, necesitaré un medio de registrar y autenticar.

En la figura 2-15, introduje la idea de que un solicitante de empleo también se considera como usuario registrado. Sin embargo, elijo requerir registro sólo si el usuario quiere publicar un currículum. Quiero saber cuáles personas están proporcionando información a nuestro sistema, pero no lo requiero de los navegadores casuales. Una vez más, para publicar algo en el sistema, requeriré que al usuario se le dé acceso y, de lo contrario, sólo ofrecer al usuario casual la oportunidad de registrarse. El concepto de usuario registrado sugiere que necesito otro caso de uso “Mantener información de registros”. Esto puede implementarse como un sencillo diagrama de casos de uso, con el actor “Usuario registrado” y una asociación al nuevo caso de uso.

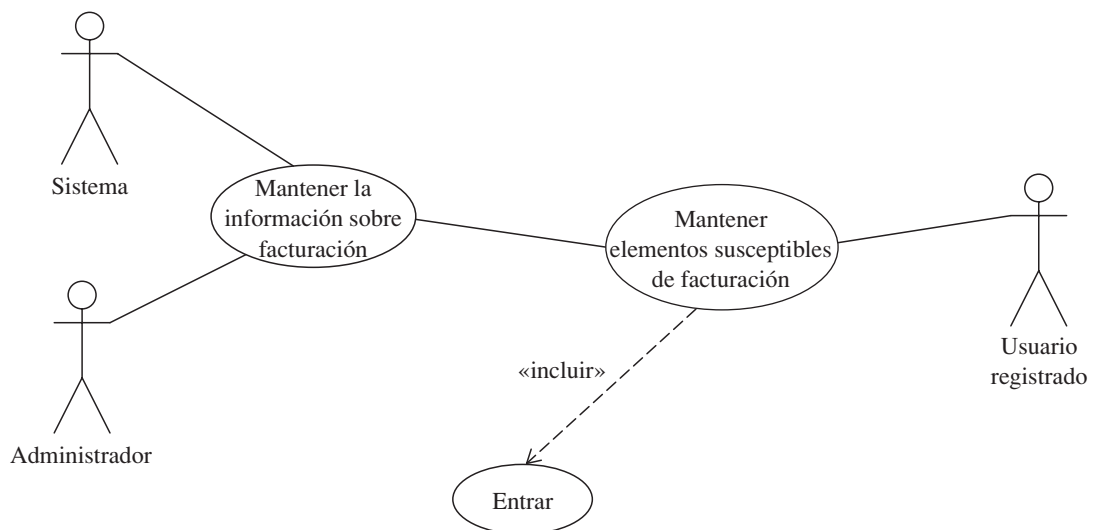


Figura 2-14 En esta figura se muestra que un nuevo actor, llamado “Usuario registrado”, puede mantener un elemento susceptible de facturación, si ese usuario entra y el sistema de facturación se asocia con los elementos susceptibles de facturación.

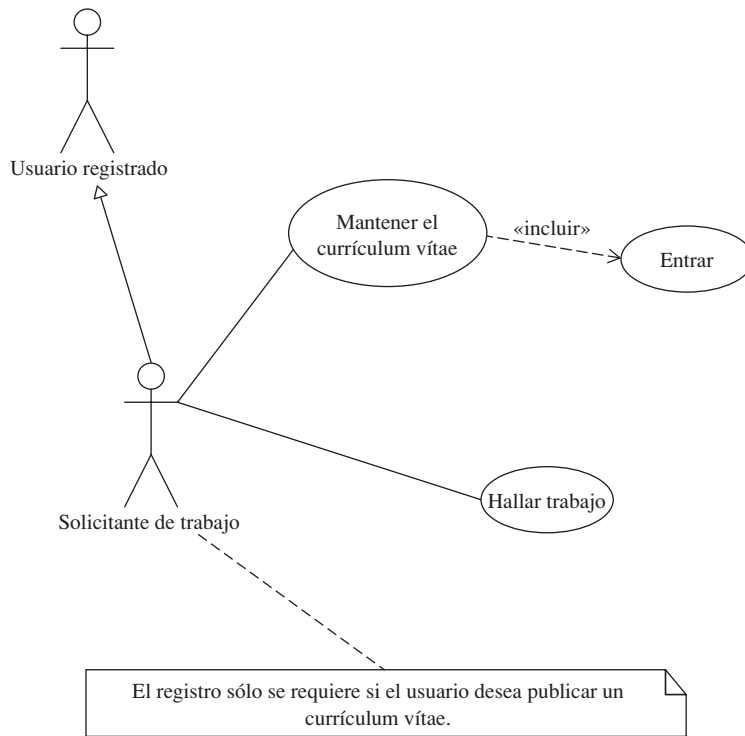


Figura 2-15 Vista ampliada de casos de uso relacionados con los solicitantes de trabajo.

En la figura 2-16, muestro que un anunciante es un usuario registrado y también incluyo que “Mantener el anuncio” generaliza “Mantener elementos susceptibles de facturación”. Dado que “Mantener elementos susceptibles de facturación” también está en el diagrama de la figura 2-16, de igual manera sé que esto significa que estoy ligado con los casos de uso de facturación, registro y autenticación (o concesión de acceso), pero de manera intencional quité esos elementos del diagrama para no abarrotarlo.

En la figura 2-17, señalo la dependencia entre “Mantener elementos susceptibles de facturación” y “Entrar” mostrando el conector de dependencia entre estos dos casos de uso. Debe resultar obvio que, como “Buscar en los currículum vitae” y “Mantener la lista de trabajos” generalizan “Mantener elementos susceptibles de facturación”, se requiere la autenticación para publicar trabajos y buscar en los currículum vitae. El uso de un solo conector simplifica el diagrama.

Ciertamente, será bienvenido el que usted intente crear un solo diagrama maestro de casos de uso, pero no necesita hacerlo. Incluso en este sistema relativamente sencillo, un solo modelo monolítico podría únicamente agregarse a la confusión; nuestro objetivo es reducir la confusión y aumentar la comprensión tan sencilla y directamente como sea posible.

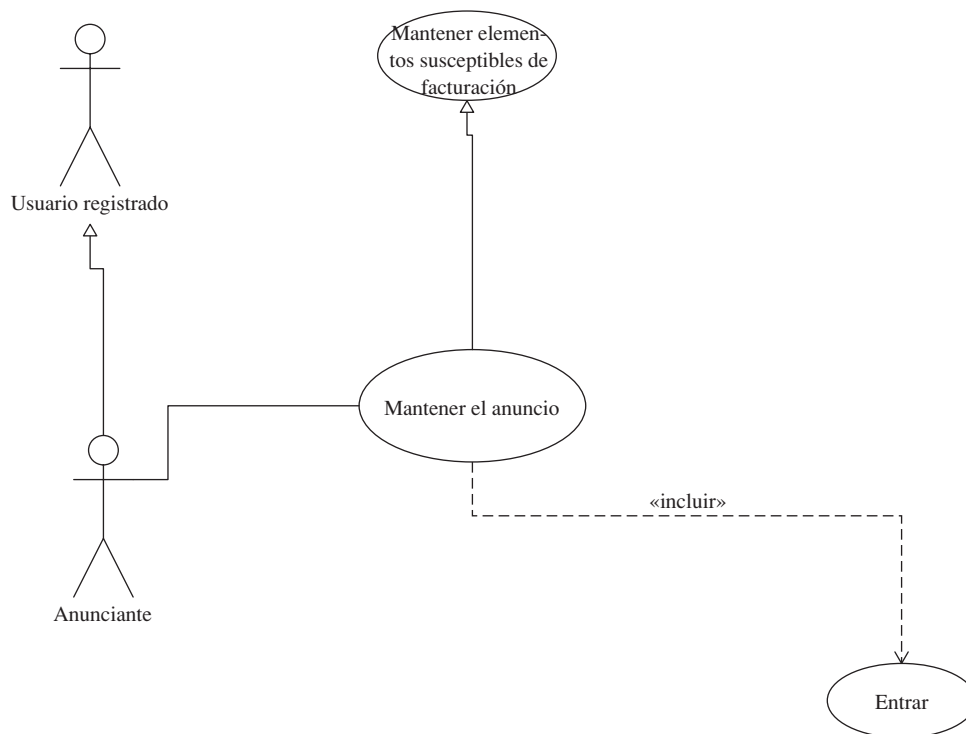


Figura 2-16 Vista cada vez más detallada de los casos de uso en los que intervienen los anunciantes.

Pienso que estos cuatro modelos hacen esto, pero la exposición ilustra en forma precisa la clasificación de temas que deberá pesar al decidir en cuáles modelos invertir su tiempo.

Manera de hallar la línea final

A medida que se evalúen sus diagramas de casos de uso y su documentación como texto escrito, le surgirán otras ideas y las cosas que pasó por alto. Esto es de esperarse. Documente estas ideas, incluso si al final las descarta. También esté preparado para revisar sus modelos a medida que cambien su comprensión y la de sus clientes o el clima de la empresa. Una comprensión creciente o un clima dinámico de la empresa significa más diagramas de casos de uso y revisiones a los ya existentes. Si anticipa la naturaleza dinámica de la comprensión, entonces no tendrá problema para continuar con los pasos siguientes, en lugar de intentar crear un juego perfecto de casos de uso sin reflexionar.

El objetivo de crear diagramas de casos de uso es documentar los aspectos importantes del sistema, para proporcionar a los usuarios una manera de baja tecnología para evaluar en forma visual sus comprensiones mutuas y, a continuación, seguir adelante. El resultado que deseamos es un juego de casos de uso “suficientemente bueno”, no perfecto.

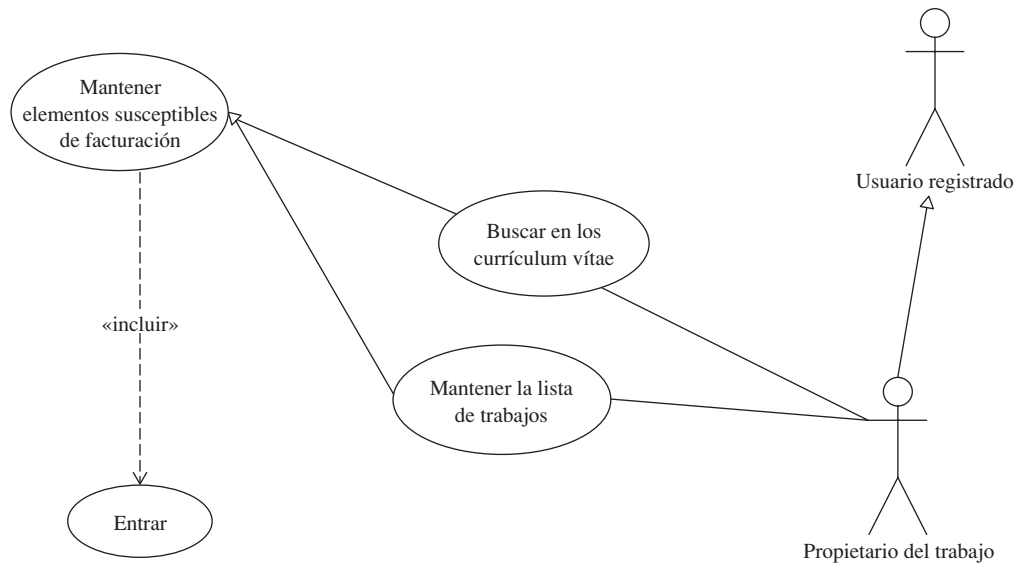


Figura 2-17 En esta figura se muestra la relación entre el propietario del trabajo y sus casos de uso, incluyendo una descripción clara de que se requiere la autenticación y que ese propietario esté administrando elementos susceptibles de facturación.

Diseño controlado con casos de uso

Hasta ahora, he definido casos de uso significativos y los diagramas de casos de uso para Motown-jobs.com. (Dejé fuera “Mantener información de los clientes”, pero sé que la necesito.) Con base en la exposición, debe resultar obvio que omití tareas menores, por ejemplo, leer listas en una base de datos y escribirlas para ésta. Sin embargo, esto queda cubierto en “Mantener la lista de trabajos”. No necesito un diagrama separado de casos de uso para mostrar que estoy “clabeando” —por CLAB, o sea, crear, leer, actualizar y borrar— listas, anuncios o currículum vitae, aunque resultará útil describir estas cosas en diagramas futuros, como los diagramas de secuencia (vea el capítulo 6 para obtener más información). Lo siguiente que me interesa realizar es el establecimiento de prioridades.

Demasiados proyectos pasan por alto por completo los casos de uso e ignoran el establecimiento de prioridades, pero los casos de uso existen para ayudarle a administrar el alcance y para establecer prioridades. El término *diseño controlado por casos de uso* significa que expresamos lo que estamos estructurando en nuestros casos de uso con el fin de limitar el alcance y evitar el desperdicio de tiempo, y establecemos prioridades en lo que estructuramos empezando con las características más críticas y de prioridad más alta. Con demasiada frecuencia, los programadores estructurarán cosas agradables o fáciles como primeros diálogos “Acerca de” y campanas y silbidos innecesarios, porque están examinando alguna nueva tecnología, y esto es un factor significativo de por qué fallan tantos proyectos.

Después de que haya definido sus casos de uso, querrá establecer prioridades y además diseñar e implementar una solución para apoyar aquellos casos de uso con la prioridad más alta o que representan el riesgo más significativo. ¿Cómo decide qué diseñar y estructurar primero? La respuesta es pregúntele a su cliente qué es lo más riesgoso, lo más importante o lo más valioso y a continuación enfoque sus energías en esos casos de uso.

NOTA *La pregunta real que debe hacer a su cliente es: “¿Qué características podemos estructurar primero de modo que si estamos fuera de tiempo y de presupuesto, todavía tendremos un producto que pueda comercializarse?” Los clientes no siempre quieren escuchar las preguntas difíciles, y usted deberá aplicar cierta diplomacia, pero hallar la respuesta correcta a esta pregunta y actuar en términos de ella puede ser lo más importante que usted haga.*

Para Motown-jobs.com, decidí —como cliente— que puedo presentarme en el mercado con un servicio de listas de trabajo con base en una tarifa. Esto significa que si implemento “Mantener la lista de trabajos”, “Buscar un trabajo” y “Mantener la información sobre facturación”, tendré un producto con el que puedo entrar al mercado. Esto no significa que no querré estructurar en el sistema la publicación de currículum vitae, la búsqueda y el apoyo para la publicidad; sólo significa que éstas no son las características más importantes.

Las prioridades que siguen son más difíciles. ¿Debo estructurar a continuación la publicación de currículum vitae y la búsqueda, o la publicidad? La respuesta es que quiero que los solicitantes de trabajo usen el servicio y los propietarios del trabajo vean que hay una gran cantidad de tráfico e interés en mi sitio, de modo que apoyaré a continuación la publicación de un currículum vitae —el cual es un servicio gratuito pero crítico— y, después, la búsqueda de currículum vitae, el cual también es un servicio gratuito, pero que depende de tener currículum vitae para revisar. Por último, apoyaré la publicidad, la cual finalmente depende de tener tráfico suficiente para interesar a los anunciantes.

Lo importante aquí es que la identificación de mis casos de uso me ayudó a establecer prioridades en mi lista de tareas e ilustra un camino crítico para mi criterio de éxito mínimo: vender anuncios de se solicitan empleados.

Examen

1. ¿Qué símbolo representa un caso de uso?
 - a. Una línea
 - b. Una línea dirigida
 - c. Una figura de palillos
 - d. Un óvalo que contiene texto

2. Un actor solamente puede serlo una persona.
 - a. Verdadero
 - b. Falso
3. ¿Qué símbolo representa una dependencia?
 - a. Una línea
 - b. Una línea con un triángulo que apunta hacia el elemento dependiente
 - c. Una línea punteada con una flecha que apunta hacia el elemento dependiente
 - d. Una línea punteada con una flecha que apunta hacia el elemento del que se depende
4. ¿Cómo se indica un estereotipo sobre un conector?
 - a. Texto entre un par de comillas angulares
 - b. Texto llano próximo al conector
 - c. La palabra estereotipo dentro del símbolo de óvalo
5. Se usa una relación de inclusión para reutilizar el comportamiento modelado por otro caso de uso.
 - a. Verdadero
 - b. Falso
6. Se usa una relación de extensión para modelar características opcionales del sistema.
 - a. Verdadero
 - b. Falso
7. En el UML la generalización se refleja en la implementación por
 - a. polimorfismo.
 - b. agregación.
 - c. herencia.
 - d. interfaces.
8. Todas las capacidades de un sistema deben representarse por un caso de uso.
 - a. Verdadero
 - b. Falso
9. En una relación extendida, la flecha apunta hacia el
 - a. caso de uso básico.
 - b. caso de uso de extensión.

10. Es importante implementar primero los casos de uso fáciles para garantizar que los primeros esfuerzos tengan éxito.
 - a. Verdadero
 - b. Falso

Respuestas

1. c
2. b
3. d
4. a
5. a
6. a
7. c
8. b
9. a
10. b

Diagramación de características como procesos

Este capítulo se refiere a los diagramas de actividades. Aun cuando lo que destaco no es el proceso, el siguiente paso después de captar los casos de uso consiste en empezar a describir cómo se llevarán a su término las características representadas por sus casos de uso. Los diagramas de actividades ayudarán a usted y a los usuarios a describir en forma visual la secuencia de acciones que le conduzcan a través de la compleción de la tarea.

La meta es convergir hacia el código en forma continua, partiendo de una comprensión del espacio del problema en general y captando los problemas que resolveremos —los casos de uso— mediante la descripción de cómo funcionan esas características y, al final, implementando la solución. Los diagramas de actividades constituyen una herramienta útil de análisis y se pueden usar para la reingeniería del proceso; es decir, el rediseño del proceso. De esta manera, los diagramas de actividades constituyen un puente progresivo que conduce del análisis hacia el diseño y, por último, a la implementación. En este capítulo aprenderá acerca de

- Los símbolos usados para crear los diagramas de actividades
- Cómo crear diagramas de actividades describiendo los casos de uso y los escenarios como una serie de acciones

- Modelar comportamientos simultáneos
- Refinar las actividades físicas con diagramas de actividades
- Comprender cuándo detener la creación de los diagramas de actividades

Elaboración de las características como procesos

Pocas ideas son por completo nuevas. Los conceptos existentes se refinan, evolucionan y maduran, llevándose con ellos algo de lo viejo y algo de lo nuevo; lo mismo es cierto para los conceptos de análisis y de diseño.

El análisis y el diseño estructurados hicieron hincapié en los diagramas de flujo. Un diagrama de actividad en el Unified Modeling Language (UML) está bastante cercano a un diagrama de flujo; los símbolos son semejantes pero no los mismos; la utilidad es semejante, pero existe una diferencia: los diagramas de actividades, a diferencia de los de flujo, pueden modelar comportamiento paralelo.

Los diagramas de actividades son buenos diagramas de análisis para los desarrolladores, los usuarios, los que hacen pruebas y los administradores, porque usan símbolos sencillos, texto llano y un estilo semejante al del conocido diagrama de flujo. Los diagramas de actividades son buenos para ayudarle a captar, visualizar y describir un conjunto ordenado de acciones, desde un principio hasta un final. Los diagramas de actividades se crean como un conjunto finito de acciones en serie o una combinación de acciones en serie y en paralelo.

Un viaje hacia el código

Un principio básico del análisis y del diseño orientados a objetos es que queremos partir de ideas y conceptos de alto nivel del espacio de problemas y movernos hacia un espacio de bajo nivel de soluciones. El espacio de alto nivel de problemas también se conoce como *dominio de los problemas*. El espacio de bajo nivel de soluciones se conoce como el *dominio de las soluciones*. El UML es un lenguaje para captar y describir nuestra comprensión a medida que avanzamos desde documentar un problema hasta codificar una solución.

Con base en la idea de trasladar nuestra comprensión desde el concepto hasta el diseño, los casos de uso constituyen una buena manera de captar las cosas que describen nuestro problema. Por ejemplo, queremos hacer corresponder a los patrones con los empleados potenciales proporcionando un tablero de listas de trabajos. Un caso de uso que da soporte a esto es administrar las listas. Un paso siguiente en un sentido abstracto consiste en describir cómo emprenderíamos la administración de una lista. En esta coyuntura, todavía es demasiado pronto para empezar a hablar acerca de bases de datos y lenguajes de programación; en cambio, queremos hablar acerca de las actividades que describen nuestro problema, y estas actividades constan de acciones.

NOTA *En un nivel ideológico, análisis y diseño son procesos según los cuales descompondremos un problema en problemas discretos menores, de tal manera que podamos componer soluciones pequeñas para cada problema discreto y, al final, orquestar las soluciones pequeñas en un todo coherente. El UML es un lenguaje para descomponer un problema y recomponerlo como la descripción de una solución. Un lenguaje como Visual Basic.NET es útil para implementar la descripción de la solución, y el proceso es la forma en que la emprenderíamos.*

Comprensión de los usos de los diagramas de actividades

Los diagramas de actividades no son en realidad acerca de los métodos o clases. Todavía es demasiado pronto para eso. La razón para que sea así es porque las cosas técnicas, como el polimorfismo, la herencia, los métodos y los atributos, en general, son conceptos sin significado para los usuarios y, a veces, para los administradores.

Los diagramas de actividades constituyen un medio a través del cual podemos captar la comprensión de las personas a las que llamamos *expertos del dominio*. Por ejemplo, si está estructurando un sistema para administración de cárceles, entonces un experto del dominio podría ser un oficial del penal, quien posiblemente no entienda la diferencia entre un espacio de nombres, una clase y la interfaz, pero, como diseñador, puede ser que usted no comprenda el significado de una compra de 50 cepillos de dientes por parte de un recluso. Un diagrama de actividad puede ayudar.

Un relato verdadero —y por qué consultar puede resultar interesante— se encuentra detrás de la metáfora de los cepillos de dientes. Mientras estaba trabajando para una cárcel grande del condado en Oregon, tuve que escribir una aplicación piloto para demostrar ASP.NET en sus primeros días. La aplicación piloto sería finalmente parte de un sistema de administración de cuentas de los reclusos para la cárcel. La idea básica es que los prisioneros no pueden tener efectivo en su posesión, pero pueden tener dinero en una cuenta para comprar artículos personales y golosinas. El condado administró las cuentas. Algunas de las reglas incluían límites sobre el número de barras de dulce que, digamos, un diabético podría comprar, así como un límite sobre el número de cepillos de dientes que podían comprarse. Al no ser oficial del penal, me pareció extraño que alguien comprara más de un cepillo de dientes y más extraño todavía por qué a alguien podría interesarle. El problema es que cuando se les talla hasta que formen una punta o se les hace una ranura con un trozo de hoja de rasurar encajada en forma segura en la punta y sostenida en su lugar con una banda de caucho, un cepillo de dientes se puede convertir en un arma formidable. (En realidad, sabía esto porque lo aprendí cuando fui policía militar o lo vi en un episodio de “Oz” en HBO.)

En la práctica, este relato es ilustrativo del hecho de que aquellos que se encuentran sobre el terreno —los expertos del dominio— conocerán los detalles en los que usted nunca pensará. Los diagramas de actividades son buenos para captar estos detalles en un sentido general y de una manera en que los expertos del dominio pueden examinar, aclarar y mejorar.

Trabajando hacia atrás, partiendo de mi relato de administración de cuentas de los prisioneros, podría tener un caso de uso “Hacer compra” y un escenario que garantice que la compra no viole una regla de seguridad. Podemos captar esto en un diagrama de actividad con la llaneza suficiente como para que un oficial del penal nos pueda decir si comprendimos el problema y lo hemos descompuesto en forma suficiente. En la figura 3-1, se muestra un diagrama de actividad para este escenario.

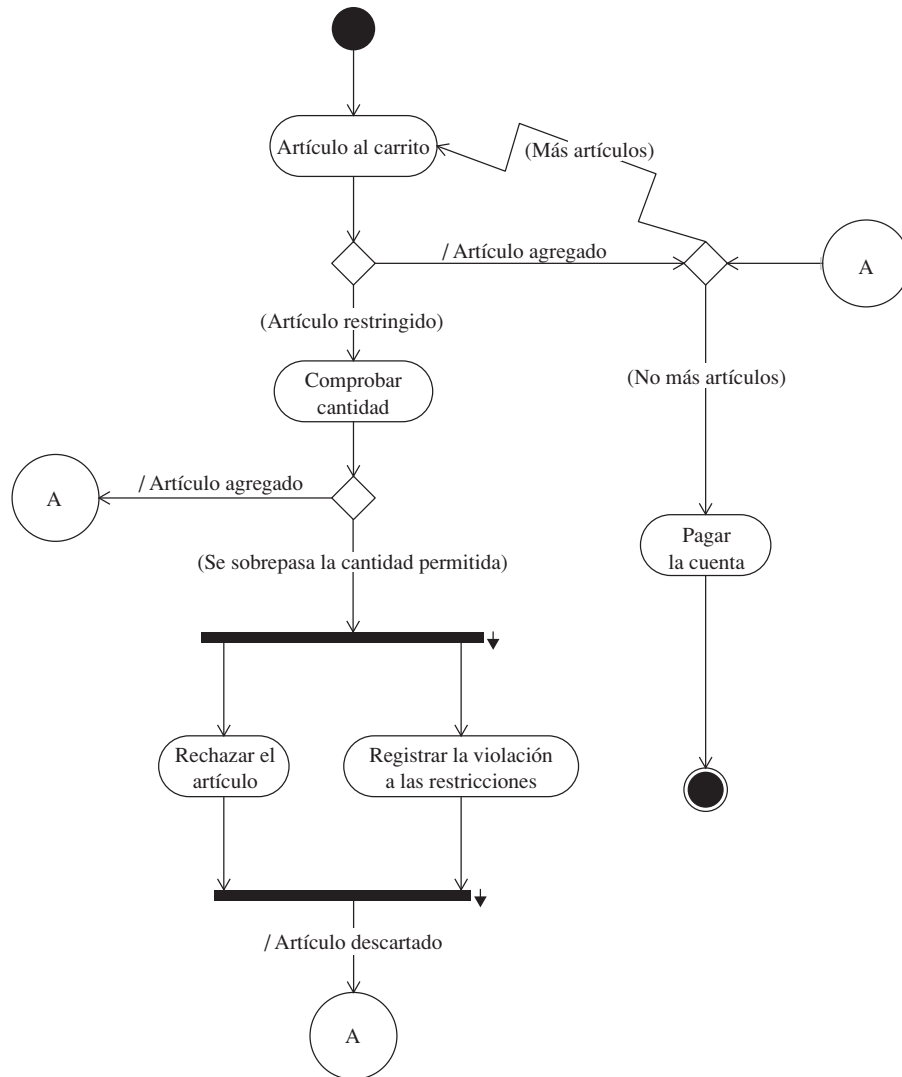


Figura 3-1 Diagrama de actividad que ilustra las restricciones sobre el tipo y el número de artículos que se pueden comprar estando en prisión.

Por ahora, no se preocupe acerca de qué significan las formas. Sólo advierta el texto sencillo y el flujo sugerido por las flechas. La idea general es que de un solo vistazo —quizás con un mínimo de explicación— este diagrama debe tener sentido para los usuarios y desarrolladores sin distinción. En la siguiente sección, empezaremos a examinar lo que significan estos elementos y más.

Uso de los símbolos de los diagramas de actividades

Los diagramas de actividades pueden ser sencillos diagramas de flujo que tienen un punto de inicio y de finalización finitos, o diagramas más complejos que modelen comportamiento paralelo y múltiples subflujos, así como que definan múltiples terminaciones. Encuentro que trazar diagramas de actividades simples es una manera excelente de arrancar y que agregar demasiados escenarios alternos en un solo diagrama lo hacen difícil de manejar y de imprimir, así como de entender.

Hacer que sus diagramas de actividades sean comprensibles puede ser más importante que hacer que el diagrama sea detallado o que abarque todo. Otro error es crear diagramas de actividades para todos los casos de uso y escenarios. La creación de diagramas lleva tiempo, y una buena manera de enfocar su tiempo es trazando los diagramas de aquellos aspectos que son más críticos para resolver su problema.

Considere un par de ejemplos. Los programas que almacenan datos por lo común lo hacen en bases de datos en forma de relación. Este comportamiento se llama comportamiento de *crear, leer, actualizar y borrar* (CLAB). Leer una base de datos o escribir en ella se comprenden tan bien que yo no trazaría un diagrama para este comportamiento como una actividad separada. (De hecho, en realidad la noción de una base de datos no debe mostrarse en un diagrama de actividad.) El comportamiento completo de leer y escribir se podría captar en algún punto en una actividad, como una acción llamada *traer y almacenar* o *leer y escribir*. Por otra parte —tomado prestado del capítulo 2— si vamos a hacer caducar una lista de trabajo de un cliente y queremos dar a ese cliente una oportunidad de extender la lista, entonces esto es menos común que el comportamiento CLAB, y yo crearía un diagrama de actividad para examinar la secuencia de acciones; construyendo el diagrama de la actividad “Hacer caducar la lista”, podría obtener la secuencia de acciones precisamente correcta y esto podría ser el catalizador para mejorar la calidad del servicio. Por ejemplo, podríamos hacer surgir la renovación por medio de la característica de correo electrónico que expusimos en el capítulo 2.

Si con anterioridad ha creado algunos diagramas de flujo con una herramienta como Visio, entonces los diagramas de actividades le parecerán bastante directos, pero tenga presente que estos últimos se pueden usar para modelar comportamiento más rico que los sencillos diagramas de flujo antiguos. Para crear diagramas de actividades, necesitará aprender acerca de los símbolos y reglas que se aplican.

SUGERENCIA Puede concebir los símbolos y reglas de cualquier diagrama UML como la gramática visual para el lenguaje.

Nodo inicial

Todo diagrama de actividad tiene un símbolo *nodo inicial*. Éste es un círculo relleno (vea la parte superior de la figura 3-1). Es posible proporcionar un nombre y alguna documentación para el nodo inicial, pero en general, yo no lo hago.

El nodo inicial puede tener una línea de transición saliendo de él. La línea de transición se llama *flujo de control* y se representa por medio de una flecha dirigida hacia fuera del nodo inicial. Por claridad, en la figura 3-2 sólo se representan el nodo inicial y el flujo de control. Puede colocar el nodo inicial en cualquier parte que le guste en el diagrama y agregar el flujo de control también en cualquier parte que le guste sobre ese nodo. Al vivir en el hemisferio occidental, tengo inclinación a poner los puntos de arranque arriba a la izquierda, y los de finalización abajo a la derecha.

Flujo de control

Como se mencionó con anterioridad, un flujo de control es una flecha dirigida. Un flujo de control también se conoce sólo como *flujo* o *estímulo*. El flujo de control empieza en el símbolo que pierde foco y apunta hacia la cosa que lo aumenta y se conecta con ésta. Por ejemplo, un flujo de control podría originarse en un nodo inicial y terminar en una acción, como se muestra en la figura 3-3.

Una manera usual de adornar un flujo de control es agregar una *condición guardián*. Una condición guardián actúa como un centinela que requiere que se pase una prueba antes de que el flujo continúe. En código, por lo común esto se implementaría como una prueba si condicional.

Uso de las condiciones guardianes

Sin desviar demasiado nuestra atención de las condiciones guardianes, una *acción* —acerca de la cual hablaremos más en la sección titulada “Acciones”— es algo que sucede en el flujo. Una acción, como el nodo inicial, es otro tipo de nodo. Los diagramas de activida-



Figura 3-2 El círculo relleno se llama *nodo inicial* —o punto de inicio del diagrama de actividades— y la flecha dirigida se llama *flujo de control*.

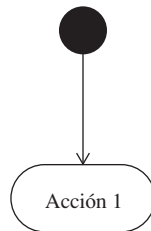


Figura 3-3 Un nodo inicial, flujo de control y una acción.

des están compuestos por completo de diversos tipos de nodos y flujos (o estímulos). Una condición guardián se muestra como texto entre corchetes y el lector puede concebirla como un portero hacia el nodo siguiente (figura 3-4).

Si alguna vez el lector ha servido en alguna clase de milicia, entonces está familiarizado con la noción de una palabra o frase contraseña:

Guardián: “El gorrión es un presagio.”

Soldado de infantería: “De muerte, que es lo único seguro junto con los impuestos.”

Guardián: “Puede pasar.”

Bien, cuando estuve en el ejército, las frases de contraseña nunca fueron ingeniosas, pero la idea es la misma. El guardián representa una prueba que se debe pasar para continuar. Lo que resulta muy extraño es que las pruebas programáticas pueden ser bastante esotéricas, pero el texto que usted escriba en sus condiciones guardianes dará mejor servicio a su clientela si son sencillas. La figura 3-5 es un ejemplo práctico de un nodo inicial, una acción y un flujo con una condición guardián.

En la figura, el nodo inicial realiza una transición hacia la primera acción, “Hallar cliente”. La condición guardián es que se conoce mi fecha de disponibilidad. De ninguna manera es bueno apilar clientes cuando no me queda tiempo disponible.

El diagrama de la figura 3-5 ilustra cómo un diagrama de actividad es una suerte de agnóstico cuando llega a la implementación. La actividad parcial de la figura 3-5 podría estar refiriéndose a un proceso físico como buscar el sitio web Motown-jobs.com y llamar

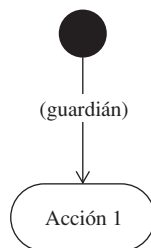


Figura 3-4 Un nodo de control, flujo con guardián y una acción genérica.

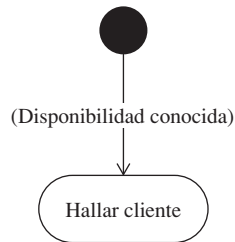


Figura 3-5 Parte de un diagrama de actividades para hallar clientes.

a los clientes que pasaron o a un proceso de software que explora en forma automática el sitio web Motown-jobs.com a través de un servicio web y envía correos electrónicos a los clientes que pasaron, notificándoles de mi disponibilidad. En todos los ejemplos de este capítulo, verá más casos de condiciones guardianes.

Diferentes maneras de mostrar flujos

La manera más común de diagramar un flujo es usar un solo símbolo de flujo de control conectado a dos nodos, pero ésta no es la única manera. Si su diagrama es muy complejo, con una gran cantidad de estímulos que se traslapan, entonces puede usar un nodo conector (figura 3-6). Un estímulo puede realizar una transición desde una acción hacia un objeto hacia una acción (figura 3-7) y entre dos clavijas (figura 3-8).

Uso de nodos conectores

No tiene usted que usar conectores, pero si sus diagramas se vuelven muy grandes o complejos, entonces encontrará que sus flujos empiezan a traslaparse o que su actividad se extiende hacia múltiples páginas. El nodo conector es una buena manera de simplificar los flujos que se traslapan o aquellos que se extienden hacia múltiples páginas.

SUGERENCIA La versión de Visio que usé para crear la figura 3-6 no soporta el nodo conector; para crear este efecto, tuve que usar la herramienta *Elipse* (Ellipse). El resultado es que el diagrama es visualmente correcto, pero Visio informará de un error. Como ocurre con muchas herramientas, se debe aceptar conceder algo a cambio.



Figura 3-6 Se puede usar un nodo conector con el fin de simplificar los diagramas de actividades que se ven muy abarrotados.

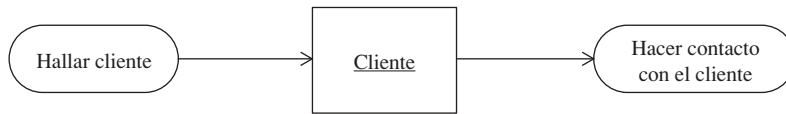


Figura 3-7 Inserción de un objeto cliente entre dos acciones relacionadas con clientes.

Para usar un nodo conector, trace un flujo que salga de un nodo y haga una transición hacia un conector. En donde se hace la conexión al nodo siguiente, dibuje un conector con un flujo que salga de él y que haga la transición hacia el siguiente nodo en el diagrama.

Los nodos conectores vienen en pares. Asegúrese de que las parejas de conectores tengan el mismo nombre; nombrar a los conectores le ayudará a hacer corresponder los puntos de conexión cuando tenga múltiples parejas de conectores en un solo diagrama.

Uso de objetos en los diagramas de actividades

Con anterioridad dije que la diagramación de actividades se presenta demasiado pronto en el análisis como para entender cuáles son los objetos; sin embargo, el UML permite la adición de objetos a los diagramas de actividades. Después de haber tenido una oportunidad de hacer que los usuarios le proporcionen alguna retroalimentación y haya crecido su comprensión del espacio de problemas, puede resultar útil agregar objetos a sus diagramas de actividades. La clave aquí es evitar la adición de conceptos técnicamente complejos demasiado pronto. Si se ha hundido en una discusión acerca de qué es un objeto o si éste se nombró o no en forma correcta, entonces elimínelo. Por otra parte, si el objeto es muy obvio —como se presenta en la figura 3-7— y ayuda a la comprensión de todos, entonces agréguelo.

Resulta valioso tener presente quién es su clientela para cada clase de diagrama. En general, concibo a los diagramas de actividades como herramientas de análisis que los usuarios finales leerán para ayudarle a usted a comprender cómo hacen su trabajo; la explicación de conceptos orientados a objetos parece, por lo común, ser una distracción, de modo que deje los objetos fuera de los diagramas de actividades.

Uso de clavijas

En el UML, las *clavijas* son análogas a los parámetros en la implementación. El nombre o valor de una clavija que sale de una acción debe concebirse como un parámetro de salida



Figura 3-8 Una técnica avanzada incluye la conexión de dos clavijas en los nodos de acción con un flujo de control.

hacia la acción siguiente. En las figuras 3-7 y 3-8 se transmite la misma información: que un cliente interviene en este flujo. Las clavijas, como los objetos, pueden ser demasiado detalladas por el uso cotidiano y pueden dar como resultado discusiones tangenciales confusas cuando se trabajan los flujos con los clientes. Sin embargo, si está explicando las actividades a diseñadores o programadores, pueden resultar útiles para mostrar objetos.

En las figuras 3-7 y 3-8 resulta claro que los nombres de las acciones —“Hallar cliente” y “Hacer contacto con el cliente”— sugieren que interviene un cliente. Dejando fuera el objeto y las clavijas —vea la figura 3-9— todavía se sugiere con mucha claridad la participación de un cliente, sin el riesgo de explicaciones largas y tangenciales.

Acciones

Los *nodos de acción* son las cosas que usted hace o que suceden en un diagrama de actividades, y un estímulo representa el camino que usted sigue para saltar de acción en acción. Los nodos de acción tienen una forma un poco más rectangular que los casos de uso. Dos de los aspectos más importantes de las acciones son el orden en el que ocurren y el nombre que les asigne. El nombre debe ser corto y directo. El uso de parejas de nombre y verbo en los nombres de las acciones puede ayudarle a hallar las clases y los métodos, pero los nombres de las acciones no tienen sólo esta finalidad y, una vez más, es bastante temprano en el análisis y el diseño para quedarse colgado en los detalles de la implementación, como las clases y los métodos.

Se permite que las acciones tengan uno o más flujos de entrada y sólo uno de salida. Si existe más de un flujo de entrada, entonces la acción no será transición hasta que todos los flujos de entrada hayan alcanzado esa acción. Las acciones se pueden dividir en caminos alternos con el uso del *nodo de decisión* —al que se hace referencia en la sección titulada “Nodos de decisión y de fusión”— o realizar una transición hacia flujos paralelos con el uso del *nodo bifurcación*, —vea la sección titulada “Bifurcaciones y uniones de transición”— pero en realidad, para una acción únicamente debe agregarse un solo flujo de salida, como un flujo saliente para una acción.

Una buena regla empírica para la creación de diagramas de actividades es describir cómo empieza un caso de uso, cómo progresa y cómo finaliza, con todas las acciones que deben completarse a lo largo del camino. Los nodos de decisión y de fusión y las bifurcaciones y uniones son medios para modelar comportamiento paralelo o alternaciones con la propia actividad. Si los flujos alternos son muy complejos, entonces puede usar el diagrama de subactividad para compartimentarla.



Figura 3-9 Este diagrama es más sencillo que aquellos en los que se muestra un objeto o se usan clavijas, pero todavía sugiere la participación de un cliente.

Las acciones también pueden usar condiciones previas y posteriores con el fin de indicar las condiciones necesarias antes y después de que ocurra la acción. Cortemos en trozos estos aspectos —nombres, subactividades y condiciones— formando subsecciones para examinar cómo anotamos cada aspecto de una acción.

Nombramiento de las acciones

Prefiero que las acciones tengan suficiente detalle —un nombre y un verbo— para describir lo que sucede y qué o quién interviene; por ejemplo, “Hallar cliente”, “Enviar correo electrónico al cliente”, “Almacenar lista de trabajo”, “Cancelar lista” y “Borrar currículo”. Sin una tremenda cantidad de texto adicional, estos nombres me dicen qué hace la acción y sobre qué actúa. Esto es importante, porque un concepto esencial en el UML es que se transmita una gran cantidad de información en forma visual, en oposición a hacerlo con una gran cantidad de texto.

Al final, los nombres y los verbos le ayudarán a encontrar las clases y los métodos, pero es una buena idea diferir pensar acerca de los detalles de la implementación todavía durante un tiempo. Sencillamente queremos entender cómo nos desempeñaremos para realizar una actividad, pero no cómo la implementaremos.

Por ejemplo, en el capítulo 2, definimos un caso de uso “Administrar una lista de trabajo”. Éste es un caso de uso acerca del cual se puede argumentar que consta de varias actividades, incluyendo “Publicar una lista de trabajo”. La publicación de una lista de trabajo es un escenario en el caso de uso “Administrar una lista de trabajo”, pero “Publicar una lista de trabajo” no es una sola acción. Se puede argumentar que deberían completarse varias acciones para captar toda la actividad. El siguiente es un ejemplo escrito que describe la publicación de una lista de trabajo, seguido de un corto diagrama de actividades (figura 3-10) que modela lo mismo:

- Proporcionar la descripción del trabajo
- Entrar
- Proporcionar información acerca del pago
- Proceso de pago
- Almacenar la descripción del trabajo
- Proporcionar confirmación

Una vez que tenemos un diagrama inicial —mostrado en la figura 3-10— contamos con una buena base para sostener una discusión acerca de la actividad. Podemos llevarla al dominio de los expertos y preguntarles acerca de los detalles del diagrama de actividades y evaluar esta información para determinar si necesitamos revisar el diagrama. Por ejemplo, tal vez queramos verificar si se puede usar información válida acerca del pago en relación con el archivo o queremos información nueva acerca de ese pago. O, si el usuario es nuevo, entonces puede ser que necesitemos agregar un punto de decisión que permita al usuario registrarse y, a continuación, entrar.

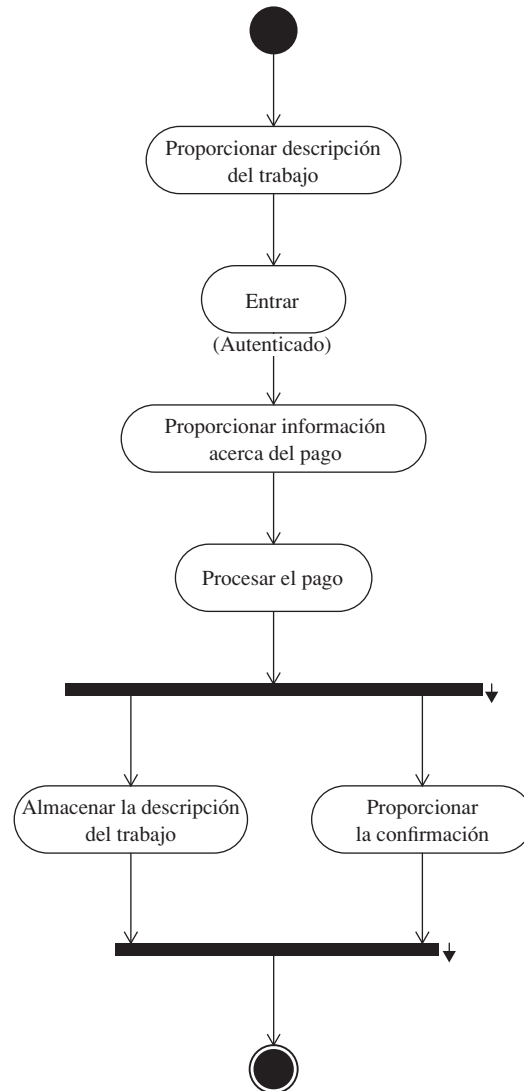


Figura 3-10 Un modelo en el que se muestra la acción requerida para publicar un trabajo.

Un beneficio real implícito aquí es que un intento razonable en un diagrama de actividades capta la comprensión del modelador y permite a otros proporcionar retroalimentación y desarrollar el flujo, agregando o eliminando detalles, según sea necesario.

Manera de agregar condiciones previas y posteriores

Se pueden agregar condiciones previas o posteriores a un modelo con el uso de una nota: los símbolos de estereotipo con las palabras *condición previa* o *condición posterior* en

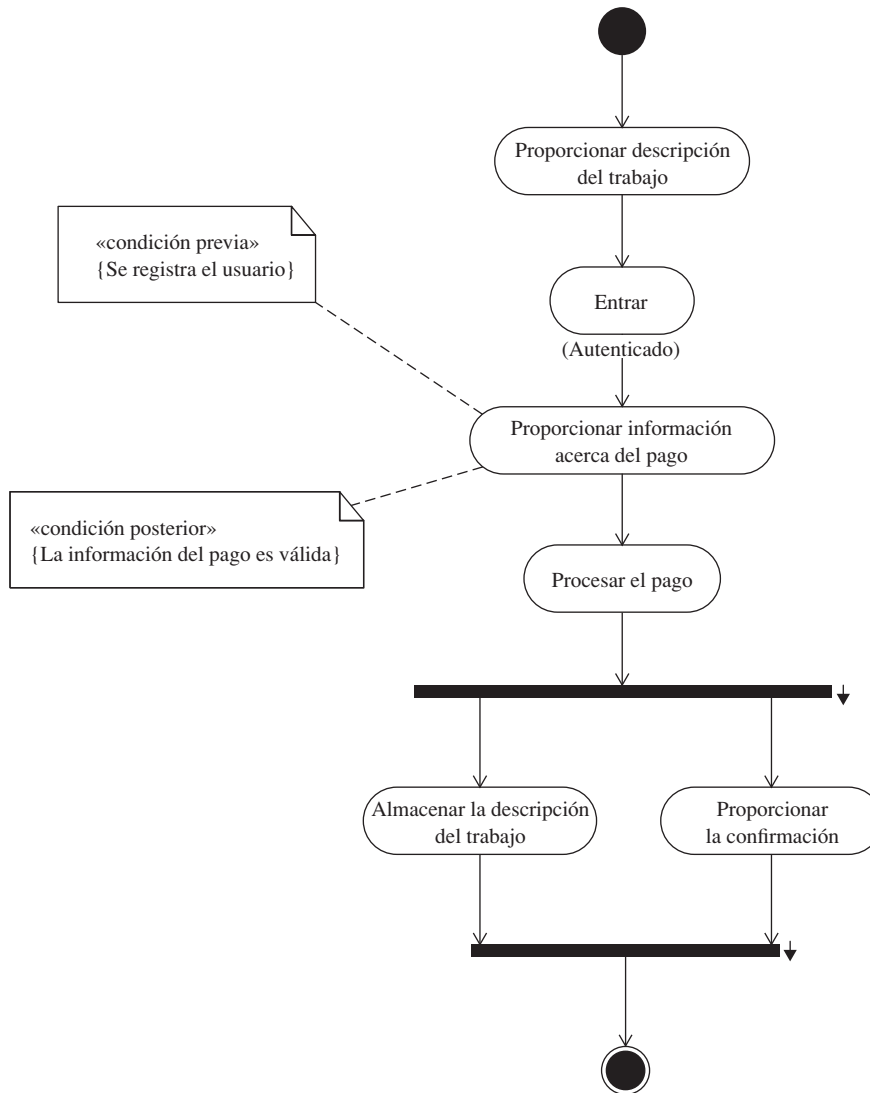


Figura 3-11 Uso de una restricción de condición previa y de condición posterior.

su interior y el nombre de la condición. La nota se agrega a la acción a la cual se aplica la condición o las condiciones. Esto se conoce como *diseño por contrato* y, con frecuencia, se implementa en el código como una afirmación combinada con una prueba condicional. La figura 3-11 muestra una condición previa y una posterior aplicadas a la acción “Proporcionar información acerca del pago”.

En la figura 3-11, el diagrama requiere la condición previa de que el usuario se registre y la posterior de que la información del pago sea válida. Como sucede con el código,

existe más de una manera de representar esta información. Por ejemplo, podríamos usar una condición guardián antes y después de la acción “Proporcionar información acerca del pago” (figura 3-12), o podríamos usar un nodo de decisión (vea “Nodos de decisión y de fusión”) para ramificar hacia una acción de registrar, antes de permitir el suministro de esa información, y podríamos tener una acción para validar la información antes mencionada, después de que se suministre (figura 3-13).

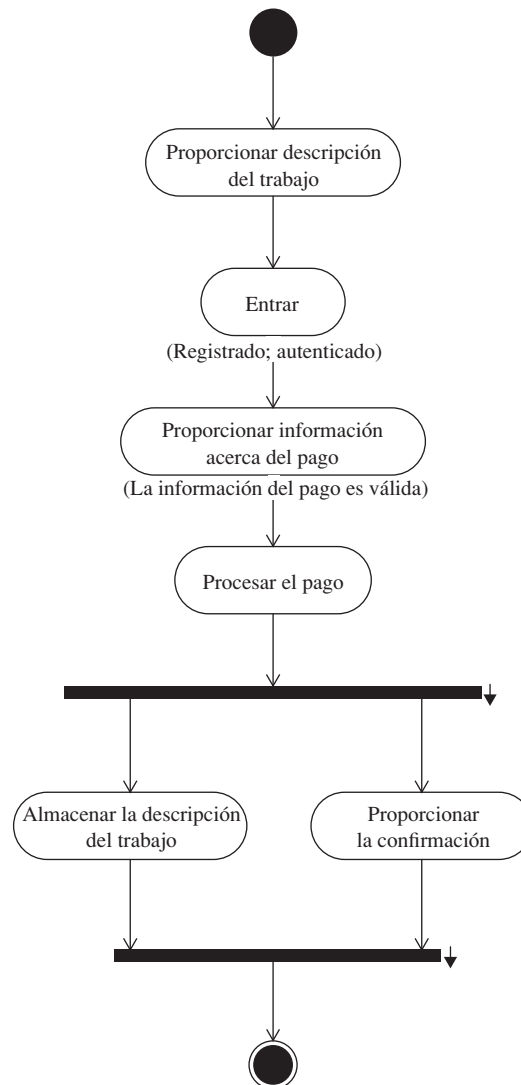


Figura 3-12 Uso de guardianes para expresar una condición previa y una posterior.

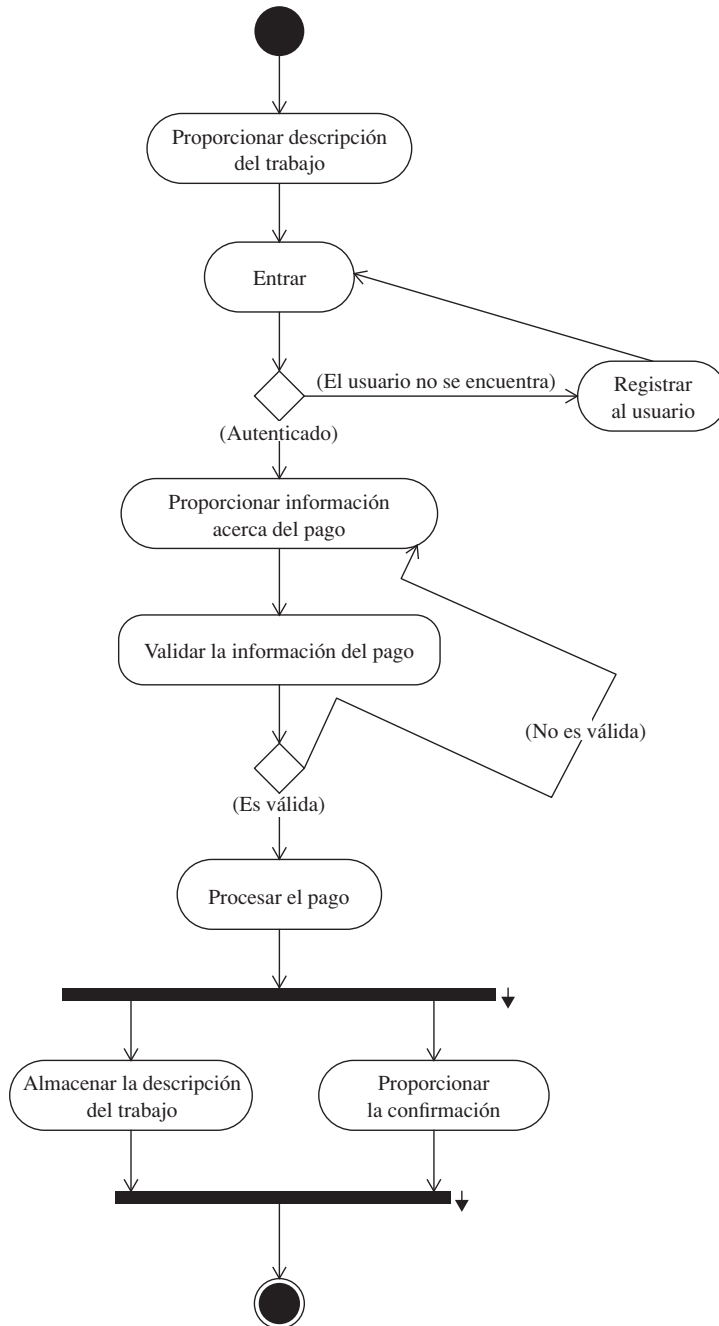


Figura 3-13 Uso de un nodo de decisión para indicar que los usuarios deben registrarse y proporcionar información válida del pago.

Estos tres diagramas —figuras 3-11, 3-12 y 3-13— transmiten la misma información. La diferencia real es de estilo. Si quiere que el diagrama aparezca menos ocupado, trate de usar la condición guardián. Si el estilo de restricción —de la figura 3-11— parece más significativo, entonces use ese estilo. Si quiere examinar el registro y la validación de la dirección, entonces use los estilos de nodos de decisión de la figura 3-13, donde los nodos de decisión están representados por los símbolos con forma de diamante.

Modelación de las subactividades

A veces es fácil agregar demasiado detalle a un solo diagrama de actividades, lo que lo hace ocupado y confuso. Por ejemplo, si desarrollamos “Registrar al usuario” de la figura 3-13, para incluir todas las acciones necesarias para registrar los usuarios, como la obtención de un nombre y contraseña únicos del usuario, y validar y almacenar la información de la dirección de correo, entonces se puede perder el enfoque principal de la actividad —creación de una lista de trabajo y pagar por ella— en el ruido de todas las acciones y estímulos adicionales.

Si, en cualquier caso, hallamos que los detalles de las subactividades hacen que un diagrama sea demasiado confuso, o encontramos que queremos volver a usar las subactividades, entonces podemos marcar una acción como una subactividad con una bifurcación en su interior. (Visio no permite el símbolo de subactividad subsidiaria, de modo que extraje uno de los garabatos de Paint de Microsoft y lo agregué a la acción “Registrar el usuario” de la figura 3-13.)

SUGERENCIA *Si quiere inventar o encuentra que un aspecto del UML no permite su herramienta específica de modelado, entonces considere la posibilidad de usar un estereotipo o una nota para documentar lo que quiere usted dar a entender.*

Nodos de decisión y de fusión

En los diagramas de flujo, a los nodos de decisión y de fusión se les llamaba *diamantes de decisión*. Este símbolo con forma de diamante es uno de los elementos que hace que el diagrama de actividades sea una reminiscencia de un diagrama de flujo. Los nodos de decisión y de fusión usan el mismo símbolo y transmiten la ramificación y la fusión condicionales.

Cuando el símbolo con forma de diamante se usa como *nodo de decisión* —después de “Entrar” en la figura 3-13— tiene un estímulo que entra al nodo y múltiples estímulos saliendo de éste. Cuando se usa como *nodo de fusión*, hay múltiples estímulos entrando y solamente uno saliendo. Un nodo de decisión sólo toma un camino de salida, y uno de fusión no tiene salida hasta que todos los flujos han llegado al mismo.

Las condiciones guardianes en un nodo de decisión actúan como la lógica si... de otro modo y deben ser mutuamente excluyentes, lo cual por necesidad implica que si se sa-

tisface una de ellas, entonces la otra debe fallar. Como se describe en la figura 3-13, puede estipular las dos condiciones guardianes literalmente, o estipular una de ellas y usar una guardián [De otro modo] para la condición alterna.

Un nodo de fusión marca el final del comportamiento condicional iniciado por un nodo de decisión. En la figura 3-13, no necesitamos un nodo de fusión porque reencaminamos al usuario recientemente registrado de regreso a la acción “Entrar”. No obstante, si quisiéramos ser un poco más amables, podríamos sencillamente autenticar al nuevo usuario en forma automática y seguir directamente a proporcionar la información acerca del pago, en donde el usuario la dejó. Esta revisión se muestra con el uso de un nodo de fusión en la figura 3-14. (Observe que se modificaron las condiciones guardianes para el nodo de decisión que está después de la acción “Entrar”, para mostrar el uso del estilo de guardián [De otro modo].)

Bifurcaciones y uniones de transición

Una *bifurcación* existe para describir comportamiento paralelo, y se usa una *unión* para hacer convergir el comportamiento paralelo de regreso a un solo flujo. En el comportamiento bifurcado no se especifica si ese comportamiento se intercala o no, o bien, si ocurre en forma simultánea; la implicación es solamente que están ocurriendo acciones bifurcadas en el transcurso de un intervalo compartido y concurrente. Se suele implementar el comportamiento bifurcado como comportamiento multiencajonado. (En la figura 3-13, se presenta un ejemplo de una bifurcación después de la acción “Procesar el pago” y una unión inmediatamente antes del nodo final.)

Cuando múltiples flujos entran a una acción, ésta, de manera implícita, es una unión y el significado es que *sólo* hay flujo saliente cuando todos los flujos entrantes han llegado a la acción. Sus diagramas serán más claros si usa bifurcaciones y uniones de manera explícita en donde quiera dar a entender que se muestra comportamiento paralelo.

En la figura 3-13, quisimos decir que podemos almacenar una descripción del trabajo y proporcionar al usuario una confirmación, en forma simultánea o de manera concurrente, pero estas dos cosas deben ocurrir antes de que se considere que se ha completado la actividad.

Partición de la responsabilidad con carriles

A veces usted quiere mostrar quién o qué es responsable de un parte de una actividad. Puede hacer esto con *carriles*. Lo común es que las herramientas de modelado muestren los carriles como un cuadro con un nombre en la parte superior y que usted coloque cualesquiera nodos y estímulos que pertenecen a esa cosa en ese carril. Usted puede tener tantos carriles como sea conveniente, pero los carriles encajonados pueden dificultar la organización de su diagrama de actividades.

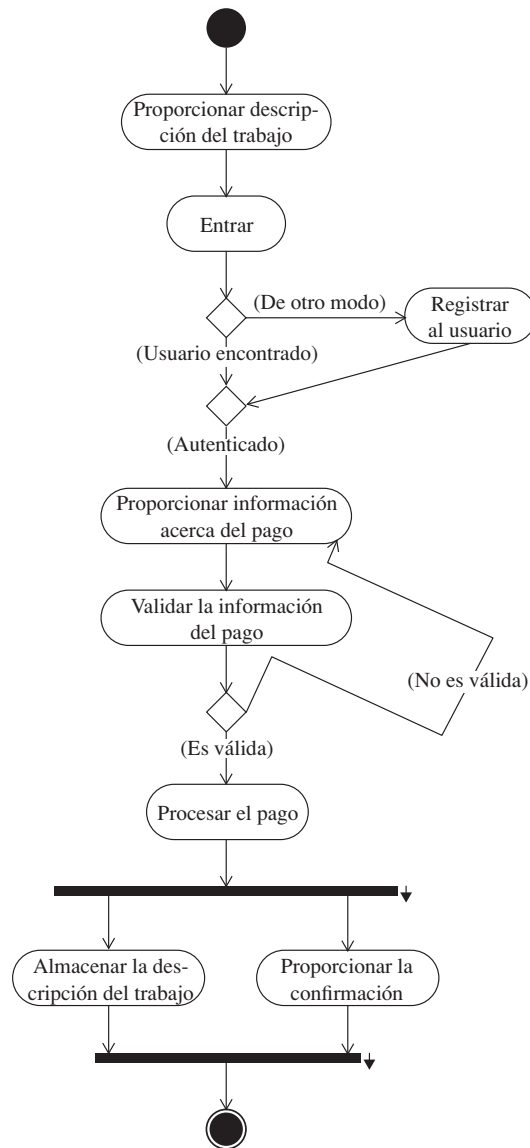


Figura 3-14 Nodo de fusión usado para hacer convergir cuando se toma una rama después de que se registra un nuevo usuario.

El UML versión 2 permite particiones verticales, horizontales y como rejilla, de modo que la metáfora carril ya no es precisa. La terminología real ahora es *partición de la actividad*, pero todavía se emplea la palabra *carril* en la conversación general y se usa en las herramientas de modelado.

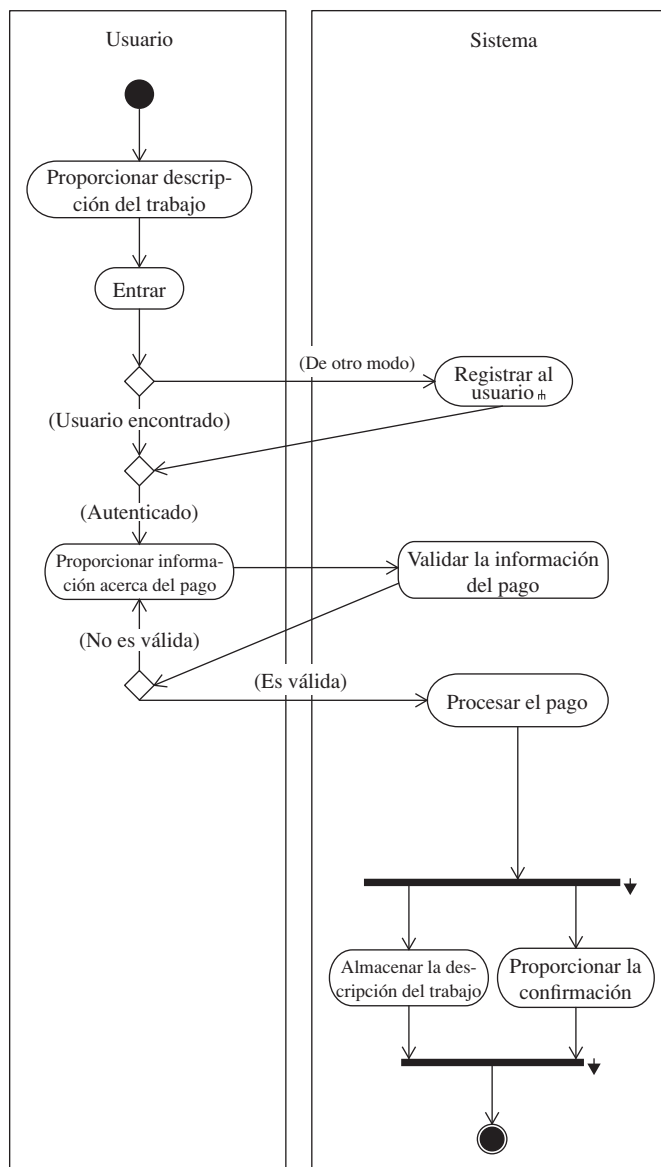


Figura 3-15 Las acciones se dividen entre un usuario y el sistema.

Uso de los carriles

Si en la figura 3-14 queremos mostrar quién o qué es responsable de varias acciones, entonces podemos agregar un carril (o partición) para lo que creemos que son las particiones. En el ejemplo, podríamos decir que la publicación de un trabajo se divide en

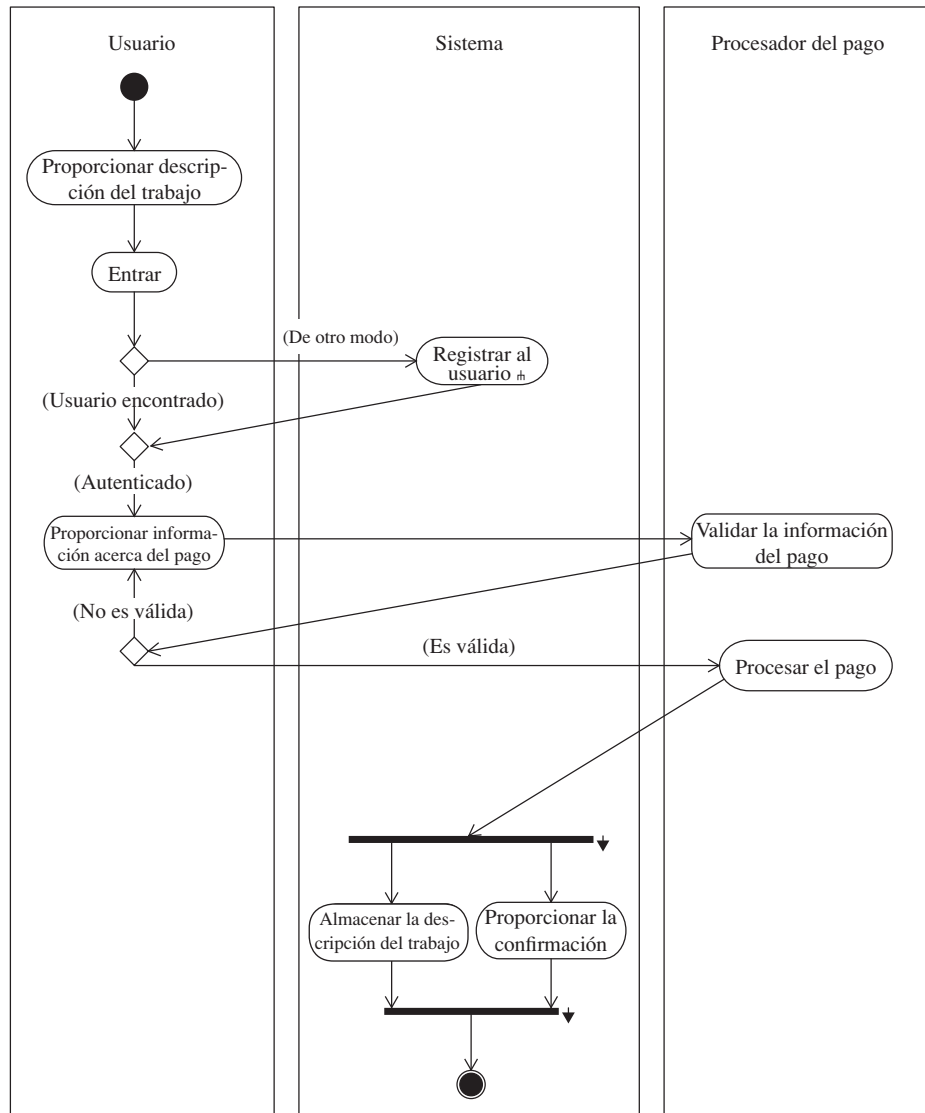


Figura 3-16 Subdivisión adicional de las responsabilidades mediante el reemplazo de las acciones “Validar la información del pago” y “Procesar el pago” en una partición separada con el nombre de “Procesador del pago”.

dos particiones: el usuario y el sistema, y agregamos un carril para cada partición (figura 3-15). Si decidimos que el procesamiento del pago representa una partición distinta, entonces podríamos agregar una tercera partición y trasladar la acción de procesar el pago hacia esa partición (figura 3-16).

Como se hace con la programación, puede dividir su análisis y diseño en tantas particiones como quiera. Se debe aceptar dar algo a cambio por agregar particiones en los modelos, precisamente como ha de aceptar dar algo a cambio por agregar particiones en el código. Partir los modelos puede ayudarle a organizar, pero todas esas particiones sugieren software partido que deberá orquestarse y reensamblarse para lograr las metas del sistema.

Modelado de acciones que se extienden sobre las particiones

A veces una acción puede pertenecer a más de una partición al mismo tiempo. Por ejemplo, “Registrar al usuario” en realidad no pertenece al usuario o al sistema. Sabemos, con base en las exposiciones anteriores, que “Registrar al usuario” es una actividad subsidiaria que puede comprender al usuario que proporciona la información personal y al sistema que valida la información de la dirección y almacena la información de ese usuario. Sin embargo, el UML no permite que un nodo se extienda en más de una partición en una sola dimensión. Como resultado, usted deberá elegir una partición para el nodo, y esto también sugiere que sabemos lo que se cumple acerca de “Registrar al usuario”; es decir, se puede descomponer en su propia actividad.

Uso de particiones multidimensionales

El modelado de particiones multidimensionales de actividades es un concepto relativamente nuevo. Parece que algunas herramientas de modelado populares y de las que se dispone en la actualidad no permiten diagramar particiones multidimensionales de actividades; sin embargo, puede simular una partición multidimensional en Visio agregando dos carriles (particiones de la actividad) y hacer girar uno de ellos. (El resultado es un diagrama semejante al de la figura 3-17.) Ahora que tenemos la mecánica para crear una partición multidimensional, podría usted preguntarse cómo se usa.

Una acción en una matriz de particiones de una actividad pertenece por completo a las dos particiones. Suponga, por ejemplo, que como estamos preparándonos para vender listas de trabajos en Motown-jobs.com, decidimos usar PayPal para procesar los pagos. Podemos decir que “Procesar el pago” es parte tanto de nuestro “Procesador del pago” como del sistema de procesamiento de pagos de PayPal, lo cual se refleja en la figura 3-17.

Indicación de las señales cronometradas

Hasta ahora no hemos hablado acerca de cuándo ocurren las cosas. Existen tres tipos de señales que facilitan hablar acerca del tiempo en los diagramas de actividades. Éstas son la *señal de tiempo*, la *señal de enviar* y la *señal de aceptar*. Una señal indica que se ha lanzado un evento exterior y ese evento inicia la actividad.

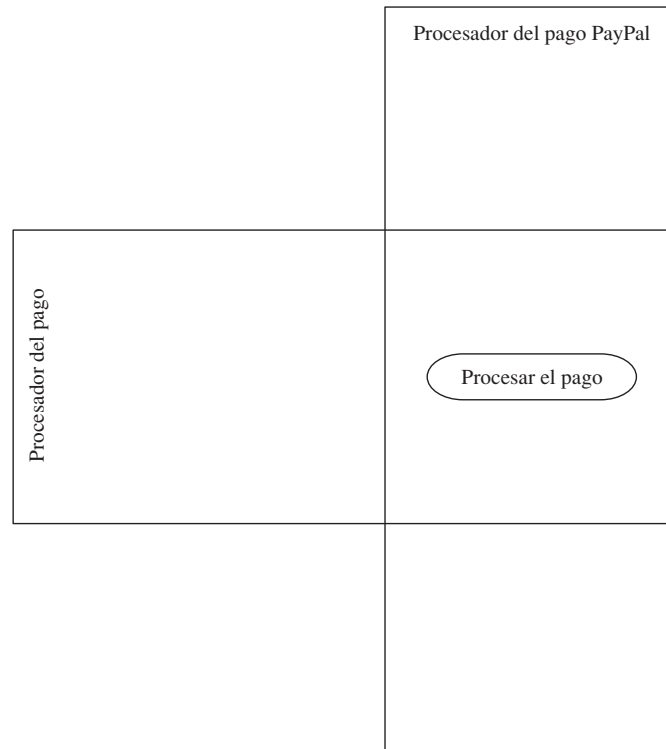


Figura 3-17 Particiones multidimensionales, en donde dos particiones en diferentes dimensiones poseen una acción al mismo tiempo.

Se usa la forma de reloj de arena de la señal de tiempo para especificar un intervalo de tiempo. Por ejemplo, podríamos usar la señal de tiempo para indicar que se iniciará la actividad “Hacer caducar la lista” después de que la lista haya estado disponible durante 30 días (figura 3-18). El símbolo de señal de recibir es un rectángulo con una muesca cortada, y el de señal de enviar es un rectángulo con una punta sobresaliente, lo que hace que los símbolos señales de recibir y de enviar luzcan un poco como las piezas de un rompecabezas (una vez más, mostrados en la figura 3-18).

NOTA Toda herramienta tiene sus limitaciones. En Visio, por ejemplo, no existe símbolo para una señal de tiempo, de modo que inventé una, y las señales de enviar y recibir se usan como una forma alternativa de documentar eventos. La implementación de Visio no es precisamente coherente con el UML; es importante no quedarse colgado en estas pequeñas incoherencias por las que usted está obligado a pasar. En vez de consumir su tiempo en dibujar imágenes para aspectos no soportados del UML, trate de usar en su lugar una nota.

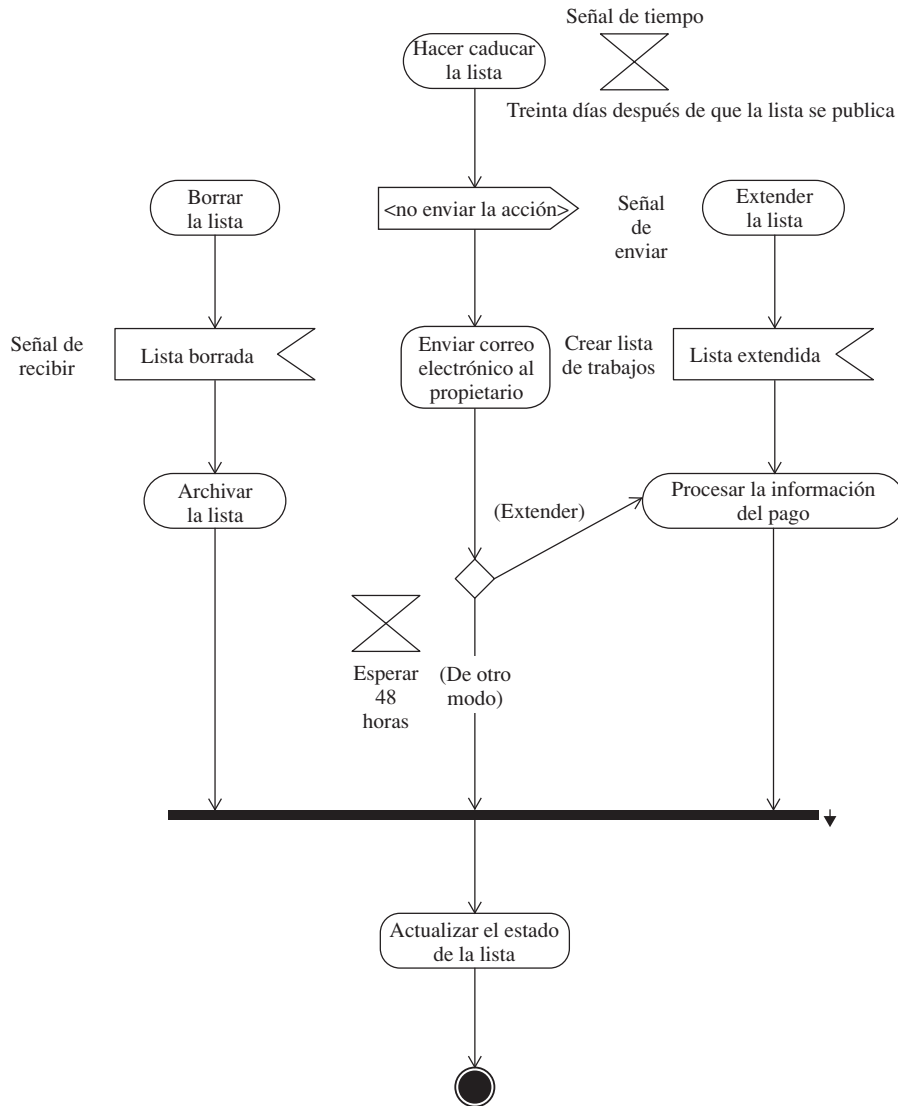


Figura 3-18 Señal de tiempo para hacer caducar una lista, dos señales de recibir para extender y borrar una lista, y una señal de enviar para notificar que una lista está próxima a caducar.

Se entiende que el modelo de la figura 3-18 quiere dar a entender que 30 días después de que se publica una lista, se la hará caducar de manera automática, a menos que un propietario notificado elija extenderla. Las señales alternas incluyen un usuario que borra una lista, lo cual hace que la lista se archive antes de quitarse, y un propietario que habla por propia iniciativa para extender la lista antes de su caducidad. Si el propietario extiende la lista, entonces esto envía una señal al sistema para que procese un pago adicional.

Configuración de los parámetros de entrada

Los diagramas de actividades pueden tener parámetros de entrada, como en la figura 3-18, en cada caso en que hablamos de hacer algo con una lista. Podríamos mostrar un objeto “Lista” como entrada para cada acción en la figura. Tomando sólo una pequeña porción de la figura 3-18, podemos mostrar la notación y el símbolo para indicar que la entrada a la acción es un objeto “Lista” (figura 3-19).

En tanto que los objetos de entrada pueden resultar útiles para los desarrolladores, éste es otro caso en donde pueden agregar confusión para la discusión de la actividad en un sentido general y analítico. Al menos en el curso de las primeras fases del análisis, considere aplazar la referencia específica a los detalles de la implementación, como las clases.

Forma de mostrar las excepciones en los diagramas de actividades

El UML permite el modelado de *excepciones*. Una excepción se muestra como una línea zigzagueante (o “rayo”) con el nombre de la clase de la excepción que la adorna. El ma-

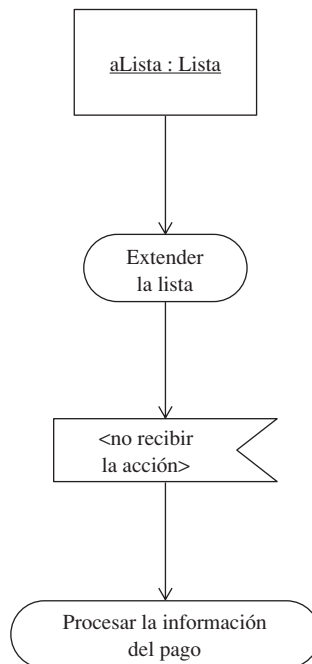


Figura 3-19 Se muestra el objeto “Lista” como un parámetro de entrada a la acción “Extender la lista” y el diagrama de actividades que lo contiene.

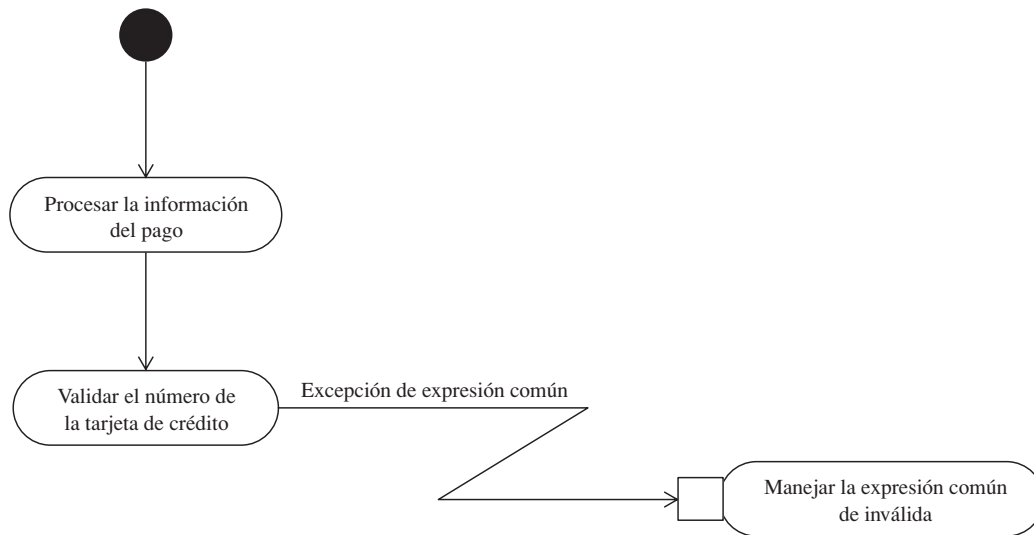


Figura 3-20 Modelado de una excepción en un diagrama de actividades.

nerador de la excepción se puede modelar como un nodo de acción con el nombre de la acción en el mismo y el flujo de excepción conectado a una clavija de entrada en el nodo de acción de la excepción (figura 3-20).

El nodo que contiene el manejador de la excepción no tiene flujo de retorno. Un manejador de excepción sólo descuelga la acción que causó que ocurriera el error. Es importante recordar que estamos captando flujo y acciones generales; en el curso de esta fase, no necesitamos indicar cómo estamos manejando la excepción.

Los conceptos como excepción, manejador de excepciones, despliegue de la pila y rendimiento pueden agregarse de manera considerable a la confusión para los usuarios no técnicos. Si puede agregar una excepción y su nodo de acción sin atascarse en discusiones acerca de cómo se implementan los manejadores de excepciones o cómo funcionan, entonces siga adelante y agréguelos a sus diagramas de actividades.

Terminación de los diagramas de actividades

Cuando llegue al final de una actividad, agregue un *nodo final de actividad*. Si llega al final de un flujo y no sucede algo más, agregue un *nodo final de flujo* (figura 3-21). Puede tener más de un nodo final de actividad y de un nodo final de flujo en un solo diagrama de actividades.

El diagrama de actividades de la figura 3-21 muestra que procesamos todas las listas caducas hasta no tener más y, para cada lista caduca, enviamos un correo electrónico al propietario, dándole una oportunidad de renovar la lista o dejarla caducar. Advierta que

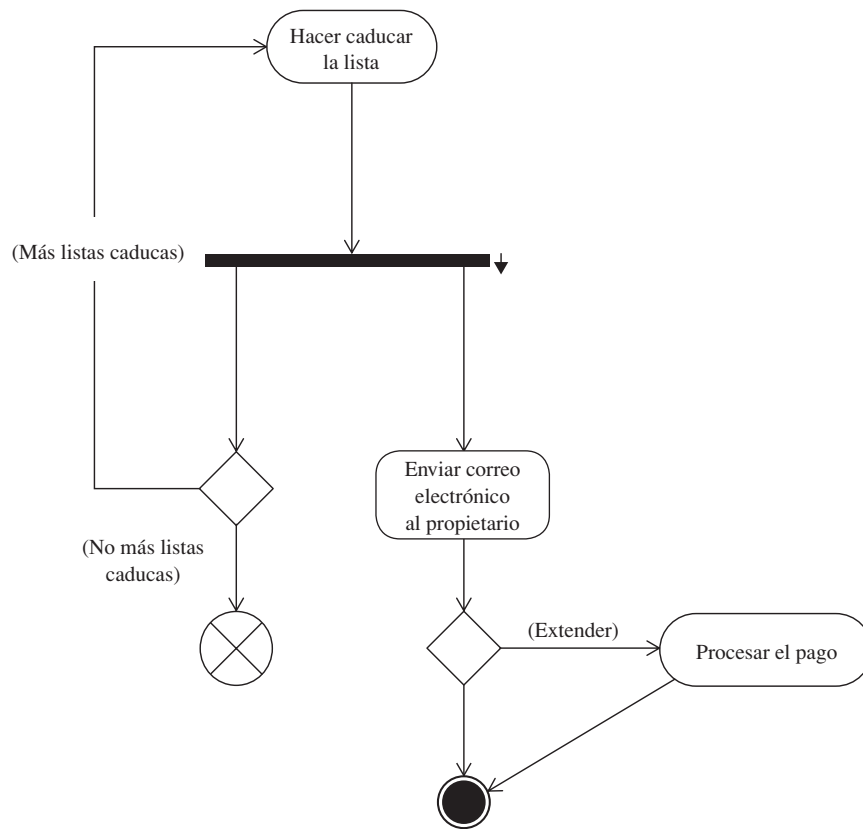


Figura 3-21 Actividad en la que se muestra un nodo final de flujo y un nodo final de actividad.

cuando se ramifica el nodo de decisión, debido a que ya no hay más listas caducas, sencillamente va hacia un extremo cerrado. Podría usted imaginarse esta suerte de actividad implementada como un proceso asíncrono, en donde cada lista caduca se expulsa de un proceso para permitir que su propietario la renueve.

Creación de los diagramas de actividades

Una decisión tan importante como qué incluye un diagrama de actividades es qué diagramar. Con demasiada frecuencia es fácil mantenerse agregando modelos adicionales y añadiendo más detalles a los modelos existentes; no obstante, la implicación es que mientras usted está modelando algo, alguien más está esperando para implementar su diseño o, lo que es peor, mientras usted está refinando sus diseños, algún pobre encargado de

implementar tendrá que modificar la implementación de esos diseños. Por esta razón, es importante hacer que sus diagramas de actividades sean relativamente sencillos; limite la creación de diagramas de actividades a lo importante, crítico o a los aspectos retadores de su problema, y evite tratar de hacerlos perfectos. Un buen modelo fácil de comprender y que se presenta oportunamente es más valioso que un modelo perfecto posterior; si es que existe tal cosa como un modelo perfecto.

Ejemplos de los diagramas de actividades que yo crearía para los casos de uso del capítulo 2 podrían ser una actividad para “Mantener la lista de trabajo”, “Hacer caducar la lista de trabajo” y “Mantener la información sobre facturación”. En especial, estoy interesado en entender los aspectos críticos del sistema, de manera particular aquellos para los servicios que son elementos susceptibles de facturación. Cosas comunes como buscar o registrar están suficientemente bien entendidas, de tal manera que es improbable que yo creara un diagrama de actividad para ellas.

Seleccionar lo que debe modelarse y lo que no, es algo semejante a añadir sal cuando se está cocinando: siempre puede agregar un poco más, pero es difícil eliminar la sal si ha añadido demasiada. Lo mismo es verdad con el modelado: no puede recuperar el tiempo consumido en modelar actividades obvias, pero siempre puede agregar diagramas de actividades más adelante, si es necesario.

Reingeniería del proceso

Es posible que el uso más benéfico de los diagramas de actividades sea ayudar al personal que no pertenezca al dominio —por lo común, los tecnólogos que implementarán una solución— a comprender este dominio. En lo anterior, está implícito que, mientras los expertos del dominio y los tecnólogos están intentando llegar a una comprensión común, existe una oportunidad de realizar la reingeniería del proceso. Tomemos un momento para revisar lo que se quiere dar a entender por *reingeniería de procesos*.

Con frecuencia, la gente realiza su trabajo de manera cotidiana sin jamás identificar un proceso formal. El conocimiento del proceso lo tienen sólo los profesionales. A menudo, estas mismas organizaciones reciben un choque al descubrir cuántos gastos generales y desperdicio existe en la forma en que están organizadas. La reingeniería de procesos es una suerte de pseudociencia que conlleva, en primer lugar, la documentación de los procesos de una organización y, en segundo, a buscar maneras de optimizar esos procesos.

No soy un experto en reingeniería de procesos, pero existen ejemplos históricos en donde empresas bien conocidas han gastado una cantidad considerable de dinero y energía para refinar los procesos de sus empresas, y los resultados han conducido a cambios amplios y arrasadores en la industria. Se puede hallar un ejemplo interesante en *Behind the Golden Arches*, en el cual se detalla el camino de evolución que condujo a McDonald's a aplicar la distribución centralizada para sus franquicias.

NOTA Resulta bastante irónico que el propio desarrollo de software sea un ejemplo de un dominio en donde los profesionales han definido el proceso de una manera ad hoc. Muchas empresas de software ahora están empezando a darse cuenta de que están muy atrasadas en relación con un examen introspectivo de los procesos que siguen al proceso estructurado. ¿Alguien en su organización ha usado alguna vez un diagrama de actividades (o un diagrama de flujo) para documentar la manera en que se estructura su software?

El desarrollo de software es un asunto de automatizar soluciones para los problemas. En un sentido general, es una idea útil documentar los procesos críticos del dominio y examinar algunas optimizaciones posibles, antes de escribir el código. Si se simplifica el proceso, también se puede simplificar marcadamente la implementación subsiguiente.

Reingeniería de una subactividad

He aquí un ejemplo que comprende una subactividad llamada “Verificación de la cabina interior”, que se relaciona con la inspección previa al vuelo de un avión pequeño. La idea que se encuentra detrás de la verificación de la cabina interior es que estamos buscando cosas necesarias o importantes en el interior del avión y realizando pasos para ayudar a algunas verificaciones exteriores. Hay una probabilidad muy buena de que si pasamos por alto algo, entonces podríamos realizar el despegue en condiciones inseguras o no contar con recursos críticos durante una urgencia. (Si le molesta que esto no suene como un problema de software, entonces sólo imagine que estamos documentando este problema para escribir software de simulación o para realizar las pruebas.)

Uno de los aviones que vuelo es un Cessna 172 Skyhawk. La verificación de la cabina interior (descrita en la figura 3-22) consiste en

- Asegurarse de que el interruptor de ignición esté en posición de apagado
- Hacer girar el interruptor maestro hacia la posición de encendido de modo que tengamos energía
- Bajar los alerones
- Verificar la existencia del registro, del certificado de que el avión está en condiciones de volar, del peso y balance de la información y del manual de operaciones, el cual incluye los procedimientos de urgencia
- Verificar los indicadores de nivel del combustible y el selector de este último
- Hacer girar el interruptor maestro hacia la posición de apagado

Como se muestra en el diagrama de actividades de la figura 3-22, los pasos se llevan a cabo de manera consecutiva. (Ésta es la manera en que realicé la inspección el primer

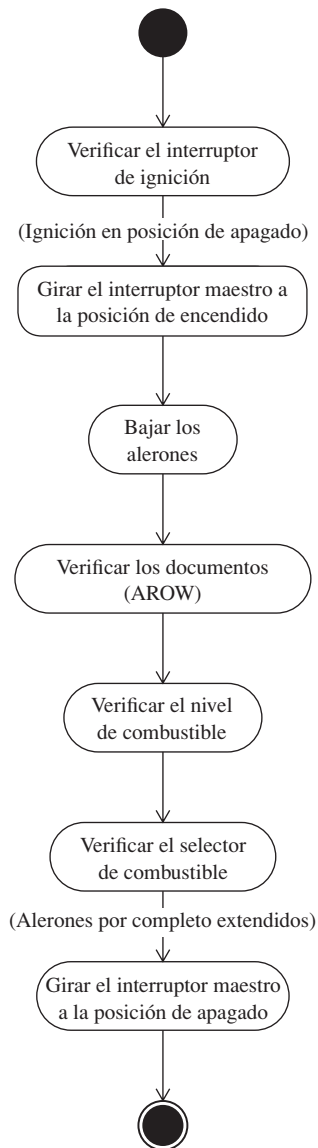


Figura 3-22 Nuestra comprensión inicial es que cada una de las tareas de la actividad se realiza en forma consecutiva.

par de veces que la llevé a cabo.) Un piloto experimentado (un experto del dominio) le dirá que se requieren unos cuantos instantes para que los alerones bajen, de modo que algunas de las otras verificaciones se pueden llevar a cabo en forma simultánea. Podemos ajustar el diagrama de actividades como se muestra en la figura 3-23.

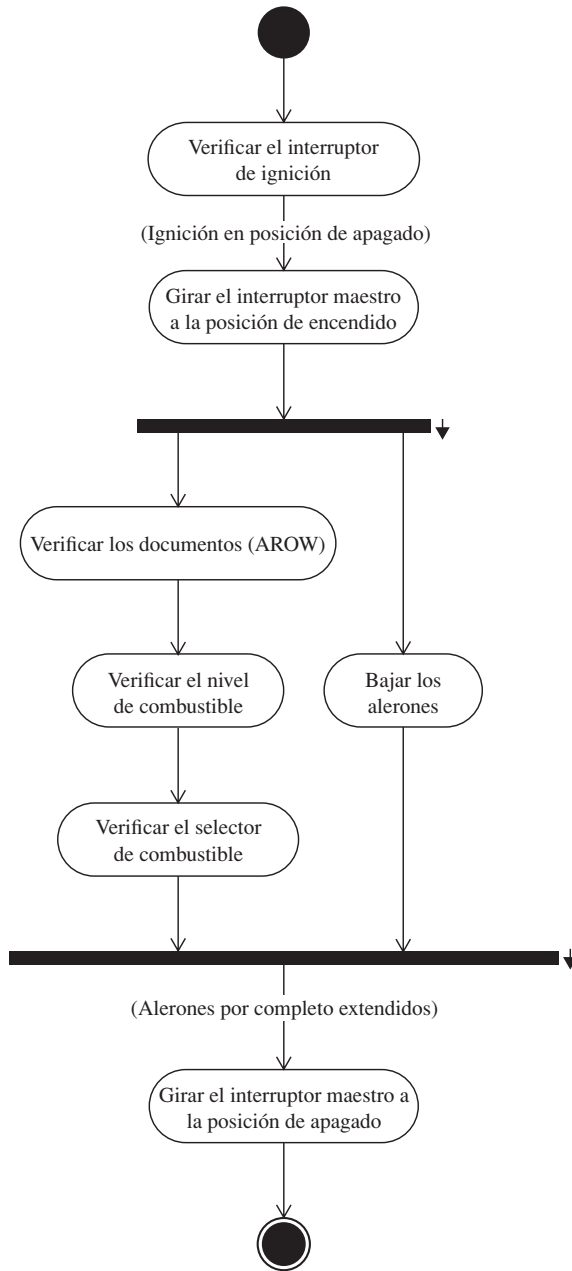


Figura 3-23 Hacer que algunas tareas se realicen de manera simultánea mejorará el tiempo para la compleción de la actividad.

Saber cuándo renunciar

La aplicación de las reglas de manera uniforme le ayudará a trabajar con eficiencia en el transcurso de la fase de modelado del desarrollo. Con esto presente, recuerde que dije que una idea importante es captar los caso de usos más críticos y abordarlos primero. Lo mismo se cumple para los diagramas de actividades. Identifique los casos de uso más críticos y cree los diagramas de actividades para aquellos que requieren algo de examen. Por ejemplo, es necesaria la autenticación de un usuario en Motown-jobs.com, pero éste es un problema bien entendido. Yo no pasaría una gran cantidad de tiempo creando un diagrama de actividades para esto y no trabajaría en él antes que en aquellos relacionados con mi caso de uso primario, “Administrar una lista de trabajo”.

Si no está seguro de cuántos diagramas de actividades debe crear, entonces intente crear un diagrama de actividades para cada una de las funciones primarias de sus casos de uso más importantes. Intente tener tanto acerca de la actividad modelada con tanta rapidez y tanta exactitud como pueda. De inmediato, regrese para comprobar con sus expertos del dominio y examine las actividades para ver si ha captado los puntos más sobresalientes.

Por último, no permita quedarse atascado aquí. Si no puede llegar a un consenso sobre lo completo de una actividad particular, entonces déjela a un lado y acuerde en regresar a ella. Puede haber otros elementos del problema que aumentarán su comprensión, o la de sus usuarios acerca del problema en general para resolver aquél que dejó a un lado. La clave es no quedarse atascado demasiado pronto en cualquier problema particular.

Examen

1. Los sinónimos para transición son
 - a. conector y flujo.
 - b. estímulo y flujo.
 - c. estímulo y conector.
 - d. acción y evento.
2. En general, los diagramas de actividades constan de
 - a. nodos y estímulos.
 - b. acciones y transiciones.
 - c. acciones, decisiones y flujos.
 - d. símbolos y líneas.

3. Se puede mostrar una excepción en un diagrama de actividades con un estímulo en forma de rayo.
 - a. Verdadero
 - b. Falso
4. En un nodo de decisión y en uno de fusión se usan
 - a. símbolos diferentes.
 - b. símbolos idénticos.
 - c. símbolos idénticos o diferentes, dependiendo del contexto.
5. Los flujos múltiples que entran a un nodo de acción constituyen
 - a. una fusión implícita.
 - b. una unión implícita.
6. Todo flujo espera en una fusión y una unión hasta que todos los flujos hayan llegado.
 - a. Verdadero
 - b. Falso
7. La metáfora de carril ya no se usa
 - a. porque los carriles ya no son parte del UML.
 - b. porque las particiones pueden ser multidimensionales y no se ven como carriles.
 - c. Todavía se usa la metáfora de carril.
 - d. Tanto b como c.
8. Pueden existir acciones en dos particiones de una actividad al mismo tiempo, en dimensión diferente.
 - a. Verdadero
 - b. Falso
9. Un nodo de decisión y uno de fusión se representan por
 - a. un óvalo.
 - b. un círculo.
 - c. un rectángulo.
 - d. un diamante.

10. Los diagramas de actividades son diferentes de los de flujo porque los de actividades permiten
- a. carriles.
 - b. comportamiento paralelo.
 - c. nodos de decisión.
 - d. acciones.

Respuestas

- 1. b
- 2. a
- 3. a
- 4. b
- 5. b
- 6. a
- 7. d
- 8. a
- 9. d
- 10. b

Comportamientos con diagramas de interacción

Desmitificar significa “exponer algo, hacerlo directo o lanzar luz sobre ello” y en cada capítulo se hace esto implícita o explícitamente. En este capítulo, me gustaría empezar mostrándole el camino ahora mismo. Hay varios tipos de diagramas de Unified Modeling Language (UML); algunos son redundantes y, definitivamente, no necesita crear todo tipo de diagramas para tener un buen diseño. Existe más de un tipo de diagramas de interacción y la regla de evitar la redundancia es de lo más pertinente para este capítulo.

Los dos diagramas comunes de interacción son los *diagramas de secuencia* y los de *colaboración* (o *comunicación*). Estos diagramas le dicen exactamente lo mismo. Las secuencias tienen un ordenamiento explícito en el tiempo y son lineales, y las colaboraciones tienen un ordenamiento “rotulado” en el tiempo y son geométricas. Sólo necesita uno u otro, pero definitivamente no ambos.

Me gustan los diagramas de secuencia; son más comunes, muy fáciles de crear y están organizados de manera natural, y no necesitamos indicar el ordenamiento en

el tiempo mediante la anotación de los mensajes. Como consecuencia, en este capítulo haré hincapié en el diagrama de secuencia, pero hablaré con brevedad (y mostraré) los diagramas de colaboración para que usted se familiarice con ellos. (Si, finalmente, decide que le gusta la organización geométrica de los diagramas de colaboración, entonces úselos. Sin embargo, recuerde que no necesita tanto secuencias como colaboraciones, y muchas herramientas UML convertirán con facilidad, en forma automática, las secuencias en colaboraciones y viceversa.)

En este capítulo, le mostraré cómo

- Identificar los elementos de los diagramas de secuencia
- Crear diagramas de secuencia y de colaboración
- Comprender el ordenamiento en el tiempo de los diagramas de interacción
- Usar los diagramas de interacción para descubrir clases y métodos
- Modelar escenarios de éxito y falla con el uso de los marcos de interacciones introducidos en el UML versión 2.0
- Usar secuencias para examinar el comportamiento de muchos objetos de uno a otro lado del caso de uso

Elementos de los diagramas de secuencia

En todo diagrama sólo se usa un subconjunto de los símbolos y de la gramática que constituyen el UML. El aprendizaje acerca de esos símbolos y de la gramática específica es un mal esencial. Es importante tener en cuenta que no necesita recordar todas las palabras de un lenguaje para comunicarse en forma eficaz; no puedo recordar con precisión qué significa *solecismo*, como en “porque es el solecismo de un príncipe pensar en controlar el fin y, sin embargo, no soportar el medio”; pero es importante dominar un lenguaje para emplearlo en forma creativa.

NOTA *Es importante recordar que el UML es un lenguaje en evolución. Como con los lenguajes hablados, se puede tener comunicación eficaz con una comprensión básica del lenguaje. La clave es recordar que hay que dejar las leyes del lenguaje a otros. (En este caso, deje las leyes del lenguaje al Object Management Group.)*

Tomemos un par de minutos para examinar los símbolos y la gramática útiles de los diagramas de secuencia. Empezaremos con los elementos básicos y esenciales de los diagramas de secuencia: las líneas de vida y los mensajes. (Vale la pena hacer notar que se puede tener un diálogo aceptable con sólo estos dos elementos de los diagramas de secuencia.)

Uso de las líneas de vida de objetos

Una *línea de vida* es un rectángulo con una recta vertical que desciende de ese rectángulo. La línea de vida representa un ejemplo de una clase, y la línea que desciende en forma vertical es un lugar conveniente para sujetar mensajes entrantes y salientes. Agregar múltiples líneas de vida a un solo diagrama y sujetarles mensajes ordenados en el tiempo le permiten mostrar todas las clases y los mensajes necesarios para completar un escenario descrito por un caso de uso. Mediante la eliminación de brechas ambiguas o evitando la repetición de clases y mensajes, puede obtener una solución completa, un escenario a la vez.

Una línea de vida de un objeto toma forma como un objeto que representa una parte de un papel en un caso de uso. Hablaré más acerca de las líneas de vida conforme avancemos; por ahora, sólo observe el símbolo de la figura 4-1.

Las líneas de vida de objetos pueden representar actores u objetos. Todos los actores y objetos pueden actualizarse o no como código. Esto puede sonar confuso, pero no lo es. Suponga, por ejemplo, que estamos estructurando un sistema de reservaciones de boletos para una línea aérea. Un actor podría ser una persona que trabaje en el mostrador en la terminal o en un quiosco [usado para e-tickets (boletos electrónicos)]. La persona es un participante importante en la secuencia de emisión de boletos, pero no se representará mediante el código. Un quiosco también es un participante importante y, hasta cierto punto, se representará mediante el código. De este modo, podemos referirnos a un actor llamado “Autoridad para emisión de boletos” y dar a entender que puede ser tanto la persona como el quiosco.

En algunas herramientas de modelado se usa el actor con figura de palillos con una línea de vida sujeta y, en otras, se usa un cuadro con una figura de palillos o el estereotipo «actor». Más importante que la notación precisa es recordar que un actor puede realizarse o no como código y que una línea de vida puede ser un actor.



Figura 4-1 Una línea de vida de un objeto representa un ejemplo de una clase y una línea colocada de manera conveniente para permitir la conexión de objetos por medio de mensajes.

Una línea de vida también puede representar una clase actualizada. Lo que es importante saber es que una línea de vida es, en general, un *nombre* que se puede codificar o no como una clase, pero que definitivamente, es algo que puede interactuar con el sistema de usted, y que una línea de vida es también sólo un rectángulo con una línea vertical que desciende de él.

Activación de una línea de vida

Los objetos tienen una duración. Por ejemplo, en lenguaje determinístico, como C++, un objeto dura hasta que se llama al destructor. En un lenguaje no determinístico como C# (pronunciado “C sharp”), un objeto dura hasta que se recoge la basura. Esto significa que el programador en realidad no sabe cuándo se va el objeto. Sin embargo, los modeladores no están por completo restringidos por el lenguaje de implementación.

Desde nuestra perspectiva, sólo nos preocupamos cuando empezamos a usar un objeto y cuando terminamos de usarlo, a menos que éste represente un recurso finito. En ambos casos, para los fines prácticos, el símbolo de activación representa la amplitud de la duración de un objeto. También es importante saber que un objeto se puede representar como creado y destruido con el uso de una sola línea de vida.

El símbolo de activación es un rectángulo vertical que reemplaza la línea de vida en el transcurso de la duración de la existencia de ese caso (figura 4-2), teniendo presente que un objeto se puede crear y destruir muchas veces y que se usa una línea de vida para representar todos los casos de esa clase en una secuencia. (Un poco más adelante, hablaré

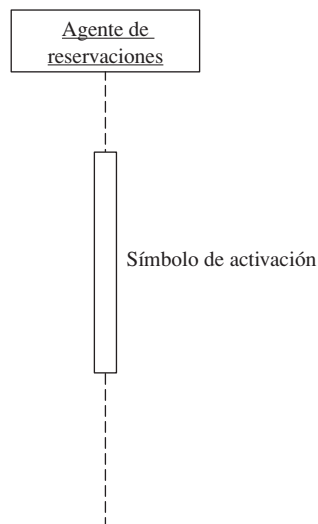


Figura 4-2 Una línea de vida con un símbolo de activación anotado.

acerca de la destrucción determinística.) Si queremos expresar mensajes anidados o recursivos, entonces podemos apilar horizontalmente los símbolos de activación.

Envío de mensajes

Los *mensajes* son líneas dirigidas que conectan líneas de vida. La línea se inicia en una línea de vida, y la flecha apunta hacia aquella línea de vida que contenga el mensaje invocado. El mensaje puede empezar y finalizar en la misma línea de vida; a esto se le conoce como *llamada anidada*. Un triángulo relleno representa un mensaje síncrono; un triángulo de palillos representa un mensaje asíncrono, y se usa una línea punteada para los mensajes de retorno. Incluidos como mensajes posibles, se encuentran los mensajes hallados y los perdidos. Un mensaje hallado tiene un receptor conocido, pero el emisor no se conoce; uno perdido tiene un emisor conocido, pero no receptor especificado. En la figura 4-3, se muestra cada tipo de mensaje rotulado con claridad.

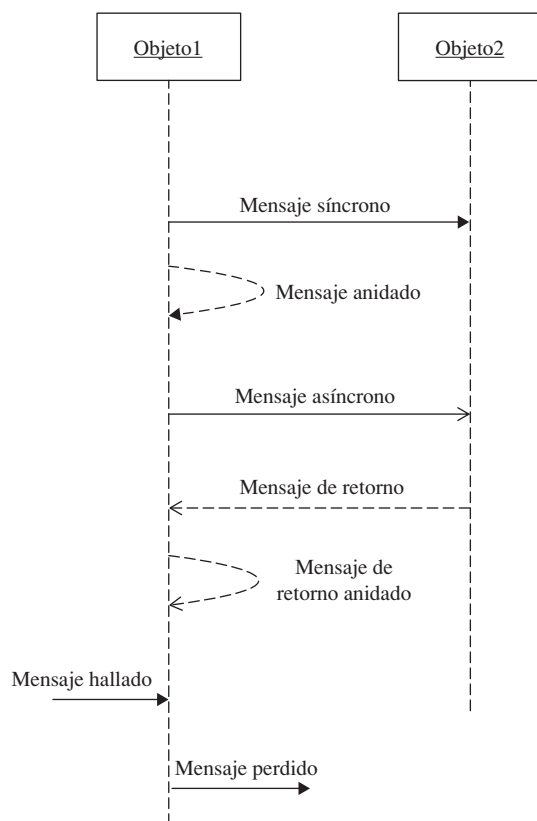


Figura 4-3 Símbolos de llamadas de los métodos síncrono y asíncrono.

También podemos especificar deconstrucción determinística de objetos agregando un círculo con una X en el origen del mensaje. Algunos lenguajes, como Visual Basic.NET y Java, no permiten el borrado determinístico de objetos, pero un lenguaje como C++ lo requiere. (Es posible que usted rara vez encuentre un mensaje de borrado, a menos que sea crítico que les recuerde a los desarrolladores que liberen recursos finitos.)

Suponga que en Motown-jobs.com queremos usar un esquema específico de autenticación y autorización. Podríamos crear una secuencia que describa cómo queremos implementar el caso de uso “Entrar”. Observe la secuencia en la figura 4-4 y vea si puede seguirla de uno a otro lado. Después de la figura está una descripción de la secuencia.

En el objeto usuario se usa el estereotipo actor. (Podría usar también un símbolo de actor.) El usuario no se realizará como código, pero participa en la secuencia. Empezando desde arriba a la izquierda y realizando nuestro camino hacia abajo a la derecha, fijamos el nombre de usuario y la contraseña, y a continuación enviamos el mensaje “Entrar”. (Esto se interpreta como la forma para “Entrar”, teniendo un método llamado “Entrar”).

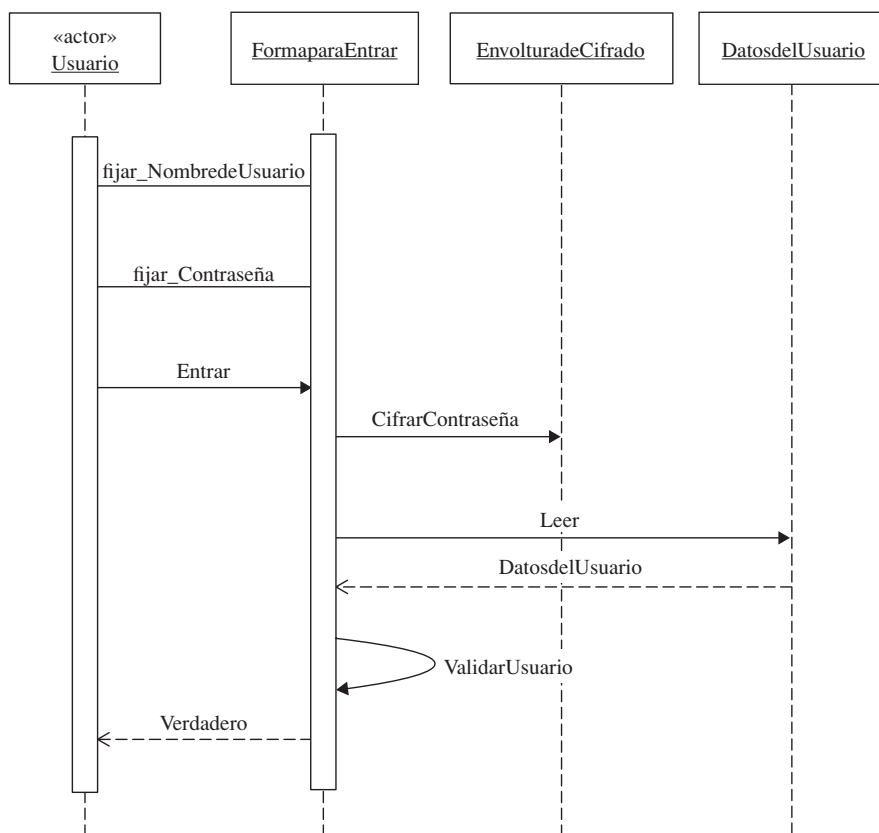


Figura 4-4 Diagrama de secuencia para autenticar un usuario.

Enseguida, la contraseña proporcionada por el usuario se cifra y se compara con la contraseña cifrada almacenada como parte de los DatosdelUsuario. Si ValidarUsuario tiene éxito, entonces retornamos un mensaje Booleano Verdadero.

El diagrama de secuencia es bueno para mostrarnos cómo se orquestan los objetos y se usan los actores de uno a otro lado de un caso de uso, pero no son buenos para mostrarnos cómo se implementa este comportamiento. Por ejemplo, pudimos usar el cifrado con Secure Hash Algorithm 1 (SHA1, Algoritmo Seguro de Verificación) con ingenio y almacenar los datos del usuario con una contraseña cifrada, pero la secuencia no aclara esto. (Para obtener una resolución en cuanto a cómo implementar una secuencia, consulte la sección titulada “Comprensión de lo que nos dicen las secuencias”.)

Adición de restricciones y notas

Puede agregar notas y restricciones con el fin de ayudar a quitar ambigüedad al significado de aspectos particulares de sus diagramas de secuencia. El UML describe la manera en que se agregan estos elementos, pero en la práctica, varían un poco, dependiendo de la herramienta que use. Por ejemplo, podríamos agregar una nota al diagrama de la figura 4-4 que indique que estamos usando SHA1 y un valor de ingenio, y almacenando los datos de la contraseña sólo en una forma cifrada (figura 4-5).

Las restricciones se pueden agregar como texto llano, pseudocódigo, código real u Object Constraint Language (OCL, lenguaje para restricciones de objetos). Las restricciones en código real o en OCL pueden ayudar a las herramientas UML de generación de código a generar líneas del mismo. En algunos procesos pesados de modelado, la habilidad para generar código puede ser una necesidad, pero hasta la fecha, parece más difícil crear modelos UML que generan código granular que escribir el propio código. Usted deberá decidir por sí mismo si necesita modelos moderadamente detallados o muy detallados.

SUGERENCIA *Los modelos con los que se generan aplicaciones completas no son realistas y resultan imprácticos. Evite caer en la trampa de tratar de crear modelos perfectos con detalle suficiente para escribir una aplicación.*

Uso de marcos de interacción

Los *marcos de interacción* (o *fragmentos combinados*) son nuevos en el UML versión 2.0. Estos marcos son regiones rectangulares que se usan para organizar los diagramas de interacción (diagramas de secuencia y de tiempos). Los marcos de interacción pueden rodear un diagrama completo de interacción o sólo parte del mismo. Cada marco de interacción se etiqueta con una palabra específica (o una abreviatura de esa palabra) y cada tipo de marco de interacción transmite alguna información específica. En la tabla 4-1, se definen los tipos actuales de marcos de interacción.

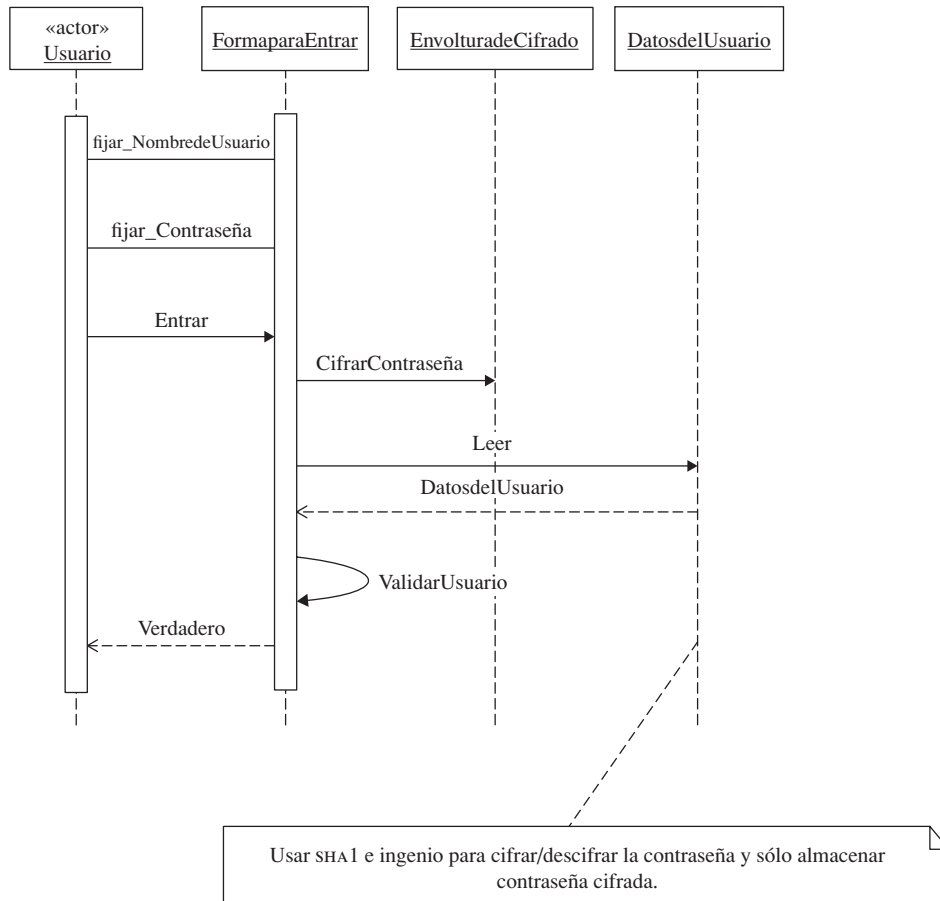


Figura 4-5 Uso de notas para agregar detalles a sus diagramas de secuencia.

Alt	Fragmentos alternativos (es decir, lógica condicional); sólo condiciones guardianes que evalúan para que se ejecute lo verdadero.
Bucle	El guardián indica cuántas veces se ejecutará esta parte.
Neg	Una interacción inválida.
Opt	Equivalente a un alt con una condición (es decir, una condición sin sentencia de otro modo).
Par	Los fragmentos se ejecutan en paralelo; piense en encaminamiento múltiple.
Ref	Hacer referencia a una interacción definida en otro diagrama.
Región	Región crítica; piense en no reentrante o sólo un camino a la vez.
Rod	Usado para rodear un diagrama completo de secuencia, si se desea.

Tabla 4-1 Tipos de marcos de interacción.

El UML nació para ampliarse. Si piensa en otra clase de marco, entonces úselo, siempre que lo defina. Desviarse del UML estandarizado es algo que se hace con mucha frecuencia, lo cual es coherente con la manera en que evolucionan todos los lenguajes. Existen ejemplos de jerga que se adoptan en los lenguajes hablados constantemente.

Pasemos unos minutos observando los marcos de interacción. La clave para usar los marcos de interacción es elegir el tipo de marco que necesita, especificar las condiciones guardianes que determinan cómo se ejecuta la interacción que está en el marco y agregar el número correcto de fragmentos (o divisiones del marco). Empecemos con el marco de bucle, el cual, básicamente, es una construcción del tipo para... a continuación, para... cada o mientras, como podría aparecer en un modelo UML (figura 4-6).

NOTA Antes, en este libro, dije que usaría Visio para demostrar que no necesita gastar miles de dólares para crear modelos UML que se puedan usar. En la figura 4-6, se demuestra que podemos crear nuevos elementos para UML versión 2.0 —por ejemplo, marco de interacción de bucle—, aun cuando Visio no los permita en forma directa. (La interacción de la figura se creó con herramientas sencillas de Visio para trazar líneas.) En el caso de los marcos de interacción, no he visto alguna herramienta actual del UML que soporte esta construcción. La versión actual de Rational para XDE y Visio no incluye marcos de interacción. Usted puede verificar las ofertas de Togethersoft y de Poseidon para UML.

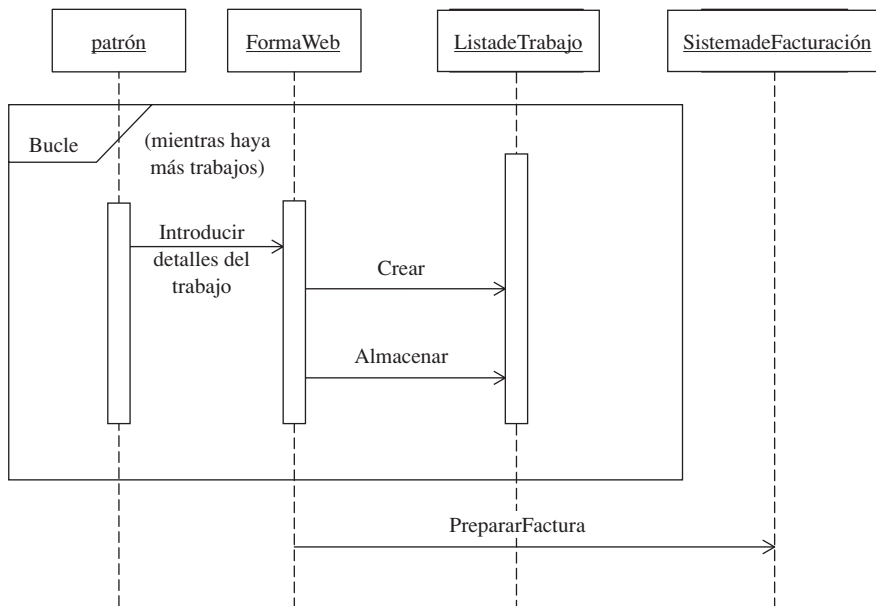


Figura 4-6 Marco de interacción en el que se muestra el marco de bucle; estamos formando un bucle mediante la creación de múltiples listas de trabajos.

Leemos el diagrama de secuencia de la misma manera que antes, excepto que todos los mensajes en el marco de bucle son parte del comportamiento repetitivo que describe esta secuencia. (Una notación de estilo más antiguo era usar un asterisco como condición guardián. En la figura 4-7, se muestra el mismo modelo usando el símbolo de multiplicidad [un asterisco]).

La clave para tener éxito al modelar es recordar que esto se hace en un mundo con restricciones reales: presupuesto para las herramientas, tiempo disponible, la compatibilidad de la herramienta, la definición actual del UML, etc. No se atasque en las leyes del lenguaje. Si su herramienta no soporta una construcción particular, invente. En la práctica, yo no pasaría tiempo para trazar en forma manual un marco de interacción, si mi herramienta no lo soporta; usaría la condición guardián asterisco.

En la figura 4-8, se muestra otro marco común de interacción, el *marco alternativo*. Suponga que ofrecemos gratificaciones para los clientes que publican con frecuencia un cierto número de trabajos. Puede ser que queramos pasar estos clientes a un sistema diferente de facturación, quizás ofreciendo un descuento especial por volumen.

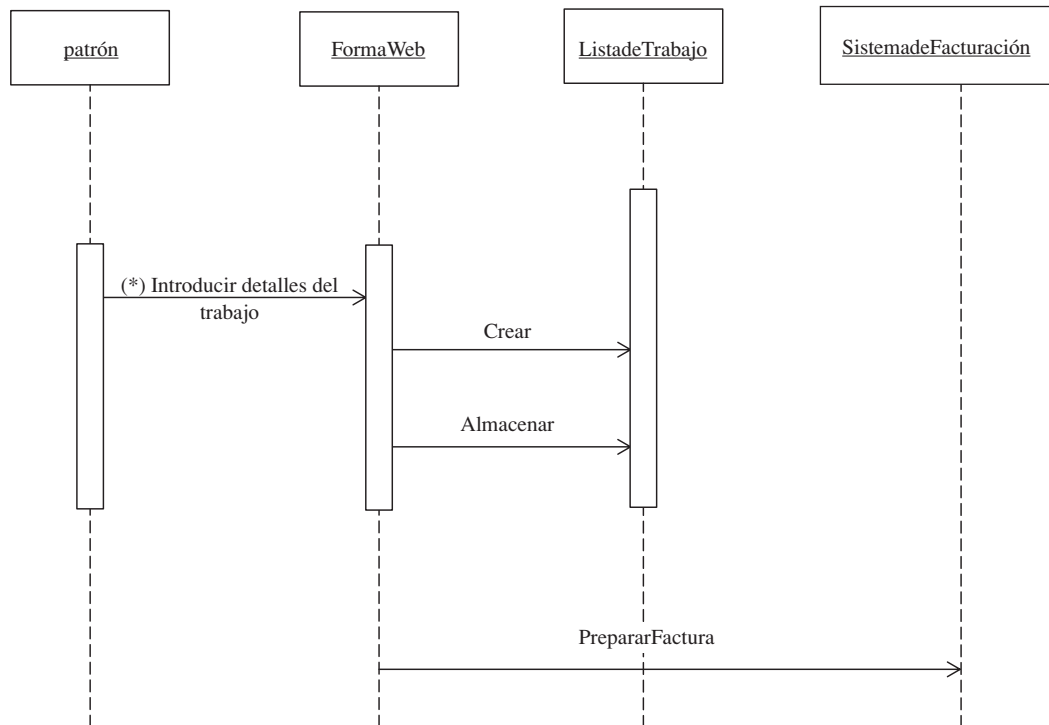


Figura 4-7 La condición guardián —[*]— por el nombre del mensaje “Introducir detalles del trabajo” indica multiplicidad o repetición, en un estilo antiguo ideado para indicar un bucle.

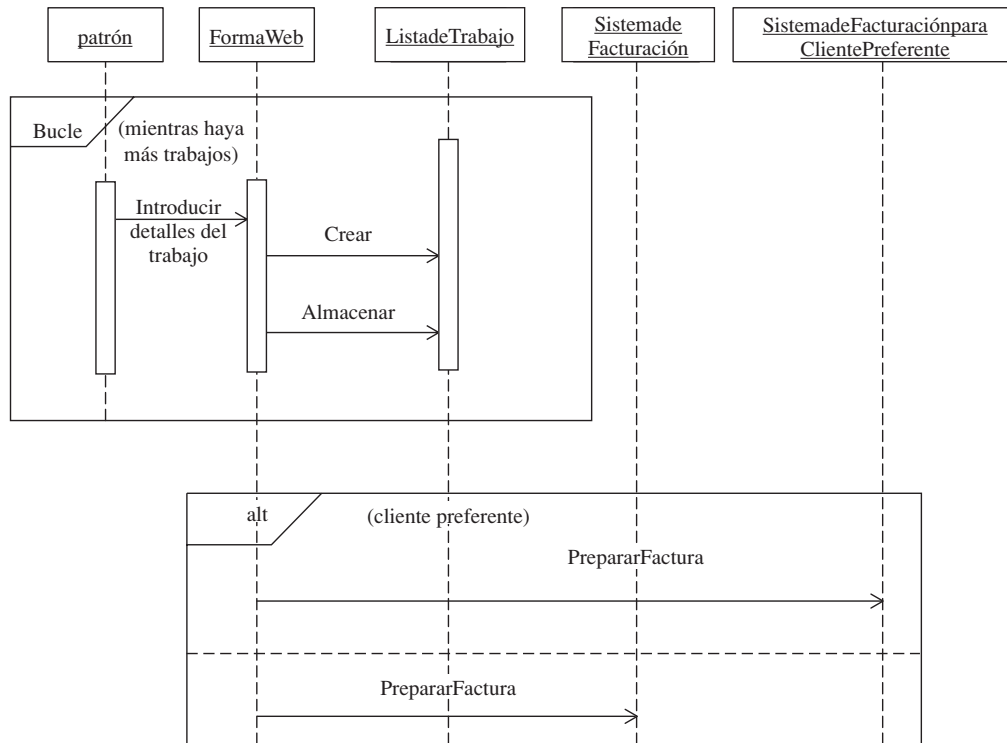


Figura 4-8 Ejemplo de un marco alternativo de interacción.

Comprensión de lo que nos dicen las secuencias

Los diagramas de secuencia de estilo más antiguo tenían una naturaleza singular, pero con los marcos de interacción podemos transmitir de manera más conveniente las alternativas de comportamiento, comportamiento paralelo y bucles y, evidentemente, secuencias relacionadas con referencias. Implícito en el ordenamiento de arriba a la izquierda hacia abajo a la derecha de los diagramas de secuencia, se encuentra un ordenamiento en el tiempo que muestra cómo un solo caso de uso queda soportado por múltiples objetos.

Las secuencias no necesitan ser complejas para ser útiles; lo más importante son los objetos de uno a otro lado de la horizontal y la línea de vida de cada uno de ellos, así como el orden y el nombre de los mensajes enviados entre los mismos. En realidad, usted tiene la opción de escalonar las líneas de vida, creando un efecto de dentado; en ocasiones verá este estilo de secuencia. Escalonado o alineado horizontalmente, el efecto es el mismo.

NOTA *Un modelo completo es subjetivo. En el Rational Unified Process (RUP, Proceso racional unificado), es preferible contar con más detalle. En el empleo de la metodología Agile, se le alienta a crear modelos que sean apenas suficientemente buenos. Al final —quizás dentro de 50 años— se requerirá que los modelos de software sean tan detallados y tan rigurosos como los diagramas de alambrados electrónicos, pero ese día no está aquí todavía. Yo prefiero algo más detallado que los modelos apenas suficientemente buenos prescritos por la metodología Agile, pero nunca tanto como para generar líneas de código.*

Use diagramas de secuencia para mostrar la manera en que varios objetos sustentan un caso de uso. Aun cuando las secuencias sean buenas como para mostrar cómo se presentan los objetos en un caso de uso, no lo son en la descripción del comportamiento específico. Si quiere modelar con más detalle del que soporta una secuencia, entonces considere usar un diagrama de actividad o el propio código; modelar el código en el nivel de sentencia generalmente se capta de modo más eficaz si se escribe el código. Si quiere tener una vista ortogonal —muchos casos de uso, un solo objeto—, entonces necesita un esquema de estado (vea el capítulo 8).

Descubrimiento de objetos y mensajes

Los casos de uso deben contener escenarios de éxito y de falla. En el UML versión 2.0, puede usar la construcción de alternación para mostrar lo que sucede cuando las cosas van como se planeó y qué hacer cuando las cosas van desorganizadas.

Los diagramas de secuencia también son buenos para ayudarle a descubrir las clases y los métodos. Las clases se pueden identificar con facilidad como un nombre para el ejemplo de sus objetos, y los métodos son los mensajes que se invocan en un objeto. Puede no ser evidente de inmediato cuáles son los parámetros para estos métodos, pero las clases y los métodos son un buen principio.

Debido a la propia naturaleza de las secuencias, también pueden ser buenos para ayudarle a identificar las brechas. Por ejemplo, suponga que descubre que una secuencia tiene una gran cantidad de notas para explicar lo que está sucediendo. Esto puede indicar que allí necesitan estar algunos objetos y mensajes bien nombrados que definan el comportamiento anotado. (En general, encuentro que las clases y los métodos bien nombrados en el código son preferibles a los comentarios que intentan aclarar los métodos largos y los objetos bien nombrados, y los mensajes en los modelos son preferibles a una gran cantidad de notas.) Permita que la secuencia se autoexplique hasta el punto en que sea posible. Considere la figura 4-9, en la cual se muestra un diseño posible para el comportamiento de búsqueda para Motown-jobs.com.

En la figura, tenemos un solicitante de trabajo, una página de búsqueda y algo llamado *motor de búsqueda*. Este diseño no nos habla de la forma de los criterios de búsqueda

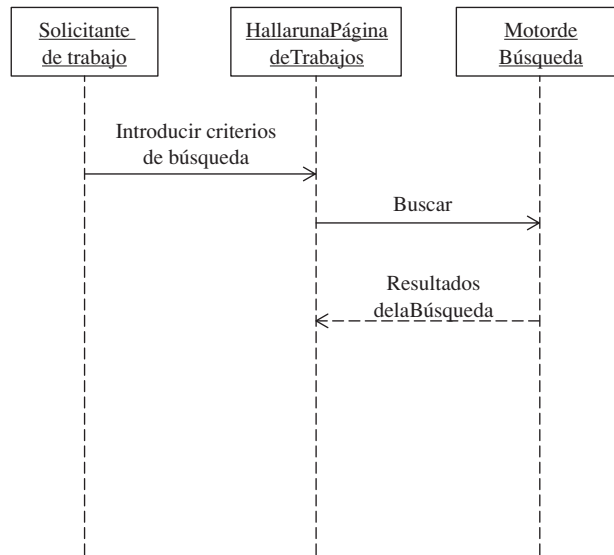


Figura 4-9 Un mal diseño para la búsqueda de los trabajos en lista.

o si los validamos o no. Nada sabemos acerca del motor de búsqueda —qué hace y de dónde recupera los datos— y no tenemos indicio acerca de la forma de los resultados. Esta secuencia necesitaría varias notas y una gran cantidad de soporte verbal. Podemos hacerlo mejor (figura 4-10).

En la secuencia revisada de búsqueda, mostramos que estamos usando un objeto parámetro —“CriteriosdeBúsqueda”— para almacenar, validar y pasar la información de búsqueda que hace entrar al usuario; también estamos describiendo que el motor de búsqueda lee las listas de trabajo desde un objeto base de datos —en este punto, el objeto base de datos sencillamente podría representar una capa de acceso a los datos— y este objeto pone la información leída en una colección tipo de objetos “ListadeTrabajo”. La nueva secuencia es algo que en realidad podemos implementar con muy poca ambigüedad.

Otra característica implícita de la nueva secuencia de la figura 4-10 es que los demás ahora entenderán con claridad lo que pretendemos al usar objetos personalizados para la “ListadeTrabajo”. Antes de proceder con la implementación, podríamos tener una discusión acerca del diseño. Además, debido a que las piezas están delineadas con mayor claridad, podríamos dividir el trabajo entre los especialistas de uno a otro lado del equipo de implementación.

NOTA *La especialización del papel es al menos tan vieja como Wealth of Nations de Adam Smith o las líneas de montaje de Henry Ford, pero, en realidad, apenas se está captando en la industria del software. En nuestra industria relativamente joven, todavía parece que se prefieren las personas de conocimientos variados y los sufrimientos como resultado.*

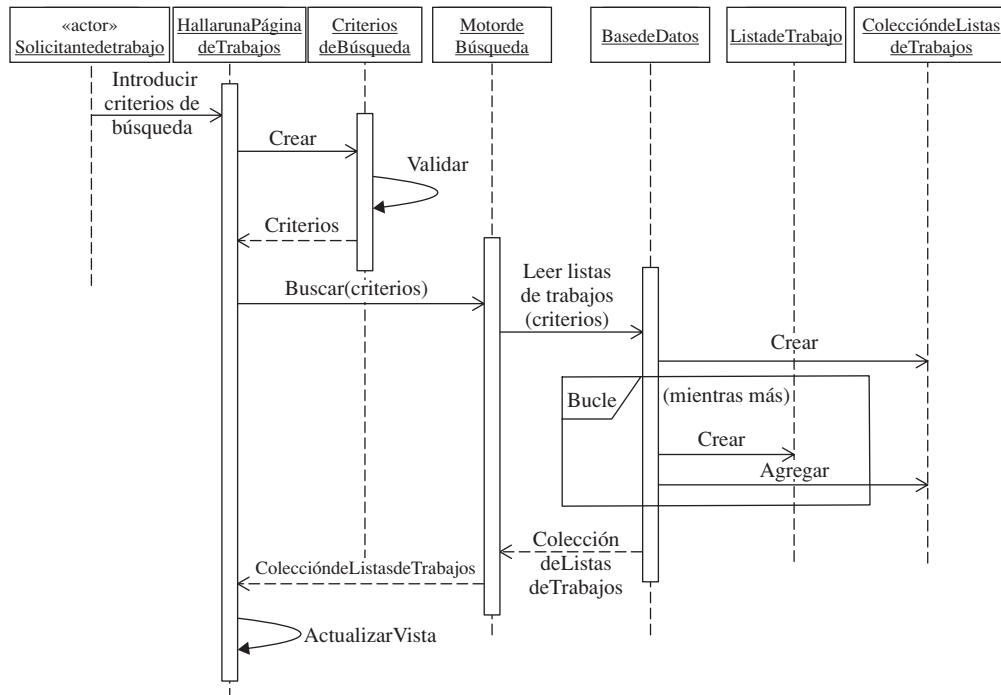


Figura 4-10 Comportamiento de búsqueda de Motown-jobs.com con un diagrama de secuencia detallado.

Elementos de los diagramas de colaboración (o comunicación)

Un *diagrama de colaboración* —reapodado *diagrama de comunicación* en el UML versión 2.0— transmite la misma información que un diagrama de secuencia. En donde el ordenamiento en el tiempo es implícito en la disposición lineal de un diagrama de secuencia, indicamos explícitamente el orden en el tiempo numerando los mensajes en los diagramas de colaboración geoméricamente organizados.

Los símbolos clave en los diagramas de colaboración son el rectángulo, llamado *papel clasificador*, y una línea que indica el mensaje, una vez más llamada *conector*. El papel clasificador representa los objetos. Los conectores representan objetos conectados y una flecha nombrada indica el mensaje así como el emisor y el receptor. En la figura 4-11, se muestra la secuencia de la figura 4-10 convertida a un diagrama de colaboración.

Como puede ver, la colaboración tiene los mismos elementos pero pocos detalles. La naturaleza compacta y la menor cantidad de elementos hacen que las colaboraciones

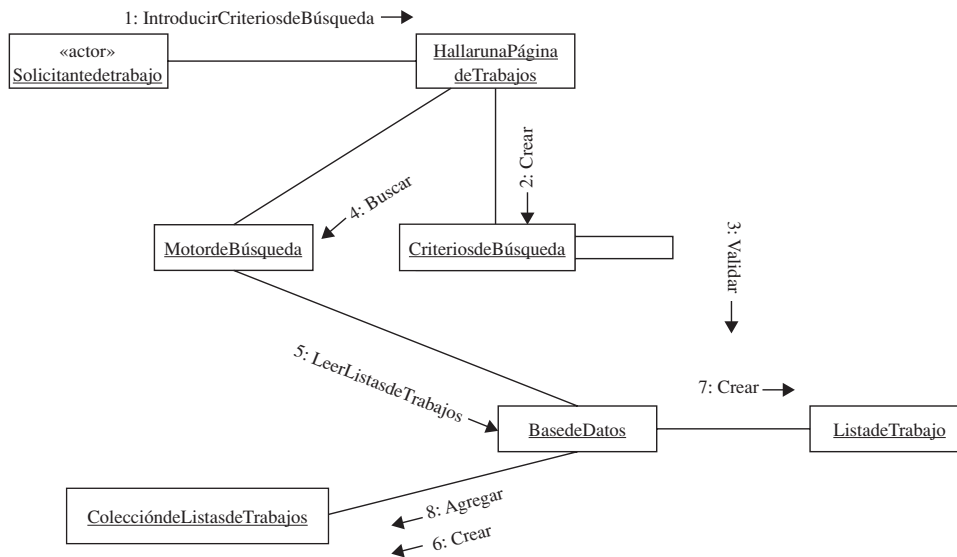


Figura 4-11 Búsqueda de listas de trabajos representada en un diagrama de colaboración.

sean convenientes al garabatear los diseños. Para leer el diagrama, parta del mensaje 1 y siga los mensajes por número. No se pretende que en los diagramas de colaboración se usen marcos de interacción y, como resultado, no transmiten tanta información como el diagrama de secuencia.

Observe el esquema de numeración de la figura 4-11. Siempre he usado un esquema sencillo de numeración, tal como el descrito en esta figura, pero el UML versión 2.0 válido requiere un esquema de numeración anidada. Un esquema sencillo de números es 1, 2, 3, 4, etc. El esquema de números anidados del UML versión 2.0 es 1.1, 1.2, 2.1, 2.2, etc. El esquema de numeración anidada está diseñado para mostrar llamadas de mensajes anidados, pero se puede salir de control con rapidez. Si quiere usar el sistema de numeración del UML versión 2.0, entonces los mensajes se renumerarían como sigue: 1 sigue siendo 1, 2 se convierte en 1.1, 3 se convierte en 1.1.1, 4 se convierte en 1.2, 5 se convierte en 2, 6 se convierte en 2.1, 7 se convierte en 2.2 y 8 se convierte en 2.3.

SUGERENCIA Considere usar los diagramas de colaboración cuando trabaje sobre un pizarrón blanco o sobre servilletas, o donde sea que garabatee sus inspirados diseños. La naturaleza compacta y el uso de pocos adornos de los diagramas de colaboración los hacen más manejables cuando se diseñan en forma manual.

Los diagramas de colaboración tienen otros elementos comunes como las notas, las restricciones y los estereotipos; estos elementos se usan de la misma manera que en los diagramas de secuencia.

Igualación del diseño con el código

Los diagramas de interacción le proporcionan información suficiente como para empezar a codificar. Los objetos son casos de clases, de tal manera que necesita definir una clase para cada objeto. En general, los mensajes se igualan con los métodos, y el método se coloca en la clase del receptor (no del llamador).

En general, he encontrado que con las secuencias tengo la mayor parte de la información que necesito para empezar a escribir el código. La manera en que se implementa el código se basa en un par de factores: 1) la experiencia que usted tenga y 2) el lenguaje de implementación. Por ejemplo, “ListadeTrabajo” y “ColeccióndeListasdeTrabajos” representan una clase y una colección de objetos de esa clase. Si tuviera que implementar esto en C#, entonces “ColeccióndeListasdeTrabajos” se podría heredar de “System.Collections.CollectionBase”, y esa decisión impulsa su implementación (vea la lista).

```
public class ListadeTrabajo
{
}

public class ColeccióndeListasdeTrabajos: System.Collections.
CollectionBase
{
    public ListadeTrabajo this [int index]
    {
        get{return (ListadeTrabajo)List[index];}
        set{List[index] = value;}
    }

    public int Add (ListadeTrabajo value)
    {
        return List.Add(value);
    }
}
```

Advierta que en esta lista heredo de una colección base específica, defino una propiedad llamada *esto* y le agrego el método mostrado en la secuencia. Es importante notar que la secuencia diseñada no indicó la propiedad *esto* o la clase padre; los diagramas de secuencia no lo harán. En este caso el lenguaje de implementación —C# y la .NET Framework de Microsoft— impulsó esta parte de la decisión. Advierta también que la lista de trabajo nada nos dice; es una clase vacía. Bien, la “ListadeTrabajo” de la secuencia tampoco nos dice nada. Los diagramas de secuencia no son buenos para la especificación de detalles del código; sin embargo, arrancamos interfaces. En este punto, depende de la experiencia de sus desarrolladores cuánto código pueden escribir. Los desarrolladores menos experimentados necesitarán más detalles, y los más experimentados necesitarán menos. Yo tiendo a modelar el detalle que es suficiente para mi audiencia; los desarrolladores realizan la implementación.

Para empezar a especificar más detalles, como las propiedades, los métodos de soporte y las relaciones de herencia, podemos usar diagramas de clase. En el capítulo 5 profundizaremos más acerca de los diagramas de clases.

Tenga presente que existe una gran cantidad de conocimiento implícito en esta etapa. En primer lugar, debe usted saber que es posible que su diseño cambie; en segundo, cosas tales como las colecciones de salida se basan en patrones y, como se demuestra en la lista del código, el lenguaje y el marco de referencia (framework) impulsan la implementación; en tercero, existen muchos patrones de diseño comunes y populares (vea Erich Gamma *et al.*, *Design Patterns*, Reading, MA: Addison Wesley, 1995) y no siempre es necesario hacer mucho más que expresar que se usa un patrón; no se requiere en absoluto que usted cree modelos para patrones públicos bien conocidos; y lo último pero no lo menos importante, existe un tema conocido como refactorización. La refactorización es un medio metódico de simplificación del código; se deriva de una tesis doctoral de William Opdike y un libro muy publicitado, escrito por Martin Fowler (vea *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison Wesley, 1999). Cuando se emplea la refactorización, puede significar en la práctica que una decisión respecto a un diseño se puede mejorar en el transcurso de la implementación. Si la refactorización es mejor que el diseño, entonces siga adelante y modifique el código, y sencillamente actualice el modelo para reflejar el cambio.

NOTA En las figuras 4-9 y 4-10, demostramos una refactorización en el diseño cuando introducimos el objeto “CriteriosdeBúsqueda”. Esta refactorización se nombra “Introducir objeto parámetro”, lo cual sencillamente reemplaza una larga lista de parámetros con un solo caso de una clase de parámetros que contiene esos valores. También nos escurrimos en un patrón de diseño, “Iterador”. La colección tipo que se implementó como una respuesta para la colección tipo de los objetos “ListadeTrabajo” de la figura 4-10 se hereda de la *CollectionBase* de .NET, la cual, a su vez, implementa un patrón *IEnumerable* (una implementación del patrón iterador). Los diseños y las implementaciones buenos se basan en patrones y refactorizaciones. Los buenos modelos de diseños se basan en un simple, exacto y directo uso del UML y en incorporar patrones de diseño y refactorizaciones.

Examen

1. Un diagrama de secuencia es un ejemplo de
 - a. un diagrama de colaboración.
 - b. un diagrama de interacción.
 - c. un diagrama de clases.
 - d. un diagrama de casos de uso.

2. Los diagramas de secuencia describen todos los objetos que soporta un solo caso de uso.
 - a. Verdadero
 - b. Falso
3. Los diagramas de secuencia son buenos para mostrar cómo implementar líneas de código.
 - a. Verdadero
 - b. Falso
4. Una diagrama de colaboración y uno de comunicación difieren
 - a. porque los diagramas de colaboración muestran aquellos objetos que interactúan y los de comunicación muestran cómo se comunican los objetos.
 - b. no en lo absoluto; los diagramas de colaboración fueron sencillamente renombrados en el UML versión 2.0.
 - c. porque los diagramas de colaboración son geométricos y los de comunicación son lineales.
 - d. Tanto a como c.
5. Los diagramas de secuencia pueden modelar comportamiento asíncrono y de encajamiento múltiple.
 - a. Verdadero
 - b. Falso
6. Los marcos de interacción usan una condición guardián para controlar cuándo y cuál fragmento del cuadro ejecutar.
 - a. Verdadero
 - b. Falso
7. El marco de interacción alt, llamado *operador de interacción*,
 - a. se usa para mostrar un fragmento inválido.
 - b. modela comportamiento opcional.
 - c. muestra lógica condicional.
 - d. modela comportamiento paralelo.

8. Un buen diseño debe incluir tanto diagramas de secuencia como de colaboración.
 - a. Verdadero
 - b. Falso
9. Se usan símbolos de activación para mostrar
 - a. la duración de un objeto en un diagrama de secuencia.
 - b. la duración de un objeto en un diagrama de comunicación.
 - c. cuando se crea un objeto.
 - d. Ninguno de los anteriores.
10. En el UML versión 2.0 válido se emplea
 - a. un esquema de numeración anidada para mostrar ordenamiento en el tiempo en un diagrama de secuencia.
 - b. un esquema de numeración anidada para mostrar ordenamiento en el tiempo en un diagrama de comunicación.
 - c. un esquema de numeración simple para mostrar ordenamiento en el tiempo en un diagrama de secuencia.
 - d. un esquema de numeración simple para mostrar ordenamiento en el tiempo en un diagrama de colaboración.

Respuestas

1. b
2. a
3. b
4. b
5. a
6. a
7. c
8. b
9. a
10. b

¿Cuáles son las cosas que describen mi problema?

En este capítulo se introducen los *diagramas de clases*, los cuales constituyen la vista más común y más importante del diseño que usted creará; se les llama *estáticos* porque no describen acción; lo que hacen es mostrarle cosas y sus relaciones. Los diagramas de clases se diseñan para mostrar todas las piezas de su solución —cuáles piezas se relacionan con ésta o se usan como partes de totalidades nuevas— y deben transmitir un sentido del sistema que se estructurará en reposo.

Para comunicarse en un nivel técnicamente preciso en el idioma del Unified Modeling Language (UML), es de gran ayuda aprender palabras como *asociación*, *composición*, *agregación*, *generalización* y *realización*, pero para comunicarse en forma suficiente y de manera eficaz, todo lo que debe conocer son palabras sencillas para describir relaciones completas y parte de ellas; es decir, relaciones padres y relaciones hijos, y ser capaz de describir cuántas cosas de un tipo están relacionadas con cuántas de otro. Introduciré los términos técnicos, pero no se atasque intentando memorizarlos. Con la práctica, llegará un momento en que incorporará el idioma UML a su lenguaje cotidiano.

Un mito común es que si encuentra usted todos los nombres y todos los verbos que describen su problema, entonces ha descubierto todas las clases y métodos que necesitará. Esto es incorrecto. La verdad es que los nombres y los verbos que describen su problema de manera suficiente para un usuario son las clases más fáciles de hallar y pueden ayudarle a completar un análisis útil del problema, pero finalizará diseñando y usando muchas más clases que son necesarias para llenar los espacios en blanco.

Este capítulo le mostrará cómo crear diagramas de clases y empezará ayudándole a deducir cómo encontrar la mayoría o todas las clases que necesitará para diseñar una solución. Un concepto importante es que muy pocos diseños requieren que se descubran todos los detalles antes de que resulte la programación. (Unas cuantas agencias gubernamentales y empresas, como la NASA y General Dynamics, pueden tener requisitos rígidos que estipulen la compleción de un diseño, pero en la mayoría de los casos esto conduce a tiempos de producción muy largos y un gasto excesivo.)

En este capítulo, le mostraré cómo usar los elementos de los diagramas de clases, cómo crearlos y cómo captar con anticipación algunas ideas; también le mostraré algunas maneras de descubrir algunas clases y comportamientos menos obvios. El lector aprenderá cómo

- Identificar y usar los elementos de los diagramas de clases
- Crear diagramas de clases simples pero útiles
- Modelar algunas expresiones avanzadas
- Deducir la manera de descubrir clases y comportamientos de soporte menos obvios

Elementos de los diagramas básicos de clase

Tontamente, en la preparatoria no me gustó la clase de Literatura y me dejaron perplejo las clases de gramática. Por fortuna, en la universidad empecé a ver el error de mi modo de pensar. Aun cuando no soy un experto en gramática inglesa, la comprensión de cosas como preposiciones, frases prepositivas, conjunciones, objetos, sujetos, verbos, tiempos verbales, adjetivos, adverbios, artículos, voz activa y voz pasiva, así como palabras posesivas plurales y singulares ayuda mucho al escribir estos pasajes. La razón por la que le digo esto es que, por desgracia, la gramática es un componente del UML porque es un lenguaje, pero la gramática de éste es mucho más fácil que la del inglés. ¿Cuánto más fácil es la del UML? La respuesta es que los dos elementos más importantes en los diagramas de clases, como en otros diagramas, son un rectángulo y una línea. Los rectángulos son clases y las líneas son conectores que muestran la relación entre esas clases.

Los diagramas de clases del UML pueden parecer tan desafiantes como *Hamlet* de Shakespeare o tan fáciles como la prosa de Hemingway en *El Sol también sale*, pero ambos pueden relatar una historia con igual propiedad. Como regla general, enfóquese en las clases y sus relaciones, y use elementos más avanzados, los cuales también expon-

dré, cuando sea necesario. Evite la idea de que los diagramas de clases deben decorarse ampliamente para que sean útiles.

Comprensión de las clases y los objetos

El rectángulo en un diagrama de clases se llama *clasificador*. El clasificador puede decirle el nombre de la clase y el nombre de un ejemplo de esa clase, llamado *objeto*. Al final, las clases incluirán comportamientos y atributos, llamados también, en forma colectiva *características*. Los atributos pueden ser campos, propiedades o ambos. Los comportamientos se considerarán como métodos (figura 5-1).

De modo significativo, en los diagramas de clases se usará el sencillo clasificador representando por la clase “Motocicleta” de la figura 5-1. Los otros tipos son importantes y vale la pena examinarlos. Tomemos un momento para hacerlo.

SUGERENCIA Cuando empiece a captar clases en sus modelos, concíbalos de modo conceptual como una fase del análisis; basta empezar solamente con clases y relaciones. Las características se agregan más adelante.

Uso de clases sencillas

La clase (mostrada en la figura 5-1 como “Motocicleta”) es el elemento más común en el diagrama de clases. Las clases, a final de cuentas, son cosas en su análisis y diseño, y pueden ser cosas específicas del dominio o cosas de apoyo. Considere el ejemplo de los dos párrafos siguientes.

La Groves Motorsports de Mason, Michigan, vende motocicletas, ATV (vehículos para todo terreno), vehículos automotores para nieve y accesorios. Si estuviéramos diseñan-

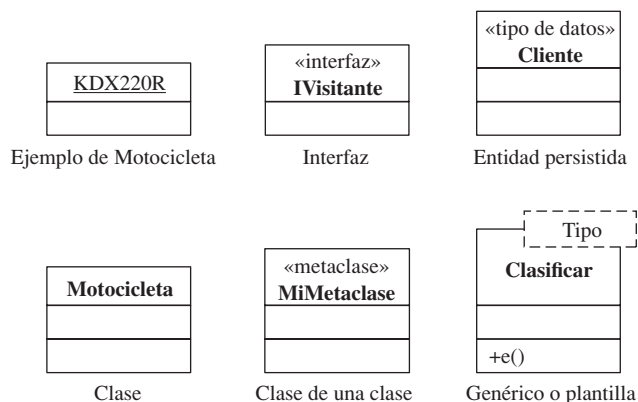


Figura 5-1 Ejemplos de clasificadores en el UML.

do un sistema de inventario para Groves Motorsports, entonces el personal de ventas, los compradores y los mecánicos podrían platicarnos acerca de las motocicletas, los ATV, los vehículos para nieve, las botas, los cascos y los artículos de ventas relacionados. Con base en esta exposición, podríamos deducir con facilidad clases iniciales como “Artículo para Ventas” y “Motocicleta”. Suponga ahora que debemos administrar el inventario usando una base de datos en forma de relación. Ahora necesitamos saber cuál tipo de base de datos y cuáles son las clases que describen cómo interactuamos con los artículos del inventario, es decir, cómo lo leemos y escribimos.

El resultado es que un diagrama de clases puede tener clases que describen los artículos de inventario, pero otros pueden describir elementos como promociones y ventas, financiamiento y administración de artículos que no son para venta, pero que pueden ser parte del inventario de artículos introducidos para mantenimiento. La parte difícil del diseño es encontrar y describir estas relaciones. Una motocicleta todavía es una motocicleta ya sea para venta o si se introduce para servicio, y podemos usar la misma clase “Motocicleta”, pero necesitaremos mostrar tipos diferentes de relaciones basadas en un ejemplo particular de esa clase.

Uso de objetos

Un tipo de diagrama de clases es un *diagrama de objetos*. Los diagramas de objetos muestran ejemplos de clases y sus relaciones. En el UML, un objeto se distingue de una clase subrayando el nombre en el compartimiento superior del rectángulo. Esto se ilustra en la figura 5-1 por mi Kawasaki KDX 220R inspirada por la crisis de mediados de mi vida.

Uso de interfaces

A menudo los programadores tienen problemas con las interfaces (vea “ivisitante” en la figura 5-1). Las *interfaces* son equivalentes a clases abstractas puras. Al decir que una interfaz es puramente abstracta, estoy afirmando que una interfaz no tendrá código ejecutable. Las interfaces constituyen un elemento crítico en los diagramas de clases y el software; tomemos un momento para entender por qué.

Cuando uso *herencia*, quiero dar a entender que una cosa también puede concebirse como otro tipo de cosa. Por ejemplo, tanto una motocicleta como un ATV son tipos de vehículos recreativos. Esta descripción representa una relación de herencia, y en la cual no se usa una interfaz. Comparativamente, un control remoto envía señales infrarrojas para cambiar los canales, atenuar el volumen, empezar a grabar o abrir y cerrar la puerta de una cochera. Los aparatos que reciben estas señales pueden no estar relacionados. Por ejemplo, tanto una TV como el abridor de la puerta de una cochera tienen una característica de hacia arriba y hacia abajo, y los abridores de puertas de cocheras y las televisiones se venden con controles remotos, pero el abridor de la puerta de una cochera no es un tipo de televisión o viceversa, pero cada uno tiene la capacidad de realizar una operación de

hacia arriba y hacia abajo. Hacia arriba y hacia abajo aumentan o disminuyen el volumen de una televisión, o hacia arriba y hacia abajo suben y bajan la puerta de una cochera. Esta capacidad que soporta hacia arriba y hacia abajo a través de un dispositivo de acción remota es una *interfaz* o *faceta relacionada* de cada uno de los aparatos no relacionados. La forma en que se implementa este comportamiento tampoco está relacionada por completo, pero no necesita estarlo.

Las interfaces se usan cuando las partes de las cosas tienen facetas semánticamente similares —comportamientos de hacia arriba y hacia abajo—, pero no tienen genealogía relacionada.

Por convención, usamos el estereotipo interfaz y colocamos el prefijo “I” a las interfaces, como se muestra en la figura 5-1. Considerando la interfaz “Ivisitante” de la figura 5-1, podríamos decir que los visitantes tienen una característica *tipo*. Las pulgas pueden visitar un perro, y su cuñado puede visitarlo a usted en su casa, pero una pulga es un tipo de visitante de perro y su cuñado, Enrique, es un tipo de visitante familiar. Las pulgas y Enrique no son tipos semejantes de cosas (ni con el juego de palabras acerca de los parásitos se pretende que haya semejanzas).

Uso de tipos de datos

Se suele usar el estereotipo de «tipo de datos» para mostrar datos sencillos como “Entero” (“Integer”). Si estuviera diseñando un lenguaje de programación, entonces sus diagramas de clases podrían mostrar tipos de datos, pero en general, yo modelo estos elementos como atributos de clases y clasificadores de reserva como “Motocicleta” y “ListadeTrabajo”.

Uso de tipos parametrizados o genéricos

Los sinónimos pueden hacer que la vida sea confusa. En el UML, *tipos parametrizados* significa lo mismo que *genéricos* en C# y Java, y *plantillas* en C++. Una clase parametrizada es aquella en la que, en el tiempo de ejecución, se especifica un tipo de datos primarios. Para entender las clases parametrizadas, considere un ejemplo clásico.

¿Qué clasifica un algoritmo clasificador? La respuesta es que este tipo de algoritmo puede clasificar cualquier cosa; números, nombres, inventario, corchetes de impuestos sobre la renta o listas de trabajos pueden todos ser clasificados. Al separar el tipo de datos —número, cadena, “ListadeTrabajo”— del algoritmo, tiene un tipo parametrizado. Las clases parametrizadas se usan para separar la implementación del tipo de datos. En la clase “Clasificar” de la figura 5-1 se muestra que en un tipo parametrizado se usa el rectángulo con un rectángulo pequeño trazado con líneas punteadas en el que se especifica el tipo de parámetro.

Vale la pena hacer notar que usar bien las plantillas se considera una parte avanzada del diseño de software y que existe una cantidad tremenda de software grande sin plantillas.

Uso de metaclases

Una *metaclass* es una clase de una clase. Esto parece haber evolucionado para manejar el problema de obtención de información del tiempo de ejecución acerca de las clases. En la práctica, se puede hacer pasar una metaclass como un objeto. Las metaclasses se soportan de manera directa en lenguajes como Delphi; por ejemplo, dada una clase “ListadeTrabajo”, podríamos definir una metaclass y nombrarla (por convención) “TlistadeTrabajo”, pasando ejemplos de esta última como parámetro. Se podría usar la metaclass “TlistadeTrabajo” para crear ejemplos de “ListadeTrabajo”. En un lenguaje como C#, las metaclasses no se soportan en forma directa. En lugar de ello, en C# se usa un objeto “Type” (“Tipo”) que representa la especie de un ejemplo de una metaclass universal; es decir, toda clase tiene un metaobjeto asociado que conoce todo acerca de las clases de ese tipo. Una vez más, en C#, existe la clase “Type” para soportar el descubrimiento del tiempo de ejecución, dinámico, acerca de las clases.

NOTA Existe otro concepto metadatos, que es semejante a la noción de metaclasses. Sin embargo, metadatos son datos que describen datos y a menudo se usan para transmitir información adicional relativa a datos; por ejemplo, a veces se usan metadatos para describir valores válidos para los datos. Suponga que estuviera usted escribiendo un sistema de contabilidad y que las fechas válidas de las facturas fueran del 1° de enero de 1990 hasta los tiempos que corren. La mayoría de los tipos de fechas soportan fechas muy anteriores a la de 1/1/1990, pero usted podría usar el objeto metadatos de fechas con el fin de indicar que, para sus fines, las fechas válidas empezaron en 1/1/1990, en lugar de la fecha más antigua para el tipo de datos de su lenguaje.

Existen algunas aplicaciones prácticas para las metaclasses. En Delphi, las metaclasses se usan para soportar la creación de un control que se arrastre desde el panel de control (caja de herramientas) hasta una forma en el momento del diseño. En .NET, se usa el objeto “Type” —un tipo de implementación de la metaclass— para soportar dinámicamente la carga, la creación y el uso de objetos. Microsoft llama a esta capacidad “Reflection” (“Reflexión”), pero básicamente es una implementación del idioma de metaclasses. Como consecuencia, cuando los diseñadores de Delphi y Visual Studio estaban diseñando sus respectivas herramientas, puede ser que hayan usado el clasificador de metaclasses en sus modelos UML, suponiendo que usaron estos modelos. Es importante reconocer que precisamente como diferentes herramientas UML soportarán diferentes niveles de compatibilidad de UML, los diversos lenguajes soportarán varias decisiones de diseño de maneras distintas.

Decoración de las clases

El símbolo de clasificador se divide en regiones rectangulares (vea la clase “Motocicleta” en la figura 5-1). El rectángulo de más arriba contiene el nombre de la clase y los estereotipos de la misma. La segunda región rectangular, viniendo de arriba, contiene los atributos

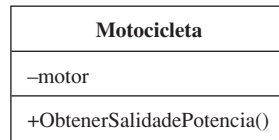


Figura 5-2 Clase “Motocicleta” con un modificador de acceso privado en un atributo motor.

butos (figura 5-2). Como se muestra en la figura 5-2, la clase “Motocicleta” tiene un atributo “motor”. El rectángulo de abajo contiene los comportamientos (o métodos). En la figura 5-2, la clase “Motocicleta” contiene un método llamado “ObtenerSalidadePotencia”.

Cada uno de los atributos y métodos se pueden decorar con modificadores de acceso. (Recuerde que el término *característica* significa en forma genérica “método o atributo”). Las características se pueden decorar con los modificadores de acceso +, – o #. El símbolo de más (+) significa que una característica es pública, o sea, disponible para consumo externo. El símbolo de menos (–) significa que una característica es privada, o sea, para consumo interno, y el símbolo de número (#) significa que una característica no es pública ni privada. Por lo común, el símbolo de número significa que una característica es para consumo interno o consumo por parte de las clases hijos. Este símbolo suele igualarse a un miembro protegido. En general, las herramientas UML harán en forma predeterminada que los métodos sean públicos y los atributos privados.

Uso de atributos

En muchos lenguajes modernos se establece una distinción entre propiedades y campos. Un *campo* representa lo que las clases de usted saben, y una *propiedad* representa una función implícita para leer campos privados y escribir en ellos. No es necesario captar tanto los campos como las propiedades; basta con capturar los campos.

Cuando agrega clases a sus diagramas de clases, agrega los campos y los hace privados. Depende de quienes implementan sus diseños el agregar métodos de propiedad, si están soportados. Si su lenguaje no soporta propiedades, entonces, en el curso de la implementación, use métodos como `get_Field1` (`obtener_Campo1`) y `set_Field1` (`fijar_Campo1`) para cada campo, con el fin de restringir el acceso a los datos de una clase.

SUGERENCIA *Agregar campos privados y depender de un conocimiento implícito de que los campos son accesados a través de métodos, ya sean públicos o privados, es una práctica recomendada pero no impuesta o parte del UML. Este estilo de implementación de diseño simplemente es considerada una buena práctica.*

Declaración de atributos

Los atributos se muestran como una línea de texto; necesitan un modificador de acceso para determinar la visibilidad. Los atributos necesitan incluir un nombre; pueden incluir

un tipo de datos y valor predeterminado, y pueden tener otros modificadores que indiquen si el atributo es sólo de lectura, sólo de escritura, estático o algo más.

En la figura 5-2, el atributo “motor” tiene un modificador de acceso privado y sólo un nombre. Enseguida se dan algunas declaraciones más completas de atributos que contienen ejemplos de los elementos que expusimos:

- Tipo : TipodeMotor = TipodeMotor.DosTiempos
- Tamaño : cadena = “220cc”
- Marca : cadena = “Kawasaki” { sólo lectura }

En esta lista tenemos un atributo privado nombrado “Tipo”, cuyo tipo de datos es “TipodeMotor”, y su valor predeterminado “TipodeMotor.DosTiempos”. Tenemos un atributo nombrado “Tamaño” con un tipo de datos de “cadena” y un valor predeterminado de “220cc”. Y el último atributo es una cadena nombrada “Marca” con un valor predeterminado de “Kawasaki”; el atributo “Marca” es de sólo lectura.

Declaración de atributos con asociación

Los atributos también se pueden describir como una *asociación*. Esto sólo significa que el atributo se modela como una clase con un conector entre la clase contenedora y la clase del atributo. Pueden estar presentes todos los elementos mencionados con anterioridad; sencillamente se disponen de manera diferente.

Considere el atributo “motor” que se muestra en la figura 5-2. Este atributo podría referirse a una asociación a una clase “Motor” (figura 5-3); además, los atributos —“Tipo”, “Tamaño” y “Marca”— se podrían poner en una lista como miembros de la clase “Motor”.

Cuando use un atributo de asociación, deje la declaración del campo fuera de la clase. El enlace de asociación (mostrado como “motor”) en la figura 5-3 desempeña ese papel; no hay necesidad de repetir la declaración en forma directa en la clase contenedora. El conector de asociación se nombra. Este nombre representa el nombre del campo: en la figura 5-3, el nombre es “motor” y la clase es “Motor”. Los atributos de asociación también pueden contener una *multiplicidad*, la cual indica cuántos de cada elemento intervienen en la asociación. En el ejemplo, una motocicleta tiene un motor. Si la relación fuera “Aviones” y “Motores”, entonces podríamos tener un asterisco enseguida de la clase “Motor” con el fin de indicar que los aviones pueden tener más de un motor.

SUGERENCIA En algunas convenciones se usa un prefijo artículo para un nombre de asociación, como “el” (“la”) o “un” (“una”), como en “elMotor” o “unMotor”.

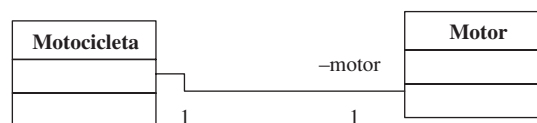


Figura 5-3 Manera de mostrar el atributo “motor” usando una asociación.

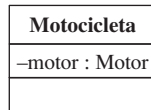


Figura 5-4 Esta figura transmite una información idéntica a la que se muestra en la figura 5-4; es decir, una motocicleta contiene un motor cuyo tipo es “Motor”.

El diagrama de clases de la figura 5-3 transmite una información idéntica a la del diagrama de la figura 5-4. Los diagramas de clase pueden volverse con facilidad demasiado complejos si todos los atributos se modelan como asociaciones. Una buena regla empírica es mostrar tipos simples como declaraciones de campo en la clase contenedora y mostrar tipos compuestos (clases) como atributos de asociación. En la figura 5-5, se muestra cómo podemos detallar la clase “Motor” de modo más completo, usando un atributo de asociación en lugar de sólo un campo “motor”. (En la figura 5-5, se agregan los campos usados para describir un motor mencionado con anterioridad.)

En la figura 5-5, queremos decir que sólo una motocicleta tiene un motor Kawasaki de dos tiempos y 220cc. (Es posible que esto no sea cierto en la vida real, pero eso es lo que transmite el modelo.)

***NOTA** Mencione que el diagrama de la figura 5-5 significa que sólo una motocicleta tiene un motor Kawasaki de dos tiempos y 220cc, pero que esta información puede ser inexacta. Al hacerlo, de manera inadvertida volví a ilustrar uno de los valores de los diagramas de clases: un diagrama de clases es una imagen que significa algo, y los expertos pueden observarla y decirle a usted con rapidez si ha captado algo que se basa en hechos y es útil.*

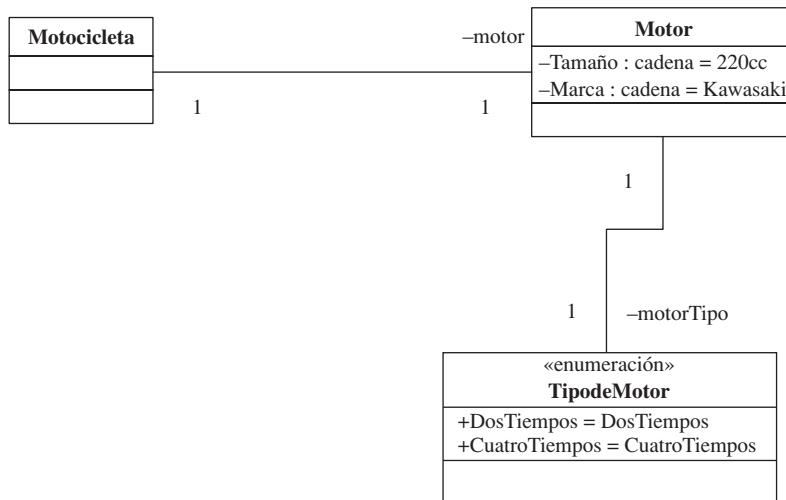


Figura 5-5 Este diagrama de clases contiene más información acerca del motor de la motocicleta al usar un atributo de asociación para el motor y un segundo atributo de asociación para los tipos posibles de motores.

Arreglos de atributos y multiplicidad

Un solo tipo de atributo podría representar más de uno de ese tipo. Esto implica la multiplicidad y, posiblemente, el ordenamiento de los atributos. Puede haber más de uno de algo; por ejemplo, se podrían modelar aviones de múltiples motores como un avión con un arreglo de motores, y los arreglos se pueden ordenar o desordenar. La multiplicidad se indica con la mayor facilidad agregando un conteo a un atributo de asociación, y los atributos ordenados o desordenados se pueden anotar usando las palabras *ordenado* o *desordenado* entre corchetes. En la tabla 5-1, se muestran los conteos posibles de multiplicidad y se proporciona una descripción para cada uno.

Los indicadores de multiplicidad se usan en otros contextos y tienen el mismo significado de conteo cuando se aplican a otros elementos UML junto con asociaciones de atributos.

SUGERENCIA Si los valores superior e inferior son idénticos, entonces use un indicador de multiplicidad de un solo valor, como 1, en lugar de 1..1.

Cuando se habla de multiplicidades, podría escuchar los términos *opcional*, aplicado a multiplicidades con cota inferior a 1, *obligatorio*, si se requiere al menos uno, de *un solo valor*, si sólo se permite uno, y de *valores múltiples*, si se usa un asterisco.

Indicación de unicidad

Los atributos se pueden anotar para indicar unicidad. Por ejemplo, si un campo representa una clave en una tabla de verificación o una clave primaria en una base de datos en forma de relación, entonces puede resultar útil anotar ese atributo con los modificadores {único} o {no único}. Por ejemplo, si quiere indicar que la “IDdeLaListadeTrabajo” es un campo con valor único, entonces lo definimos en la clase como sigue:

–IDdeLaListadeTrabajo : entero {único}

Si quiere indicar que el valor clave de una colección debe ser único, entonces use el modificador {único}. Si las claves se pueden repetir, entonces use {no único}. Rara vez

1	Sólo 1
*	Muchos
0..1	Cero o 1
0..*	Una cota inferior a cero y una superior a infinito; esto es equivalente a *
1..1	Uno y sólo uno; esto es equivalente a 1
1..*	Una cota inferior de por lo menos uno y una superior de infinito
<i>m, n</i>	Indicación de una multiplicidad no contigua, como 3 o 5; ya no es válido en el UML

Tabla 5-1 Indicadores de multiplicidad.

los modeladores tienen tanto tiempo que usan diagramas muy detallados que incluyen {no ordenada} para dar a entender tabla de verificación. En general, los modeladores sencillamente expresan el tipo de datos del atributo, pero vale la pena saber que en el UML se especifica ordenado contra no ordenado, y único contra no único, y no arreglo o tabla de verificación. Los arreglos y las tablas de verificación representan soluciones conocidas de diseño, no aspectos del lenguaje UML.

Modo de agregar operaciones a las clases

Puede ser útil pensar en el modelado como algo que pasa por un ciclo desde una macrovisión de alto nivel hasta, en forma sucesiva, microvisiones de nivel inferior y, por último, al código, la microvisión más detallada. La macrofase se puede concebir como una fase del análisis. En el curso de esta fase, podría bastar captar clases y relaciones conforme usted empieza a entender el espacio del problema. Conforme mejora su comprensión y empieza a captar los detalles de una solución —avanzando de una macrocomprensión hacia una microcomprensión más detallada— empieza a desarrollar el diseño. En esta coyuntura, puede regresar a sus diagramas de clases y empezar a agregar operaciones y atributos. Las operaciones, los comportamientos y los métodos se refieren, todos, a lo mismo. En el UML, por lo general decimos *operación* y al codificar, por lo general decimos *método*.

Las operaciones se muestran en el rectángulo que está más abajo en un clasificador. Las operaciones tienen un modificador de visibilidad como los atributos. Las operaciones incluyen un tipo de datos de retorno; un nombre; una lista de parámetros que incluye nombres, tipos de datos y modificadores, y modificadores adicionales que pueden indicar si una operación es estática, virtual o algo más.

Como mencioné con anterioridad, no es necesario mostrar los métodos de propiedades. También puede ahorrar algo de tiempo al no desarrollar las operaciones no públicas con gran detalle. En general, las operaciones públicas describirán en forma suficiente los comportamientos de la clase, y puede dejar los miembros no públicos a los dispositivos de sus programadores.

Como en realidad yo no tengo una aplicación que represente motocicletas o un inventario de vehículos para una tienda de vehículos motorizados deportivos, cambiemos un poco los ejemplos. En ocasiones, voy a Las Vegas y participo en un pequeño “BlackJack” (figura 5-6). Debido a que me gusta entretenerme tanto como sea posible a cambio de mi dinero, quise practicar “BlackJack” de una manera en que me hiciera un mejor jugador. Por tanto, escribí un juego de “BlackJack” que proporcionaba sugerencias con base en el mejor curso de la acción para ganar una mano. (Esta aplicación está terminada y el código se encuentra en línea en www.softconcepts.com.) En ese ejemplo, hay muchas clases, incluyendo una que representa la mano de un jugador como una lista de cartas. En el clasificador de la figura 5-7, se muestran algunas de las firmas de operaciones usadas para implementar la clase “Mano” (“Hand”).

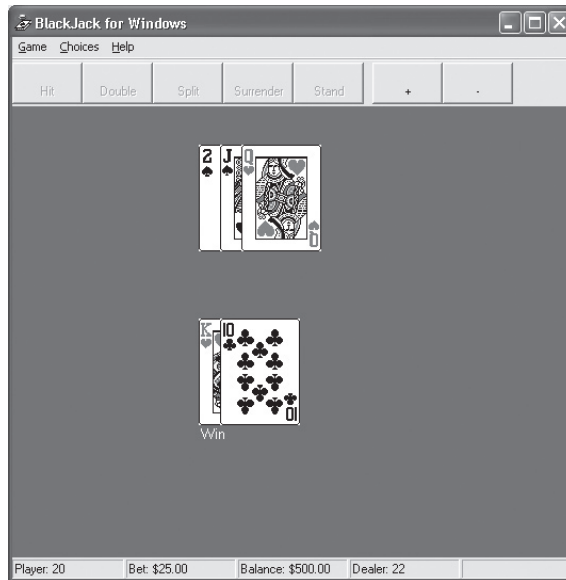


Figura 5-6 El juego “BlackJack for Windows”.

Hand
<pre> +Add(in card : Card) : int +New() : void +GetTextHand() : string +TextPrintHand() : void +IsBlackJack() : bool +IsBust() : bool +BustedAtValue() : int +GetHandValueLow() : int +GetHandValueHigh() : int +GetBestValue() : int #GetHandWidth() : int +GraphicPrintHand(in g : Graphics, in x : int, in y : int, in cardWidth : int, in cardHeight : int, in focused : bool) : void +Stand() : void +Dump() : void </pre>

Figura 5-7 Clasificador que muestra varias de las firmas para la clase “Mano” (“Hand”).

Modelado de relaciones en los diagramas de clases

Los diagramas de clases constan principalmente de clasificadores con atributos y operaciones así como de conectores que describen las relaciones entre las clases. En alrededor del 80% de sus diagramas de clases sólo usará estas características. Sin embargo, aun cuando esto suena sencillo, se pueden usar estos diagramas para describir algunas relaciones muy avanzadas. Por nombre, estas relaciones incluyen generalización, herencia,

realización, composición, agregación, dependencia y asociación. Con mayor refinación, los conectores que describen estas relaciones pueden ser dirigidos o no dirigidos y bidireccionales o no direccionales, y pueden expresar multiplicidad (precisamente como la multiplicidad de los atributos). En esta sección introduciré estos conectores, pero esperaré hasta el capítulo 6 para examinar ejemplos con más detalle.

Modelado de asociaciones

El conector de asociación es una línea continua. Si es dirigida, entonces la línea continua puede tener una flecha de figura de palillos en cualquiera de los dos extremos o en ambos. Por ejemplo, en la sección anterior impliqué que una “Mano” de blackjack está compuesta de objetos “Carta” (“Card”). Podría modelar esta relación agregando una clase “Carta” a la clase “Mano” introducida en la figura 5-7 y conectando los clasificadores “Mano” y “Carta” con un conector de asociación. Vea la figura 5-5 en relación con un ejemplo visual de dos asociaciones, una entre “Motocicleta” y “Motor” y otro entre “Motor” y “TipodeMotor”.

Precisamente como en la figura 5-5, las asociaciones pueden expresar multiplicidad en cualquiera de los dos extremos del conector. En la figura 5-5, se indica que una “Motocicleta” está asociada con un “Motor”, y en la 5-8 se indica que existe por lo menos una mano y que cada una de éstas puede contener muchas cartas.

Si hay una flecha en cualquiera de los dos extremos de una asociación (figura 5-8), entonces se dice que la asociación es *dirigida* o *direcciona*l. El extremo con la flecha es el objetivo o el objeto hacia el que se puede navegar. El extremo sin la flecha se llama *fuentes*. *Navegación* sencillamente significa que la fuente —“Mano” de la figura 5-8— tiene un atributo del tipo del objetivo —“Carta”. Si la asociación fuera bidireccional, entonces “Mano” tendría un atributo “Carta”, y ésta tendría un atributo “Mano”. Si la asociación fuera no dirigida —no hay flechas— entonces se supone una asociación bidireccional.

Modelado de agregación y composición

La agregación y la composición tienen que ver con las relaciones de totalidad y parte. El conector para la agregación es un diamante hueco, una recta y, de manera opcional, una flecha de figura de palillos. El diamante se agrega al clasificador de totalidad y la flecha al de parte. Un conector de composición se parece al de agregación, excepto que el diamante está relleno.



Figura 5-8 “Mano” y “Carta” están asociados de manera unidireccional, lo cual significa que “Mano” tiene un atributo “Carta”.

Imaginarse cómo usar la agregación y la composición se puede decidir de manera muy sencilla. La agregación es azúcar sintáctica y no es diferente de una asociación; usted no la necesita. La composición es agregación, excepto que la clase totalidad es responsable de la creación y de la destrucción de la clase parte, y esta última no puede existir en alguna otra relación al mismo tiempo. Por ejemplo, el motor de una motocicleta no puede estar en una segunda motocicleta al mismo tiempo; eso es composición. Como Fowler dice: en una relación de composición hay una regla de “no compartir”, pero los objetos parte se pueden compartir en las relaciones de asociación y agregación.

Antes de observar la figura 5-9, compare la agregación (o asociación) con la composición pensando en el popular juego de póquer Texas hold'em, en el cual cada jugador tiene dos cartas y, a continuación, se dan cinco cartas. Cada jugador forma la mejor mano posible de cinco cartas usando sus dos cartas y las cinco compartidas. Es decir, la mano de cada jugador es un agregado de cinco de las siete cartas, cinco de las cuales están disponibles para todos los jugadores; o sea que se comparten cinco cartas. Si fuéramos a escribir una versión en software del Texas hold'em usando nuestra abstracción “Mano”, entonces cada uno de los jugadores tendría una referencia hacia las cinco cartas compartidas. En la figura 5-9, se muestra la agregación a la izquierda y la composición a la derecha.

Modelado de la herencia

Es importante tener presente que el UML es un lenguaje distinto, distinto de su lenguaje favorito de programación orientado a objetos y, en general, distinto de los lenguajes de programación orientados a objetos. Por tanto, para ser modelador con UML, necesita ser multilingüe; los modeladores con UML necesitan hablar este lenguaje y, en realidad, ayuda hablar el lenguaje orientado a objetos que se usará para implementar el diseño. En el idioma UML, la herencia es la *generalización*. Esto significa que los programadores pueden decir *herencia* cuando quieren decir *generalización*, y cuando dicen *generalización*, puede ser que quieran decir *herencia*.

NOTA Desafortunadamente, las relaciones de herencia sufren de una plétora de sinónimos. Herencia, generalización y es un(a) *se refieren a lo mismo*. Las palabras padre e hijo también se mencionan como superclase o clase base y subclase. Base, padre y superclase *significan lo mismo*. Hijo y subclase *significan lo mismo*. Los términos que escuche dependen de quién le está hablando a usted. Para empeorar las cosas, a veces estas palabras se usan en forma incorrecta.

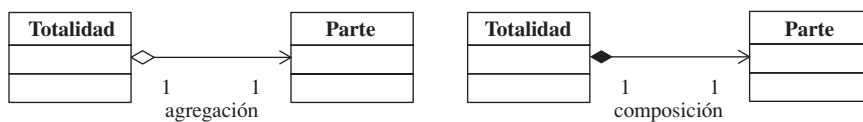


Figura 5-9 La agregación es semánticamente idéntica a la asociación, y la composición significa que la clase compuesta es la única clase que tiene una referencia hacia la clase propietaria.

La *generalización* se refiere a una relación del tipo *es un(a)* o de posibilidad de sustitución y se refleja en un diagrama UML de clases por medio de un conector de línea continua con un triángulo hueco en uno de los extremos. El triángulo apunta hacia el padre y el otro extremo se conecta al hijo.

En una *relación de herencia*, la clase hijo recibe todas las características de la clase padre y, a continuación, se pueden agregar algunas características propias. El polimorfismo funciona porque las clases hijos son factibles de sustituirse por clases padres. La posibilidad de ser *sustituido* significa que si se define una operación o sentencia para usar un argumento de un tipo padre, entonces cualquier tipo hijo se puede sustituir por cualquier tipo padre. Considere un ejemplo de Motown-jobs.com (www.motown-jobs.com). Si se define una clase “Lista” como una clase padre y “Currículo”, “Trabajo” o “Anuncio” se definen como clases hijo para “Lista” (padre), entonces en cualquier parte en la que se defina un argumento “Lista”, se puede sustituir con uno de “Currículo”, “Trabajo” o “Anuncio”. En la figura 5-10, se muestra esta relación.

Cualquier miembro público o protegido de “Lista” se convierte en un miembro de “Trabajo”, “Currículo” y “Anuncio”. De manera implícita, los miembros privados son parte de “Trabajo”, “Currículo” y “Anuncio”, pero estas clases hijos —y cualesquiera clases hijos— no pueden tener acceso a los miembros privados de la clase padre (o clases padres, si se soporta la herencia múltiple).

Modelado de realizaciones

Las *relaciones de realización* se refieren a heredar de interfaces de realización o las propias interfaces. El conector es casi idéntico a uno de generalización, excepto que la línea de conexión es punteada con un triángulo hueco, en lugar de ser continua y con un triángulo del mismo tipo. Cuando una clase realiza una interfaz, o se hereda de ésta, básicamente la clase está aceptando que proporcionará una implementación para las características declaradas por esa interfaz. En la figura 5-11, se muestra la representación visual de una clase “Radio” que realiza la interfaz “IVolumen”. (Tenga presente que el prefijo “i” es sencillamente una convención y no parte del UML.)

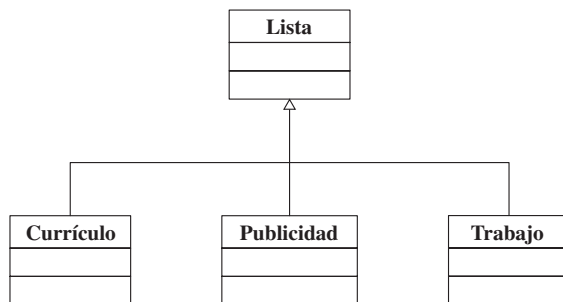


Figura 5-10 Esta figura muestra que “Currículo”, “Trabajo” y “Publicidad” se heredan de “Lista”.

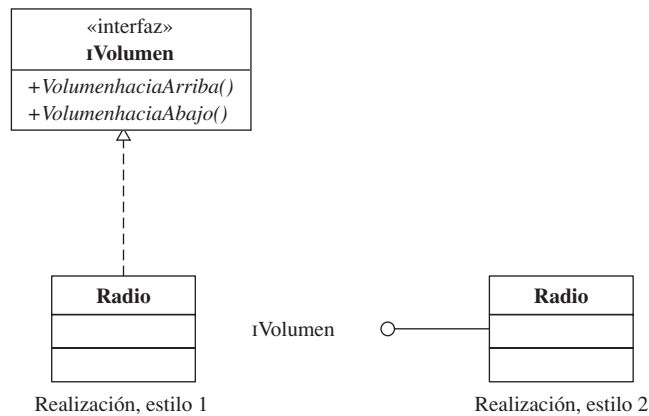


Figura 5-11 La realización, o herencia de interfaz, se puede mostrar en cualquiera de los dos estilos, como se ilustra en la figura.

Para ayudarle a familiarizarse con la herencia de interfaz, agregué un estilo alternativo a la derecha de la figura 5-11. Muchas herramientas de modelado soportan los dos estilos. Elija un estilo y adhiérase a él. (Yo prefiero el de la izquierda de la figura 5-11, descrito en el párrafo anterior.)

Modelado de dependencia

La *relación de dependencia* es de cliente y proveedor. Una clase, el *cliente*, depende de una segunda clase, el *proveedor*, para proporcionar un servicio. El símbolo para una relación de dependencia luce como una asociación unidireccional, excepto que la línea es punteada en lugar de continua (figura 5-12).

Suponga, por ejemplo, que decidimos soportar varios estilos de presentación para los usuarios de “BlackJack”. Podríamos ofrecer una consola, Windows o una interfaz gráfica web del usuario (GUI). A continuación, podríamos definir un método “Imprimir” que dependa de una “ImpresoradeCartas” específica. Si la “ImpresoradeCartas” es una impresora gráfica, entonces podríamos presentar un mapa de bits de la carta, pero si la “ImpresoradeCartas” es una impresora basada en DOS, entonces puede ser que sólo escribamos texto en la consola. En la figura 5-13, se muestra la relación de dependencia combinada con generalización para reflejar diversas clases de “ImpresoradeCartas”.

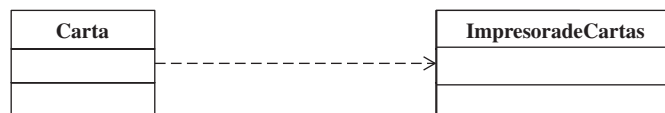


Figura 5-12 Esta figura muestra que la “Carta” depende de la “ImpresoradeCartas”, en donde “Carta” es el cliente e “ImpresoradeCartas” es el proveedor.

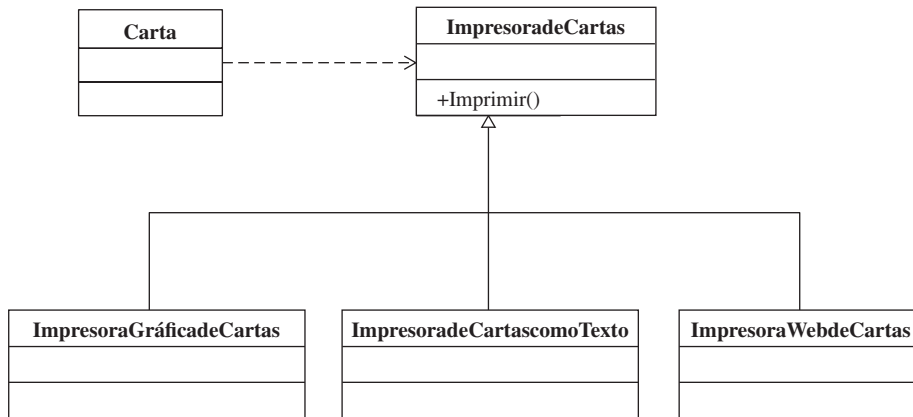


Figura 5-13 La relación de dependencia ahora incluye generalización que muestra tipos específicos de objetos “ImpresoradeCartas”.

SUGERENCIA Vale la pena hacer notar que en la figura 5-13 se introduce un concepto: resulta una buena práctica captar varias facetas de un diseño en diagramas separados. Por ejemplo, en esa figura, puede ser que no estemos mostrando todas las clases del juego “BlackJack”, pero estamos mostrando relaciones útiles entre la clase “Carta” y las clases que suministran impresión.

Otra característica útil es que los conectores como la dependencia se asocian con estereotipos predefinidos. Un estereotipo agrega significado. En el capítulo 6, examinaremos los estereotipos, cuando examinemos con mayor detalle cómo se relacionan las clases.

Estereotipado de las clases

El *estereotipo* es un medio por el cual el UML se puede extender y evolucionar. En forma visual, los estereotipos aparecen entre comillas angulares («estereotipo»). Hay varios estereotipos predefinidos para los símbolos de UML, como el clasificador; el lector tiene la libertad de adoptar nuevos estereotipos, si surge la necesidad. En la figura 5-11, se muestra un ejemplo en donde se usó el estereotipo «interfaz», con el fin de indicar que un clasificador representa una interfaz.

SUGERENCIA Algunas herramientas de modelado del UML reemplazarán a los estereotipos con símbolos específicos, cambiando la apariencia de un diagrama, aun cuando no se altere el significado. Por ejemplo, tanto el clasificador con el estereotipo «interfaz» como el círculo hueco de la figura 5-11 reflejan con exactitud la interfaz “iVolumen”.

Uso de paquetes

El símbolo *paquete* tiene la apariencia de una carpeta de archivos. Este símbolo (figura 5-14) se usa en forma genérica para representar un nivel más elevado de abstracción que el clasificador. Aun cuando, por lo común, un paquete se puede implementar como un espacio de nombre o un subsistema, con un estereotipo, también se puede usar para la organización general y sencillamente representar una carpeta de archivos.

SUGERENCIA *Los espacios de nombres resolvieron un problema, acarreado durante largo tiempo, de múltiples equipos de desarrollo que usan nombres idénticos para las clases. Una clase nombrada “Customer” (“Cliente”) en el espacio de nombre de Softconcepts es distinta de “Customer” en el espacio de nombre de IBM.*

En el juego “BlackJack” se usan las API contenidas en el cards.dll que vienen con Windows (y se usa en juegos como el Solitario). Podríamos usar dos paquetes y una dependencia para mostrar que el juego “BlackJack” depende de las API del cards.dll.

Uso de notas y comentarios

La anotación de diagramas es un aspecto importante del modelado. Los diagramas de clases permiten el uso de la nota, pero vea si puede transmitir tanto significado como sea posible sin agregar una gran cantidad de notas. (Vea la figura 5-15 en relación con un ejemplo del símbolo de nota con la punta doblada que se usa en el UML.)

Muchas herramientas soportan la documentación del modelo que se almacena con éste, pero que no se presenta en los diagramas. La documentación específica del modelo más allá de notas, comentarios y restricciones no es una parte real del UML, pero es un buen auxiliar para la creación de modelos.

Restricciones

En las restricciones se usa el mismo símbolo de punta doblada en todos los diagramas. En realidad, las restricciones pueden ser una parte engañosamente compleja del UML y pueden incluir información que ayuda mucho a los generadores de código. Por ejemplo, se pueden escribir restricciones en texto llano o en Object Constraint Language (OCL). Aunque a todo lo largo de este libro proporcionaré ejemplos de restricciones, de manera intencional omito una exposición del OCL como no muy desmitificadora.

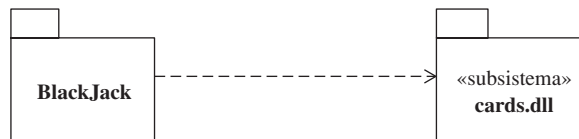


Figura 5-14 El diagrama muestra que el paquete “BlackJack” depende del paquete “cards.dll”, en el cual se usa el estereotipo «subsistema».

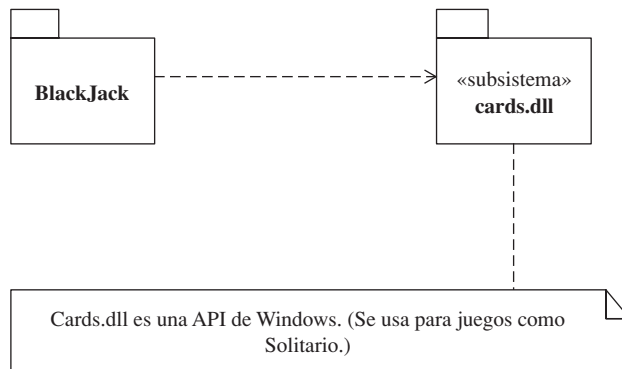


Figura 5-15 Se usa el rectángulo con la punta doblada para agregar notas o comentarios a los elementos de los diagramas UML.

Para demostrar una restricción, podemos agregar el símbolo correspondiente e introducir un texto de restricción que exprese que el número de cartas en un “Monte” debe ser 52 (figura 5-16). También es posible expresar esto sin una restricción, cambiando la multiplicidad del extremo de * al número 52. Otro ejemplo podría ser una restricción que

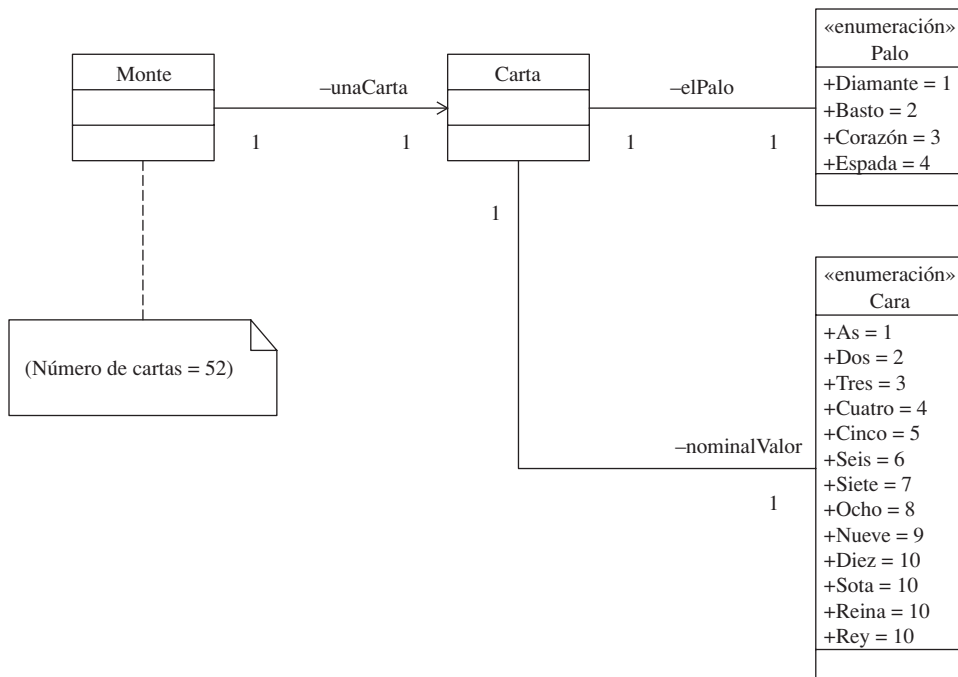


Figura 5-16 En esta figura se ilustra cómo podemos mezclar las restricciones —“Número de cartas = 52” en la figura— con otros elementos del diagrama para aumentar su precisión.

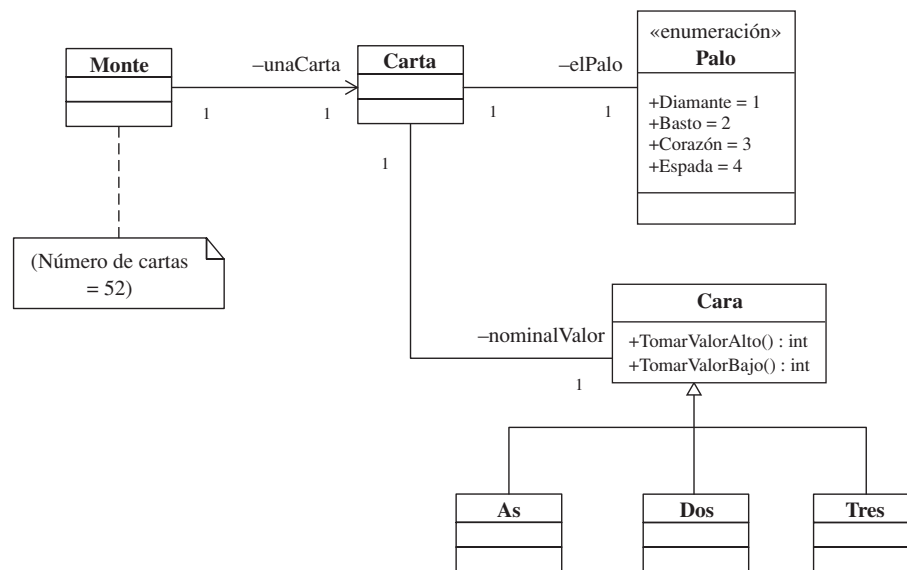


Figura 5-17 Diagrama de clases de la figura 5-16 modificado para captar el hecho de que las cartas pueden tener valores nominales dinámicos (int = entero).

expresar algo acerca del valor nominal o el número y variedad de palos, y también podríamos expresar estos elementos con enumeraciones.

En la figura 5-16, incluí la restricción de que el número de cartas en un “Monte” debe ser 52, una enumeración para indicar que hay cuatro palos y una enumeración para indicar que existen 14 valores nominales posibles únicos. Desafortunadamente, la figura todavía queda corta, porque en el juego “BlackJack” el as no tiene un valor sencillo único. Un análisis de este modelo con un experto en el dominio podría revelar con rapidez un problema posible con el uso de una enumeración para “Cara”. Debido al valor dual del as, podemos elegir volver a diseñar la solución para usar una clase —“Cara”— y una generalización —valores nominales específicos, como “As”, “Dos”, “Tres”, etc.— para resolver el problema de los ases (figura 5-17).

Modelado de primitivos

El UML define *primitivos* como “Integer”, “Boolean”, “String” y “UnlimitedNatural” (“Entero”, “Booleano”, “Cadena” y “Naturallimitado”) para usarse en la especificación del propio UML, pero la mayoría de los lenguajes y herramientas definen sus propios tipos primitivos. El lector puede modelar primitivos usando un clasificador, el estereotipo «primitivo» y el nombre del tipo.

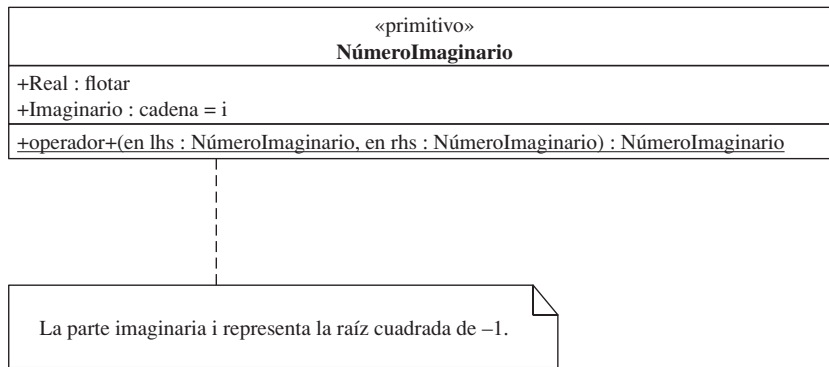


Figura 5-18 Los números imaginarios son números reales multiplicados por el número imaginario i , el cual representa la raíz cuadrada de -1 .

En general, los primitivos se modelan como atributos de otras clases. Sin embargo, en algunos casos usted tal vez deseará definir sus propios primitivos —siendo un ejemplo el número imaginario canónico (figura 5-18)—; existen algunos lenguajes, por ejemplo, Common Language Specification (CLS, especificación de lenguaje común) de Microsoft para .NET, en donde aparentemente los tipos primitivos en realidad representan objetos y se tratan como tales.

A veces resulta útil desarrollar primitivos, y es aceptable modelarlos como una clase usando el conector de asociación, como demostré con anterioridad en este capítulo. El diagrama de la figura 5-18 documenta un “NúmeroImaginario” y desarrolla lo que representan las partes real e imaginaria, así como la incorporación de un operador sobrecargado —una función operador— para el tipo primitivo.

SUGERENCIA Los lenguajes como C++, C# y, recientemente, incluso Visual Basic.NET soportan la sobrecarga de operadores; esto significa que los comportamientos para operadores como +, −, * y / se pueden definir para tipos nuevos. El modelado de tipos primitivos y los lenguajes que soportan la sobrecarga de operadores pueden ser muy útiles si el lector necesita definir tipos de datos extendidos en su solución.

Modelado de enumeraciones

Las *enumeraciones* son valores nombrados que tienen una semántica que significa mayor que su valor subyacente. Por ejemplo, se podrían usar los enteros 1, 2, 3 y 4 para representar los palos en un monte de cartas de juego, pero una enumeración tipo “Palo” que contiene cuatro valores nombrados transmite más significado (vea la figura 5-17).

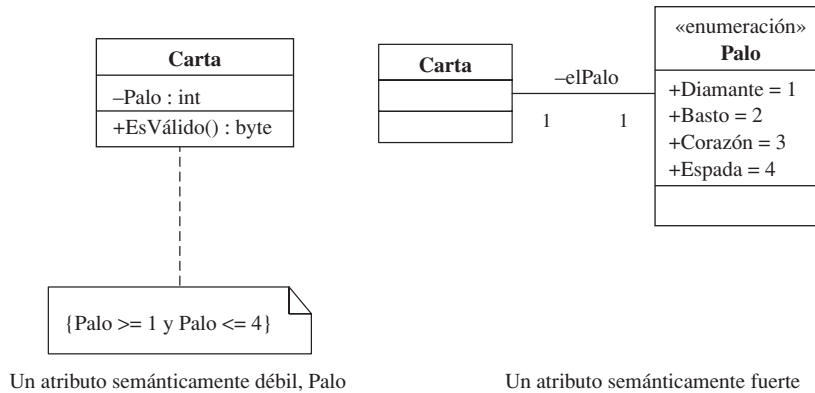


Figura 5-19 El entero “Palo” de la izquierda necesita explicación por el camino de una restricción con el fin de limitar y aclarar los valores posibles del entero, en tanto que la enumeración semánticamente más fuerte del “Palo” que se da a la derecha no necesita esa explicación.

Muchos lenguajes modernos soportan un sistema fuerte de tipos. Esto significa que, si usted define un argumento de enumeración, entonces sólo los valores definidos por esa enumeración son apropiados, y el compilador impone el uso de los valores del tipo más significativos semánticamente. En contraste con el uso de un tipo del tipo subyacente —por ejemplo, enteros para representar palos— que permitirían cualquier valor de ese tipo subyacente, las enumeraciones transmiten más información y rigor en el código, y más información en los modelos UML. En la figura 5-19 se ilustra este contraste.

NOTA A veces los modeladores y programadores hacen concesiones. Por ejemplo, podemos saber que una enumeración bien nombrada puede transmitir más significado, pero, de todas maneras, elegir no usar tipos semánticamente más fuertes. Suponga que, como el cereal Lucky Charms, los diamantes, bastos, corazones y espadas podrían evolucionar en el futuro; un monte de cinco cartas podría incluir tréboles. Si fuéramos a usar una enumeración, entonces tendríamos que abrir el respaldo del código en ese tiempo futuro y redefinir la enumeración. Sin embargo, si usamos un entero y almacenamos el rango de valores en una base de datos, entonces podríamos extender o cambiar los valores posibles de “Palo” ejecutando un comando SQL UPDATE. Saber acerca de estos tipos de juicios de valor y hacerlos es una de esas cosas que hacen que el desarrollo de software sea un reto.

Indicación de espacios de nombres

El *espacio de nombre* es una invención reciente en los lenguajes OOP. El espacio de nombre es una manera de agrupar elementos del código. El problema se originó a medida que las empresas de software empezaron a usar las herramientas de otro más ampliamente

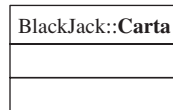


Figura 5-20 A menudo, los paquetes se codifican como espacios de nombres y se muestran en los diagramas UML en el lado izquierdo del operador de resolución de alcance, dos puntos dobles.

hasta que se hizo más común que el vendedor A produjera software útil con entidades nombradas de manera semejante a las del vendedor B. El espacio de nombre es una solución que permite que dos o más elementos nombrados en forma idéntica coexistan en la misma solución; el espacio de nombre establece una distinción entre estos elementos.

Con frecuencia, los paquetes son representaciones visuales de espacios de nombres, y éstos se pueden mostrar en diagramas para distinguir los elementos con el mismo nombre de clasificador. Se usa el operador de alcance `::` para concatenar un espacio de nombre con un elemento en ese espacio. Los espacios de nombres se pueden anidar, hermanar o disponer en cualquier manera jerárquica razonable en el contexto de un problema. Si la clase “Carta” se define como un elemento en el espacio de nombre “BlackJack”, entonces podemos captar esto agregando esa clase al paquete “BlackJack”, como se ilustra en la figura 5-20.

Cómo saber qué clases necesita

Existen dos modalidades para el desarrollo de software orientado a objetos: consumo y producción. Los equipos pueden trabajar en colaboración en cualquiera de las dos modalidades o en ambas, pero no entender si las habilidades de un equipo soportan el consumo de objetos, la producción de éstos, o ambas cosas, puede conducir a problemas.

Es perfectamente aceptable usar componentes, controles y objetos por otros y armar una solución tan bien como sea posible. La analogía más cercana a este estilo de desarrollo es la manera en que los programadores de C++ consideran a los programadores de Visual Basic (aunque esta creencia puede ser un poco injusta). En esta modalidad, un equipo se da cuenta de que su concepción de cómo usar los objetos es buena, pero que su propia producción de objetos es defectuosa. Una segunda modalidad aceptable es que un equipo sabe que es conocedor de los patrones de diseño y de la refactorización, y tiene una historia de éxitos en la arquitectura de soluciones orientadas a objetos, incluyendo la producción de sus propios objetos. Las dos modalidades son aceptables, pero es importante saber en cuál de ellas tiene usted la mayor oportunidad de éxito. (Como dijo Harry el Sucio: “Un hombre debe conocer sus limitaciones.”) Si va a tener éxito en la creación de modelos UML que describan algo más que las clases creadas por expertos, entonces necesitará saber cómo hallar las clases, así que hablemos de eso durante unos cuantos minutos.

NOTA En 2005, el autor Richard Mansfield, en un editorial publicado en DevX.com, desafió a oo (orientado a objetos) como un paradigma válido. Dejando a un lado todos los chistes acerca de perros viejos y nuevas bromas, Mansfield se anotó un punto de manera accidental. La cuestión es que si usted conoce oo suficientemente bien como para consumirlo pero trata de producirlo, entonces es posible que oo sea decepcionante. Sospecho que muchos proyectos oo fallan porque los consumidores competentes de oo no son productores tan competentes del mismo. La producción de objetos de calidad es difícil en el mejor de los casos, y sin conocimiento previo de patrones y refactorización, así como sin experiencia, puede ser imposible producir un oo bueno.

Hallar las clases correctas es lo más difícil que el lector hará; es mucho más difícil que trazar los diagramas. Si encuentra las clases correctas, bastan servilletas para modelar. Si no puede hallar las clases correctas, entonces no importa cuánto dinero gaste en herramientas; es posible que sus diseños den por resultado implementaciones fallidas.

Uso de un enfoque ingenuo

Cuando aprendí acerca de oo, fue por aprender primero C++ por mí mismo, un proceso muy doloroso, y entonces dí la vuelta para leer acerca de oo. Lo primero que aprendí fue que se trataba de hallar los nombres y después asignarles verbos. Los nombres se convierten en clases y los verbos en métodos. Ésta es la parte fácil, pero es posible que produzca sólo alrededor del 20% de las clases que usted necesitará.

Si el análisis sólo conduce a los nombres y verbos descritos por el dominio, entonces habrá un déficit de clases y se requerirá gran cantidad de habilidad en computación (hacking, “hackeo”). No obstante, empezar con los nombres y los verbos del dominio es un buen inicio.

Descubra otros beneficios del análisis de dominios

Además de las cosas que los expertos de sus clientes le digan, también necesitará concebir cómo poner estas cosas a la disposición de sus clientes y, en casi todas las circunstancias, guardar la información que proporcionen los usuarios. Estos fragmentos de información se conocen en forma genérica como clases: de *frontera*, de *control* y de *entidad*. Una clase frontera es aquella que se usa para conectar elementos exteriores al sistema con elementos del interior. Las clases de entidad representan datos. Por lo común, las entidades representan datos que persisten, como los que el lector podría encontrar en una base de datos, y las clases de control administran otras clases o actúan sobre ellas. Por lo general, los usuarios le dicen a usted mucho acerca de las clases de entidad, y esto puede ayudar a definir las GUI con base en cómo completan ellos las tareas, pero debe trabajar mucho más para hallar las clases de control y frontera.

SUGERENCIA *Si alguna vez ha trabajado como analista, no diga: “Me ha hablado acerca de las clases de entidad; hábleme ahora de las clases de frontera.” El análisis es una tarea importante y es posible que no deba dejarse a aquellos que usan protectores de lápices en los bolsillos de sus chalecos. Las habilidades interpersonales y un enfoque de baja tecnología, en tono de conversación, producen un buen intercambio de ideas.*

Una perspectiva importante es saber que los expertos de la empresa le dirán mucho acerca de los datos que tienen que almacenar, algo acerca de los procesos que siguen para obtener los datos y un poco acerca de una buena manera de capturar esos datos en una computadora. Una segunda perspectiva importante es que los usuarios —los asignados para explicar las cosas a los ingenieros especialistas en software se llaman *expertos del dominio*— pueden hacer una gran cantidad de cosas que no tienen sentido para los que se encuentran fuera. Desde un punto de vista racional, esto significa que un ingeniero de procesos puede ser que nunca haya trabajado con su organización para examinar qué es lo que hace esa organización y cómo lo hace, y para determinar si existe una mejor manera de hacerlo. El resultado es que puede ser que usted obtenga una gran cantidad de información que no se pueda traducir bien en software —lo que se llama una *razón baja señal-a-ruido*—, pero el experto del dominio puede sentir que es importante.

SUGERENCIA *Cuando se llega al análisis, el mejor consejo que puedo ofrecer es comprar una pluma cara y un cuaderno de notas forrado en piel, enfrascarse activamente en la conversación y tomar copiosas notas. Además de hacer que los usuarios se sientan halagados de que se les esté poniendo una atención tan espléndida, es difícil saber con tanta prontitud en el análisis qué constituye la señal y qué el ruido, de modo que una gran cantidad de información es buena.*

Habiendo aprendido de los usuarios acerca de las clases de entidad, su trabajo es concebir cuáles son las clases fronteras y de control, y cómo modelarlas. El modelado es más fácil, de modo que empecemos allí.

Bastante sencillo, una *clase entidad* está constituida por datos y suele tener larga vida o persistir, y las clases de entidad se pueden modelar añadiendo el estereotipo «entidad» al símbolo de clase o usando el símbolo de clase entidad, del que se dispone en muchas herramientas de modelado (figura 5-21). Una *clase de control* es un código transitorio que, en general, controla otras clases o actúa sobre ellas, y es responsable de transportar los datos entre las clases de entidad y las clases de frontera. Las clases de control se modelan agregando el estereotipo «control» a una clase o usando el símbolo de clase (también mostrado en la figura 5-21). *Las clases de frontera* suelen encontrarse entre subsistemas. Éstas se pueden modelar como se muestra en la figura 5-21 o adornando una clase con el estereotipo «frontera».

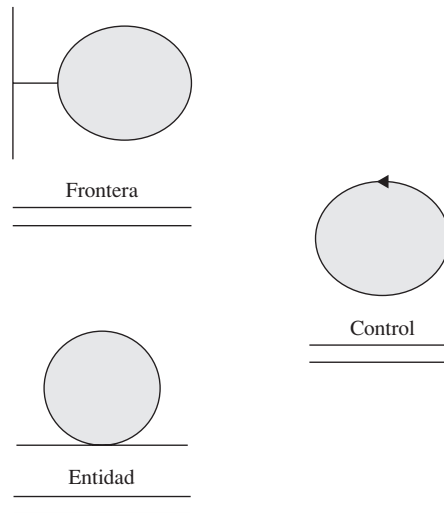


Figura 5-21 Los símbolos rectangulares para las clases y los estereotipos se pueden reemplazar con símbolos que representan específicamente las clases de frontera, control y entidad.

Una inclinación de cabeza para los ejercicios CRC

Las fichas de responsabilidades y colaboradores de las clases (CRC, class responsibility and collaborator) constituyen un concepto que comprende un uso de baja tecnología de fichas de 3×5 . La idea es que un grupo de personas interesadas se reúnan y escriban, en la parte superior de una ficha, las clases que han descubierto. Debajo escriben una lista de responsabilidades, y a un lado de las responsabilidades escriben los colaboradores de clase necesarios para apoyar esas responsabilidades. Si no existe ficha para una responsabilidad, entonces se crea una nueva ficha.

La idea básica que se encuentra detrás del uso de pequeñas fichas es que son demasiado pequeñas como para contener una gran cantidad de comportamientos, lo cual está dirigido a una división razonable de las responsabilidades.

La creación de fichas CRC es una buena idea, pero puede ser que el lector quiera traer un experto para hacer caminar su grupo a través de este paso el primer par de veces. Como ése es un consejo práctico, pero no puedo meter un experto en CRC en este libro, hablaré acerca de alternativas, las cuales se describen en las tres subsecciones próximas.

Manera de hallar clases de entidad

Como mencioné con anterioridad, las clases de entidad representan los datos que necesitará almacenar. También abarcan entidades lógicas. Por lo común, una entidad lógica está constituida por vistas o por el resultado de consultas heterogéneas; por ejemplo,

seleccionar campo1, campo2 del cliente, pedidos en donde pedido.clienteid = cliente.id

De manera simplista, esta consulta produce un resultado proveniente de cliente y pedidos, lo cual representa una entidad lógica cliente - pedidos.

Hallar entidades y entidades lógicas es relativamente fácil, debido a que la teoría de la base de datos relacional está bastante bien comprendida, y las bases de datos relacionales constan de un depósito significativamente recurrente para las entidades. El lector necesitará entidades para tablas simples y vistas heterogéneas compuestas por tablas múltiples. De ese punto en adelante, las entidades se modelan sencillamente como clases. Puede usar un estereotipo «tabla» si las entidades representan tablas, o ningún estereotipo particular si usa clases personalizadas.

Modo de hallar clases de control

Las clases de control representan el puente entre las clases de entidad y las clases de frontera, y la lógica de la empresa entre ellas. La manera en que implemente estas clases depende de su estilo de implementación. Si selecciona un estilo de implementación, entonces la manera de hallar las clases de entidad se puede derivar desde allí.

Suponga que la herramienta de implementación que usted selecciona predefine clases como filas, tablas y conjuntos de datos. Si elige usar las clases de su herramienta, entonces sus clases de entidad se compondrán de esas clases, y las clases que forman el puente hacia sus clases de entidad se definirán por medio del marco de referencia de su herramienta. Por otra parte, si selecciona clases de entidad personalizadas, entonces éstas serán análogas a filas, tablas y conjuntos de datos, pero las clases de control todavía serán las clases del marco de referencia que lee su almacén de persistencia y escribe en él o a partir de él, por lo común una base de datos.

Las clases de control pueden administrar la manera en que los datos se ponen en orden para las clases de entidad, la manera en que se ponen en orden para las clases de presentación y la manera en que se ponen en orden para otros sistemas, a través de las clases de frontera. Existen muchos patrones que incluyen patrones de control general; la clave es reconocerlos. Un patrón famoso se llama *controlador de la vista del modelo* (MVC, model view controller). En el MVC, el modelo se representa por objetos de la empresa, la GUI es la vista de usted, y las clases de control entre ellos representan su controlador. La implementación del MVC o el reconocimiento de una implementación del mismo requiere estudio y práctica adicionales. Por ejemplo, Microsoft considera que las páginas ASP.NET en .NET son una implementación del MVC. La página ASPX o HTML es la vista, el controlador es el código que se encuentra detrás de la página y el modelo está constituido por los objetos cuyos datos se muestran en esa página. La implementación de un patrón MVC personalizado en este contexto sería redundante. Existen muchos libros sobre patrones; *Design Patterns* (Reading, MA: Addison-Wesley, 1995), escrito por Erich Gamma es un buen lugar para empezar.

NOTA Hay muchos patrones de diseño que pueden guiarlo cuando busca clases de frontera, control o entidad. Una clave aquí es seleccionar un estilo de implementación y adherirse a él. El lector puede componer una solución hallando primero las entidades —llamado composición de la base de datos— o hallando objetos de la empresa —llamado composición de objetos—, o diseñando primero las GUI —llamado composición de la presentación o, a veces, mencionada como habilidad en la computación (“hacking”)—. Cualquiera de estos estilos de composición pueden tener éxito, pero algunos de ellos funcionan mejor que otros, dependiendo del tamaño y complejidad del problema. Desafortunadamente, no hay un solo mejor estilo para todas las circunstancias, y las opiniones sobre este tema varían mucho.

Modo de hallar las clases frontera

Las clases frontera se usan para formar puentes hacia los subsistemas. En este caso, el objetivo es aislar su sistema de la interacción directa con subsistemas externos. De esta manera, si el subsistema externo cambia, su implementación sólo necesitará cambiar en las clases fronteras. Aquí pueden ayudar un buen conocimiento de los patrones y un estudio de los sistemas que han tenido éxito.

Este libro es acerca del UML y no pretende que ser un how-to sobre el diseño de software. Sin embargo, un recorrido por la bibliografía le conducirá a algunos libros excelentes sobre el UML y el diseño de software.

Examen

1. Se usa el mismo símbolo básico para las interfaces y las clases.
 - a. Verdadero
 - b. Falso
2. Al agregar clases a un diagrama, usted debe
 - a. mostrar propiedades, campos y métodos.
 - b. mostrar sólo propiedades y campos.
 - c. mostrar propiedades y métodos.
 - d. mostrar campos y métodos.
3. Un atributo se puede modelar como una característica de una clase, pero no como una clase asociación.
 - a. Verdadero
 - b. Falso

4. Al modelar atributos, se
 - a. requiere que modele métodos atributos.
 - b. recomienda que no muestre métodos atributos.
 - c. recomienda que muestre los campos subyacentes para esos atributos.
 - d. Ninguno de los anteriores.

5. Tanto los tipos simples como los complejos se deben modelar como
 - a. atributos.
 - b. clases asociación.
 - c. atributos y clases asociación.
 - d. Los tipos simples se modelan mejor como atributos, y los complejos se modelan mejor como asociaciones.

6. Una asociación unidireccional tiene una flecha en uno de los extremos, conocido como la fuente; el otro extremo se conoce como el objetivo.
 - a. La fuente tendrá un campo cuyo tipo es el del objetivo.
 - b. El objetivo tendrá un campo cuyo tipo es la fuente.
 - c. Ninguno de los dos.

7. ¿Una agregación y una asociación son
 - a. semánticamente semejantes?
 - b. directamente opuestas?

8. ¿Cuál es la diferencia más importante entre una agregación y una composición?
 - a. Composición significa que la clase totalidad, o compuesta, será responsable de la creación y destrucción de la parte o clase contenida.
 - b. Agregación significa que la clase agregada totalidad será responsable de la creación y destrucción de la parte o clase contenida.
 - c. Composición significa que la clase totalidad, o compuesta, es la única clase que puede tener un caso de la clase parte en cualquier momento dado.
 - d. Agregación significa que la clase totalidad, o agregada, es la única clase que puede tener un caso de la clase parte en cualquier momento dado.
 - e. a y c
 - f. b y d

9. Generalización significa
 - a. polimorfismo.
 - b. asociación.
 - c. herencia.
 - d. composición.

10. A una asociación se le da nombre. El nombre es
 - a. el tipo de la clase asociada.
 - b. el nombre implicado de la asociación y representa el nombre de un campo.
 - c. una dependencia.
 - d. una generalización.

11. El «primitivo» se usa en conjunción con el símbolo de clase. Éste introduce
 - a. tipos simples existentes.
 - b. tipos nuevos semánticamente simples.
 - c. tipos complejos existentes.
 - d. tipos nuevos semánticamente complejos.

Respuestas

1. a
2. d
3. b
4. b
5. a
6. a
7. e
8. c
9. b
10. b
11. b

Cómo se relacionan las clases

En el capítulo 5, se introdujeron los diagramas de clases como vistas estáticas de su sistema. Por *vista estática*, quiero decir que las clases sólo están ahí, pero sus clases definen las cosas que se usan para examinar comportamientos dinámicos descritos en diagramas de interacción y esquemas de estado.

Debido a que las clases y los diagramas de clases contienen elementos centrales para el sistema del lector, ampliaré el uso básico de los símbolos y las relaciones básicas del capítulo 5. En este capítulo, se examinarán relaciones más avanzadas e información más detallada de las clases, estudiando

- Diagramas con un mayor número de elementos
- Relaciones anotadas, incluyendo la multiplicidad
- Modelado de clases abstractas e interfaces
- La manera de agregar detalles a los diagramas de clases
- La comparación de la clasificación con la generalización

Modelado de la herencia

Existen beneficios al heredar, así como retos. Una clase hijo hereda todas las características de su clase padre. Cuando se define un atributo en una clase particular, es incorrecto repetir el atributo en las clases hijos. Si repite un método en la clase hijo, entonces está describiendo la anulación del método. En el Unified Modeling Language (UML), además de anular, puede redefinir los métodos; esto se soporta en algunos lenguajes, pero puede conducir a confusión. La anulación de los métodos es central para el polimorfismo; use la redefinición de métodos con moderación.

Cuando hereda clases, sus clases hijos heredan las restricciones definidas por todos los antepasados. Cada elemento tiene la unión de las restricciones que define y las restricciones definidas por sus antepasados.

El lector tiene varias opciones de herencia que explicaré en esta sección. En esta sección, se considerarán la herencia simple y la múltiple, y se comparará la generalización con la clasificación. Para evitar árboles profundos de herencia, también explicaré la herencia de interface y composición en las dos secciones que siguen.

Uso de la herencia simple

La *herencia simple* es la forma más fácil de herencia. Una clase hijo que hereda de una clase padre hereda todas las características de esta última, pero sólo tiene acceso directo para los miembros públicos y protegidos. La herencia, llamada *generalización* en el UML, se indica por una sola línea que se extiende de la clase hijo a la clase padre, con un triángulo hueco fijado a esta última. Si múltiples clases heredan de la misma clase padre, entonces puede usar una sola línea unida que se conecte a esta última.

Generalización contra clasificación

En el capítulo 5, introduje una prueba fácil para determinar si existe una relación de herencia. Ésta se llama *es una prueba*. Esta prueba sola puede ser engañosa y conducir a resultados incorrectos. Es una prueba implica transitividad estricta. Por ejemplo, si la clase B es una hija de la clase A, y la clase C es una hija de la B, entonces la clase C es una hija de la clase A. (Decimos que la clase C es una *nieta* de la clase A o que la A es una *antepasada* de la C.) No obstante, la transitividad implicada por la prueba es una no es estrictamente correcta.

Supongamos que tenemos las proposiciones verdaderas que siguen:

Pablo es un programador de C#.

Programador C# es una descripción de trabajo.

Pablo es una persona.

Programador de C# es una persona.

“Pablo es un programador de C#” funciona. “Un programador de C# es una persona” funciona y “Programador de C# es una descripción de trabajo” funciona, pero “Pablo es una descripción de trabajo” no funciona. El problema es que Pablo es un ejemplo de programador de C#, pero la generalización (es decir, la herencia) se usa para describir relaciones entre subtipos. Por lo tanto, tenga cuidado al usar es una como un solo determinante de la herencia. Una prueba más precisa es determinar si algo describe un ejemplo (clasificación) o un subtipo (generalización).

Si la clase B es un subtipo de la A, entonces tiene usted una herencia. Si la relación describe una clasificación —es decir, describe un contexto o papel en el cual algo es cierto— entonces tiene un relación de clasificación. Las clasificaciones se pueden manejar mejor con las asociaciones.

Clasificación dinámica

La exposición anterior sugiere que la herencia a veces se aplica mal. Regresando a nuestro ejemplo, Pablo es un programador describe un papel, o clasificación, más precisamente que una generalización, porque Pablo también es un esposo, un padre y un pagador de impuestos. Si hubiéramos tratado de generalizar a Pablo como todas esas cosas a través de la herencia, hubiéramos tenido que usar herencia múltiple, y las relaciones hubieran sido bastante complejas.

La forma de modelar y captar la clasificación es a través de la asociación. De hecho, podemos usar un patrón de comportamiento de estado para captar la dinámica y los papeles cambiantes que describen a una persona o la manera como se comporta el ejemplo Pablo en un contexto dado. Usando una asociación y, de modo específico, el patrón de comportamiento de estado, podemos implementar la clasificación dinámica; es decir, podemos cambiar el comportamiento de Pablo con base en el contexto o el papel que está representando en un momento dado.

El *patrón de comportamiento de estado* se implementa con el uso de una asociación y generalización. Sin tratar de reproducir la discusión completa acerca de este patrón —consulte *Design Patterns*, escrito por Erich Gamma *et al.*— podemos hacer un resumen. El patrón de comportamiento de estado se llama patrón de *comportamiento* porque describe la manera en que actúa algo. Los otros tipos generales de patrones son de *creación* —cómo se crea algo— y *estructural* —cómo se organiza algo—. El patrón se llama de *estado* porque describe cómo se comporta algo con base en el estado. En nuestro ejemplo, usaríamos este patrón para describir cómo se comportan las personas con base en alguna condición: el estado. Por ejemplo, cuando Pablo está en el trabajo, se comporta como un programador de C#. Cuando Pablo está en casa, se comporta como un esposo al interactuar con su esposa y como padre cuando interactúa con sus hijos.

Si modelamos en forma incorrecta la clasificación de Pablo con el uso de la generalización, entonces crearíamos un modelo como el de la figura 6-1, mostrando toda la he-

rencia. Sin embargo, si modelamos los papeles de Pablo de modo más preciso con el uso de la asociación, tendríamos un modelo mejor (vea la figura 6-2).

En la figura 6-1, se intenta mostrar que una “Persona” es un ejemplo de “Programador”, “Esposo” y “Padre”. En realidad, esto implica que debería crearse un tipo diferente de objeto para Pablo, dependiendo del contexto. Sin embargo, en realidad, Pablo siempre es una persona y las personas representan papeles: a veces, una persona es un esposo; a veces, un padre; a veces, un trabajador, y así sucesivamente. El papel de la asociación significa que Pablo siempre es un ejemplo de “Persona”, pero el papel de una persona cambia en forma dinámica. La clase en cursivas, “*Papel*”, significa que el papel es abstracto y la asociación “papel” (en minúsculas) en realidad es un caso de “Programador de C#”, “Esposo” o “Padre”.

El patrón de comportamiento de estado se implementa principalmente por la relación entre “Persona” y la clase abstracta “*Papel*”. Lo que está faltando para completar el patrón son los comportamientos abstractos que es necesario definir por persona e implementarse por generalizaciones del papel. Por ejemplo, “Persona” podría tener un método llamado “ProcederconPaciencia” y ese método se declararía en “*Papel*” y se implementaría llamándolo “papel”. Se implementaría “ProcederconPaciencia”, es decir, el comportamiento de “Persona” nombrado “ProcederconPaciencia”, por medio de una subclase específica de “*Papel*”. Por ejemplo, en el papel de “Programador de C#”, si usted les grita a los clientes, entonces puede perder su trabajo; pero gritarle a su esposa puede dar por resultado que usted duerma en el sofá. El subtipo específico del papel determina el comportamiento, sin cambiar el ejemplo de “Persona”.

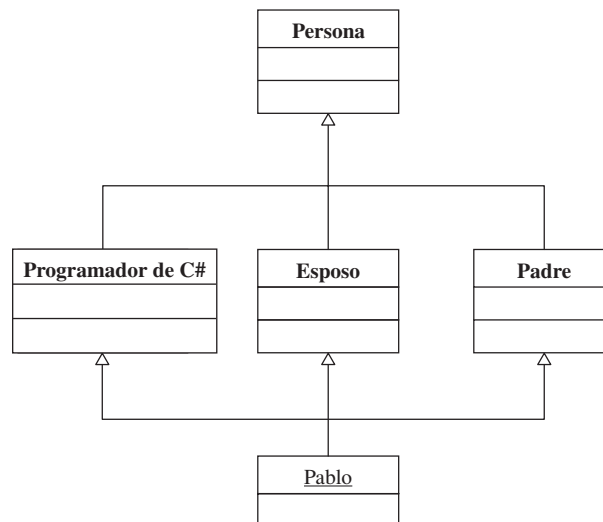


Figura 6-1 Diagrama UML de clases en el que se muestra una generalización rígida donde el objeto “Pablo” intenta reflejar de manera incorrecta “Padre”, “Esposo” y “Programador de C#”.

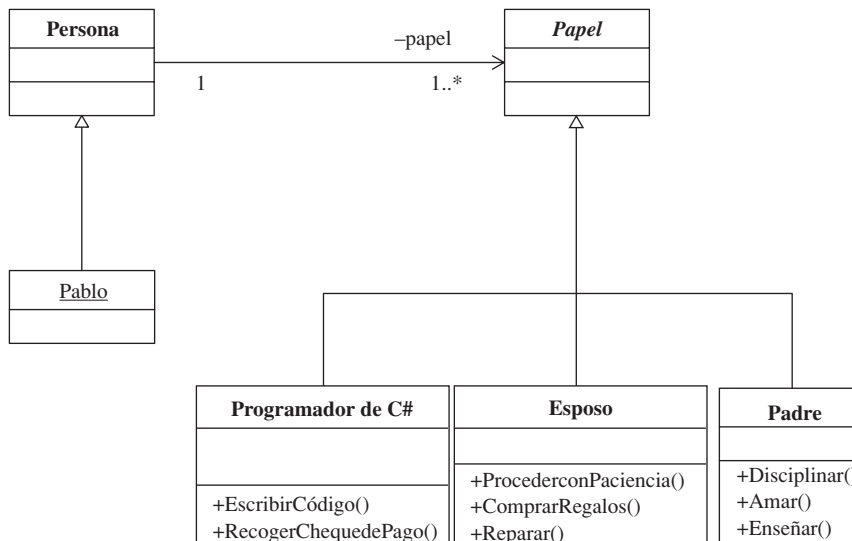


Figura 6-2 Un segundo diagrama UML de clases en el que se usa la asociación para un papel que refleja cómo se comportan las personas en ciertos papeles.

NOTA En la compleja sociedad de hoy en día, modelar las relaciones familiares, por ejemplo, para el gobierno estatal, podría ser excesivamente difícil. Los niños tienen múltiples padres y madres, a veces el sexo de los dos padres es idéntico y algunas personas tienen trabajos múltiples y familias nucleares. Sin embargo, esto ilustra que algo tan aparentemente sencillo, como la gente y sus papeles, puede ser muy complejo, dependiendo del dominio del problema.

Si el género va bien con el contexto de nuestro diseño, entonces podríamos clasificar todavía más “Esposo” y “Padre” asociado con una enumeración, “Género” (figura 6-3). La clave es no modelar todo lo que podría; en lugar de ello, modele lo que necesite modelar con el fin de describir el problema de manera suficientemente adecuada para su espacio de problema.

Uso de la herencia múltiple

La herencia sencilla puede ser difícil porque la prueba es un(a) no es por completo suficiente. La generalización implica subtipo, pero el lector podría implementar relaciones de subtipo con el uso de la composición o asociación. La herencia simple representa, además, un reto porque la clasificación significa que usted está hablando de un caso y es un(a) parece funcionar durante las discusiones verbales, pero puede ser incorrecta o demasiado rígida para implementar.

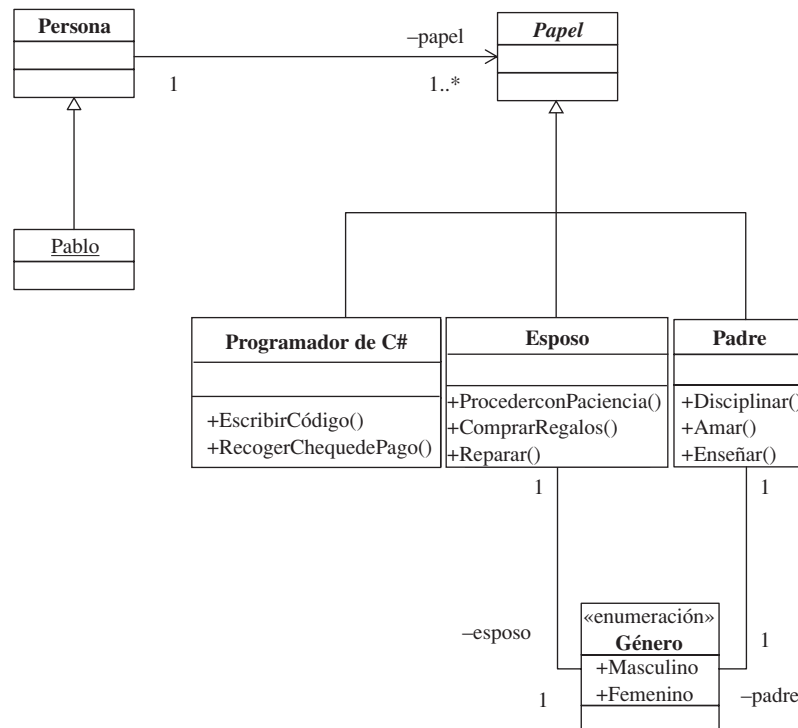


Figura 6-3 Abstracción del género a partir de los papeles de esposo y padre.

La herencia múltiple es incluso más difícil porque todavía tenemos los problemas de generalización y clasificación, y éstos se exageran al tener más de un supertipo. Cuando un subtipo hereda de más de un supertipo, se entiende que aquél contiene la unión de todas las características de todas sus clases padres. Hasta ahora todo va bien. Se presenta un problema cuando más de un supertipo introduce una característica que tiene el mismo nombre que la de otro supertipo. Por ejemplo, la clase C hereda de la B y de la A, y tanto la clase A como la B introducen una operación nombrada “Foo”. ¿Por cuál versión de Foo se resuelve “C.Foo()”, por “A.Foo()” o por “B.Foo()”? Aun cuando el UML soporta la resolución dinámica de conflictos, la mayoría de las implementaciones de herencia múltiple requieren que el programador resuelva el conflicto. Esto significa que el programador debe decidir que “C.Foo()” llama a “A.Foo()”, “B.Foo()” o tanto a “A.Foo()” como a “B.Foo()” (figura 6-4). Una buena práctica al usar la herencia múltiple es resolver los conflictos de nombre de manera explícita.

Queda indicada la herencia múltiple cuando una clase tiene más de un supertipo inmediato. Albert Broccoli e Ian Fleming, el mismo par que produjo los filmes de James Bond, produjeron la película *Chitty Chitty Bang Bang*. En la película, el auto también era una nave sobre colchón de aire, propulsada por un chorro de agua y un avión. En un

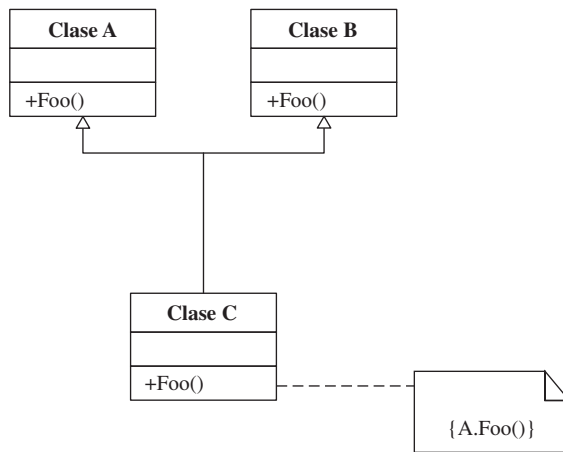


Figura 6-4 Resuelva de manera explícita los conflictos de nombres en las clases con herencia múltiple, lo que se muestra aquí con el uso de una restricción.

diagrama de clases, esto se podría modelar como una clase (la llamaremos “CCBB”) que se herede de “Bote”, “Automóvil” y “Avión”. El problema surge porque en cada modo se usó una forma diferente de propulsión; como consecuencia, “CCBB.Propulsar()” podría ser difícil de resolver en un modelo, e igualmente difícil de implementar.

NOTA *El lector podría pensar que los vehículos anfibios y el verbo propulsar resultan un poco forzados, pero con base en la experiencia real, le puedo decir que, en la actualidad, esos conceptos existen en los diseños. Sin embargo, yo sólo tengo conocimiento de ejemplos reales en aplicaciones militares.*

Debido a las dificultades técnicas reales con la herencia múltiple, muchos lenguajes poderosos, como C# y Java, no soportan el idioma. Otra razón por la que la herencia múltiple no se soporta en forma universal es que usted puede simular este tipo de herencia mediante la composición y la promoción de características constituyentes, o a través de herencia múltiple de interfaces. Desde la perspectiva del UML, la composición y las características constituyentes que la revisten significan que “CCBB” sería un auto y contendría los objetos avión y bote, y las características de bote y avión se harían disponibles en forma indirecta redefiniendo las características al nivel de auto. Entonces se implementarían estas características al invocar las características compuestas internamente de bote o avión. Por ejemplo, “CCBB.Volar” invocaría el método interno “Avión.Volar”. La herencia múltiple de interfaces tan sólo significa que una clase implementará todas las características definidas por todas las interfaces realizadas.

Evite la herencia múltiple, incluso si es soportada en el lenguaje de implementación que usted elija o, de lo contrario, use la composición o la herencia de interfaz. En la figura 6-5,

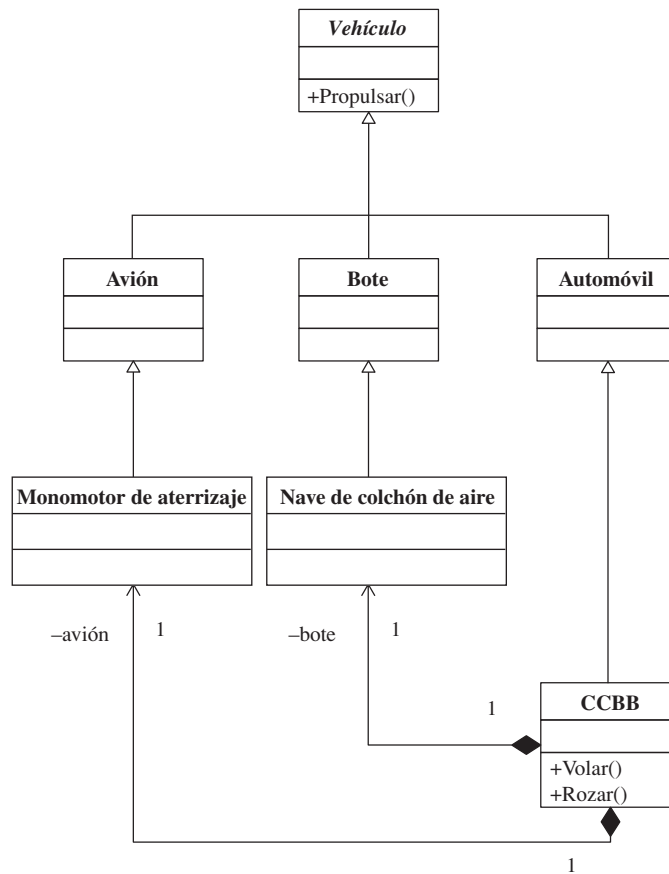


Figura 6-5 En esta figura, mostramos que “CCBB” hereda de “Automóvil”, pero usa la composición para mostrar sus capacidades de bote y avión.

se muestra una manera en que podríamos trazar un diagrama de clases para ilustrar las características aerodinámicas y anfibas de CCBB. En la figura, entendemos que el diagrama quiere decir que “CCBB” crea un ejemplo de “Nave sobre colchón de aire” nombrada “bote” y un ejemplo de “Aeronave monomotor de aterrizaje” nombrada “avión”. “Rozar()” se implementaría al llamar “bote.Propulsar”, y se implementaría “Volar()” al llamar “avión.Propulsar”.

Otra opción sería definir tres interfaces: “Avión”, “Automóvil” y “Bote”. Cada una de estas interfaces podría definir métodos, “Volar”, “Conducir” y “Propulsar”. Entonces “CCBB” podría implementar cada una de estas interfaces.

La solución que se muestra en la figura 6-5 no es perfecta, y puede no ser atractiva para todos; no obstante, es importante tener presente que nos estamos esforzando por lograr modelos *suficientemente buenos* o *susceptibles* de lograrse, no perfectos.

Modelado de la herencia de interfaces

Existen tres actividades primarias asociadas con el modelado. Los modeladores necesitan concebir con rapidez una solución para los problemas y, a menudo, en situaciones de grupo. Los modeladores tienen que usar las herramientas UML y esto, con frecuencia, lo hace una persona en aislamiento o en un grupo pequeño y, por último, en general se solicita documentación de apoyo en forma de texto. Escribir la documentación arquitectónica está más allá del alcance de este libro, pero tanto el modelado en grupo, sobre servilletas y pizarrones blancos, como el uso de las herramientas UML son importantes. A veces, pienso que los pizarrones blancos y las servilletas son más importantes que las herramientas UML, porque el modelado en grupo hace intervenir a más personas y no estoy por completo convencido de que los modelos UML reales sean leídos por cualesquiera que no sean los modeladores y programadores y, en ocasiones, sólo por los programadores.

Boceto de diagrama

Trazar un esbozo de los modelos puede resultar conveniente porque es fácil de hacer y se puede obtener retroalimentación del grupo que observa y cambiar el dibujo. Sin embargo, si intenta usar sobre un boceto la formalidad y las características de las herramientas UML, se quedará atascado en el dibujo de imágenes bonitas, en lugar de en la solución de los problemas. Por esta razón, es aceptable usar notaciones abreviadas y símbolos más pequeños, y no importa si sus rectángulos no son perfectos sobre un pizarrón.

Por ejemplo, en el UML, usamos cursivas para indicar que una clase es abstracta. Sobre un boceto, podemos usar una abreviatura para la palabra clave abstracta (A), para dar a entender que una clase es abstracta. En lugar de escribir el estereotipo «interfaz» para las interfaces, podemos usar «I» o la paleta de caramelo. Suponga que estamos discutiendo las propiedades del vuelo en un escenario de grupo. Podríamos definir una interfaz “IAptoparaVolar” con los métodos “TenerResistenciaalMovimiento”, “TenerSustentación”, “TenerEmpuje” y “TenerPeso”, y mostrar que un paracaídas implementa estas operaciones (aun cuando es muy difícil simular un boceto en un libro). En la figura 6-6, se muestra cómo podríamos presentar el UML sobre un pizarrón; en la 6-7 se muestra el mismo UML captado en nuestra herramienta de modelado.

La figura 6-7 es más clara y mejor que el UML, pero muchos modeladores, en especial aquellos con poca experiencia en el UML, reconocerán que las dos presentaciones representan la misma solución. Además, la sola explicación de qué es la paleta de caramelo satisfará a los principiantes, y es mucho más fácil de dibujar sobre un pizarrón.

NOTA “Sustentación”, “Resistencia al movimiento”, “Peso” y “Empuje” son los valores necesarios para los principios del vuelo descritos por la física de Bernoulli y Newton. En realidad, estas propiedades vinieron a colación cuando estaba exponiendo soluciones para

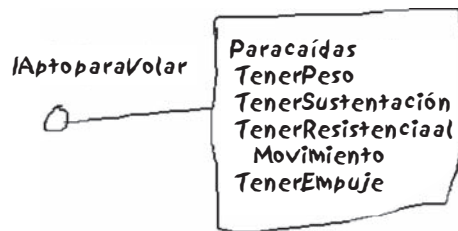


Figura 6-6 Se muestra realizada la interfaz “IAptoparaVolar” por medio de “Paracaídas”, como podríamos dibujarla sobre un pizarrón.

un sistema de evitación de colisiones de compañero contra compañero entrelazados, para los paracaídas de alta velocidad que usan los paracaidistas de gran altitud, o Halo.

La cuestión es que, en una situación dinámica de grupo, resulta de ayuda ser rápido, porque se puede botar una gran cantidad de información, a veces toda a la vez. El uso de una notación abreviada puede no dar siempre como resultado un UML perfecto, pero el lenguaje es una herramienta para entender y resolver problemas, y es el medio, no el fin. Usted siempre puede trazar un UML bonito cuando la reunión haya terminado.

Uso de la realización

Si la generalización se usa en exceso, entonces es posible que la realización se use menos de lo debido. *Realización* significa herencia de interfaces y se indica al usar una clase con el estereotipo «interfaz» y un conector con una línea punteada conectada a un triángulo hueco. El triángulo se fija a la interfaz, y el otro extremo se fija a la clase que implementará la interfaz.

El símbolo de la paleta de caramelo dibujado a mano en la figura 6-6 todavía se usa en algunas herramientas de modelado y se trata de una forma abreviada que se puede reconocer junto con un conector de línea continua para la herencia de interfaces. La dificultad en el uso de símbolos múltiples para dar a entender lo mismo es que hace que el lenguaje sea más difícil de entender y, si se usa de manera imprecisa, puede conducir a afectar el

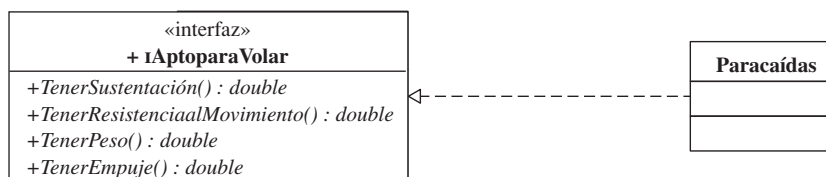


Figura 6-7 El mismo diagrama que se muestra en la figura 6-6, presentado en Visio (double = doble).

lenguaje por parte de los pequeños del UML. La afectación del lenguaje es casi siempre un desperdicio de tiempo, excepto para los académicos.

Relaciones de proveedor y relaciones requeridas

En el UML, la paleta de caramelo se usa en realidad para mostrar relaciones entre interfaces y clases. La paleta significa que la clase fijada proporciona la interfaz. Una mitad de paleta de caramelo o una línea con un semicírculo significa que se requiere una interfaz. Si aplicamos los símbolos para las relaciones requeridas y del proveedor a nuestro ejemplo del paracaídas, entonces podemos modelar nuestros paracaídas de alta velocidad proporcionando “IAptoparaVolar” (a la izquierda) y requiriendo “INavegable” a la derecha (figura 6-8).

En la figura 6-8, el propio paracaídas tiene propiedades de vuelo, incluyendo “Sustentación”, “Empuje”, “ResistenciaalMovimiento” y “Peso”, aun cuando el “Empuje” sea probablemente 0, pero un paracaídas navegable puede depender de un dispositivo GPS (Global Positioning System; Sistema de posicionamiento global) que sabe acerca de la longitud y latitud y un altímetro que sabe acerca de la altitud (y de la velocidad y dirección del viento). También podríamos mostrar la relación idéntica usando el conector de realización para “IAptoparaVolar” y el de dependencia para “INavegable” (figura 6-9). Si está usted interesado en hacer hincapié en las relaciones, entonces puede usar paletas de caramelos; si quiere hacer hincapié en las operaciones, entonces el símbolo de clase con los estereotipos es una mejor selección.

Reglas para la herencia de interfaces

La idea básica que se encuentra detrás de las interfaces es que una interfaz describe una especificación de comportamiento, sin proporcionar los comportamientos, como la navegabilidad. En nuestro ejemplo del paracaídas, sólo estamos diciendo que nuestros paracaídas de alta velocidad que evitan las colisiones interactuarán con un dispositivo que actúa como una ayuda para la navegación, quizás incrementando la resistencia al movimiento. La presencia de la interfaz no prescribe cuál es el dispositivo; sólo impone los comportamientos que soporta ese dispositivo.

SUGERENCIA *El uso de un adjetivo —por ejemplo, atributo se convierte en atributivo— para los nombres de interfaces es una práctica común. A veces un diccionario resulta útil.*

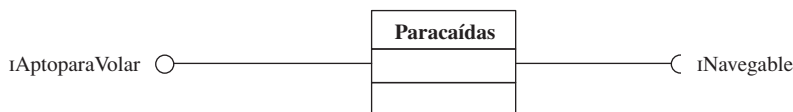


Figura 6-8 “IAptoparaVolar” es una interfaz proporcionada por “Paracaídas”, e “INavegable” muestra una interfaz requerida por “Paracaídas”.

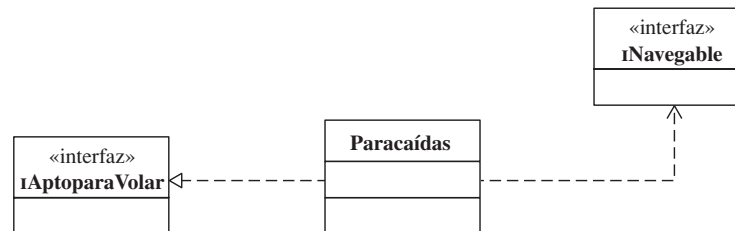


Figura 6-9 En esta figura, se ilustran las mismas relaciones que las descritas en la 6-8; específicamente, que “Paracaídas” realiza “iAptoparaVolar” y depende de “iNavegable”.

Las interfaces no proporcionan comportamientos; sólo estipulan cuáles deben ser. La regla es que debe implementarse una interfaz mediante la realización o herencia. Esto significa que

- Dada la interfaz A, la clase B puede implementar todos los comportamientos descritos por esa interfaz.
- Dadas las interfaces A y B, la cual hereda de la A, la clase C puede implementar todos los comportamientos descritos por aquéllas.
- Dada la interfaz A y las clases B y C, en donde la clase C hereda de la B, o ésta se compone de la C, las clases B y C juntas implementan todos los comportamientos descritos por la interfaz A. En el escenario de composición, B realiza A, y en el escenario de herencia, C realiza A.

Suponiendo que la especificación de comportamiento “iNavegable” incluyera “TenerLongitud()”, “TenerLatitud()” y “TenerAltitud()”, entonces, en el primer escenario, “iNavegable” podría realizarse por medio de un dispositivo que pudiera determinar la longitud, la latitud y la altitud. En el segundo escenario, “iNavegable” podría heredar de una interfaz “iDeAltitud”, y las dos interfaces se realizarían por un solo dispositivo de orientación tridimensional. Por último, en el tercer escenario, “iNavegable” podría definir las tres posiciones tridimensionales e implementarse por generalización o composición, como se muestra en la figura 6-10. (Sólo para satisfacer mi curiosidad, en realidad existe un dispositivo de ese tipo —el Garmin eTrex Summit GPS con brújula electrónica y altímetro. Yo quiero uno.)

Una vez más, vale la pena hacer notar que los tres escenarios descritos satisfacen de manera adecuada el requisito de navegabilidad. El escenario real que diseño depende de las clases de las que dispongo o de qué es conveniente. Si ya tengo parte de la interfaz realizada por otra clase, entonces podría obtener el resto a través de herencia de composición. Recuerde que el diseño no necesita ser perfecto, pero los modelos deben describir de modo adecuado lo que usted quiere dar a entender. Siempre puede cambiar su modo de pensar, si debe hacerlo.

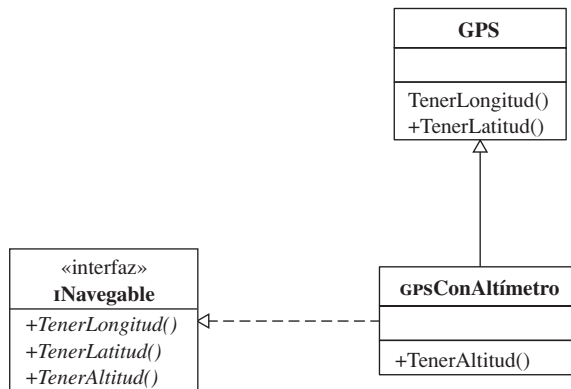


Figura 6-10 Implementación de una interfaz a través de una herencia.

Descripción de la agregación y la composición

La *agregación* logra una mención aquí porque es el término que se usa más a menudo en el diseño de software orientado a objetos cuando se habla acerca de la composición, es decir, cuando se habla acerca de una clase que está compuesta por otras. Se usa el término *agregación*, pero se quiere dar a entender *composición*. Como dije con anterioridad, el conector agregación —compuesto por un diamante hueco y una línea continua— tiene un significado ambiguo que no es diferente de una asociación, y se prefiere asociación.

En la composición se usa un diamante relleno y una línea continua. Cuando use la composición, significa que la clase que representa la totalidad, o clase compuesta, contiene aquél y sólo el caso de la clase que representa la parte; también significa que la clase totalidad es responsable de la duración de la clase parte.

La composición significa que la clase compuesta debe garantizar que se crean todas sus partes y se fijan a la compuesta, antes de que ésta esté por completo construida. En tanto exista la clase compuesta, se puede implementar el confiar que ninguna de sus partes sea destruida por cualquier otra entidad. Cuando se destruye la compuesta, debe destruir las partes, o puede eliminar en forma explícita las partes y llevarlas hacia algún otro objeto. La multiplicidad de la compuesta siempre es 1 o 0.1.

Para demostrar la composición, podemos modificar la relación que se ilustra en la figura 6-10. En este figura, demostré cómo satisfacer una interfaz a través de herencia, pero el nombre de la clase hijo “gpsConAltímetro”, suena como una relación de composición. La palabra *con* me sugiere composición más que herencia. Para satisfacer la interfaz, podemos definir “gpsConAltímetro” como la compuesta, definir “Altímetro” como la parte y promover el método “TenerAltitud” desde “Altímetro”. En la figura 6-11, se muestra la revisión, y la siguiente lista muestra cómo podríamos fragmentar cada uno de estos elementos en C#.

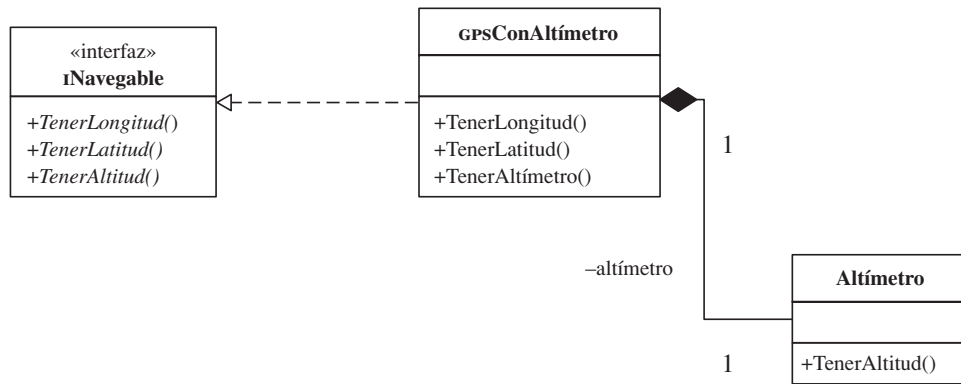


Figura 6-11 Figura 6-10 revisada para usar la composición con el fin de agregar el comportamiento del altímetro.

```

public interfaz INavegable
{
    double TenerLongitud();
    double TenerLatitud();
    double TenerAltitud();
}

public class Altímetro
{
    /// <summary>
    /// Return meters MSL (mean sea level)
    /// </summary>
    /// <returns></returns>
    public double TenerAltitud()
    {
        return 0;
    }
}

public class GPSConAltímetro : INavegable
{
    private Altímetro altimeter;

    public GPSConAltímetro()
    {
        altimeter = new Altímetro();
    }
}

```

```
#region INavegable Members

public double TenerLongitud()
{
    return 0;
}

public double TenerLatitud()
{
    return 0;
}

public double TenerAltitud()
{
    return altimeter.TenerAltitud();
}

#endregion
}
```

En esta lista, podemos ver que “GPSConAltímetro” contiene un campo privado “altímetro”. El constructor crea un caso del altímetro, y “TenerAltitud” usa ese altímetro para retornar la altitud. Debido a que C# es un lenguaje de “basura recolectada”, no necesitamos mostrar un destructor de manera explícita que libere el caso de la parte altímetro. (Ahora todo lo que queda por hacer es implementar los comportamientos.)

Asociaciones y las clases asociaciones

En el capítulo 5, se introdujo la asociación. Tomemos un momento para recapitular y, enseguida, introduciré algunos conceptos avanzados relativos a las asociaciones.

Cuando vea un campo en una clase, ésta es una asociación. Sin embargo, a menudo una asociación en un diagrama de clases se limita a las clases, en lugar de a tipos simples. Por ejemplo, un arreglo de valores cardinales se podría mostrar como un campo de arreglo o como una asociación hacia el tipo cardinal, con una multiplicidad de 1 en el extremo que representa la clase que contiene el arreglo, y una multiplicidad de muchos (*) en el tipo cardinal. Además, los campos y las asociaciones soportan navegabilidad, posibilidad de cambiarse y ordenamiento. Ese mismo arreglo de tipos cardinales se podría representar fijando la flecha de palillos conectada al tipo cardinal. Si quisiéramos indicar que el arreglo fuera de sólo lectura —quizás después de la inicialización— entonces colocaríamos el modificador {sólo lectura} en el campo y en la asociación. El significado es el mismo.



Figura 6-12 Podemos agregar modificadores y detalles a las asociaciones precisamente como las agregaríamos a los campos.

Si el arreglo estuviera ordenado, entonces podríamos colocar el modificador { ordenado} en el campo del arreglo o en la asociación. En la figura 6-12, se muestra nuestro arreglo de valores cardinales representado con el uso de una asociación directa de valores cardinales ordenados (clasificados).

Si una asociación tiene características, entonces podemos usar una clase asociación. Piense en una clase asociación como una tabla de vinculación en una base de datos en forma de relación, pero es una tabla de vinculación con comportamientos. Por ejemplo, podemos indicar que un “Patrón” está asociado con sus “Empleados”. Si quisiéramos indicar que “Empleados” es una colección que se puede ordenar, entonces podemos agregar una clase asociación llamada “ListadeEmpleados” y mostrar el método “Clasificar” en esa clase (figura 6-13).

En nuestro ejemplo, podemos elegir el uso de una asociación para reflejar que los patrones y los empleados están asociados, en lugar de que un patrón es una clase compuesta formada por empleados. Esto también funciona muy bien porque muchas personas tienen más de un patrón.

Una clase asociación tiene un conector de asociación fijo a una asociación entre las clases que vincula. En el ejemplo, la clase “Patrón” tendría un campo cuyo tipo es “ListadeEmpleados”, y éste tiene un método “Clasificar” y está asociado con los objetos “Empleados” (o los contiene). Si dejamos la clase de vinculación “ListadeEmpleados” fuera del modelo y todavía mantuviéramos la relación uno a muchos, entonces se supondría que existe alguna suerte de colección, pero el programador tendría la libertad de idear esta relación. La clase de vinculación aclara la relación con mayor precisión.

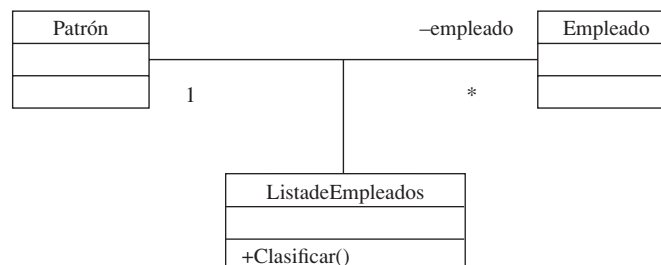


Figura 6-13 Una clase asociación que muestra que la clase “ListadeEmpleados” vincula en forma indirecta “Patrón” con “Empleados”.

También podríamos modelar la relación usando por asociación el “Patrón” a la “ListadeEmpleados” y la “ListadeEmpleados” al “Empleado”. El diagrama de la figura 6-14 muestra que, básicamente, las tres variaciones son lo mismo.

La parte superior de la figura 6-14 implica un arreglo o colección y, con instrucciones sencillas, como “Usar una colección con tipo de los objetos Empleado”, suele ser suficiente para realizar una implementación adecuada. Los dos diagramas de abajo en la figura 6-14 proporcionan un poco más de información e indican propiedad del comportamiento de clasificación, pero la implementación de cualquiera de las tres figuras debe ser casi idéntica.

Suponga además que elegimos mostrar cómo se tuvo acceso a un empleado específico de la colección por medio de un tipo, quizás un número de identificación del empleado. Por ejemplo, dado un número de identificación del empleado, podríamos indicar que uno de esos números conduce a un empleado único. Esto se conoce como *asociación calificada* y se puede modelar agregando la clase del calificador, como se muestra en la figura 6-15.

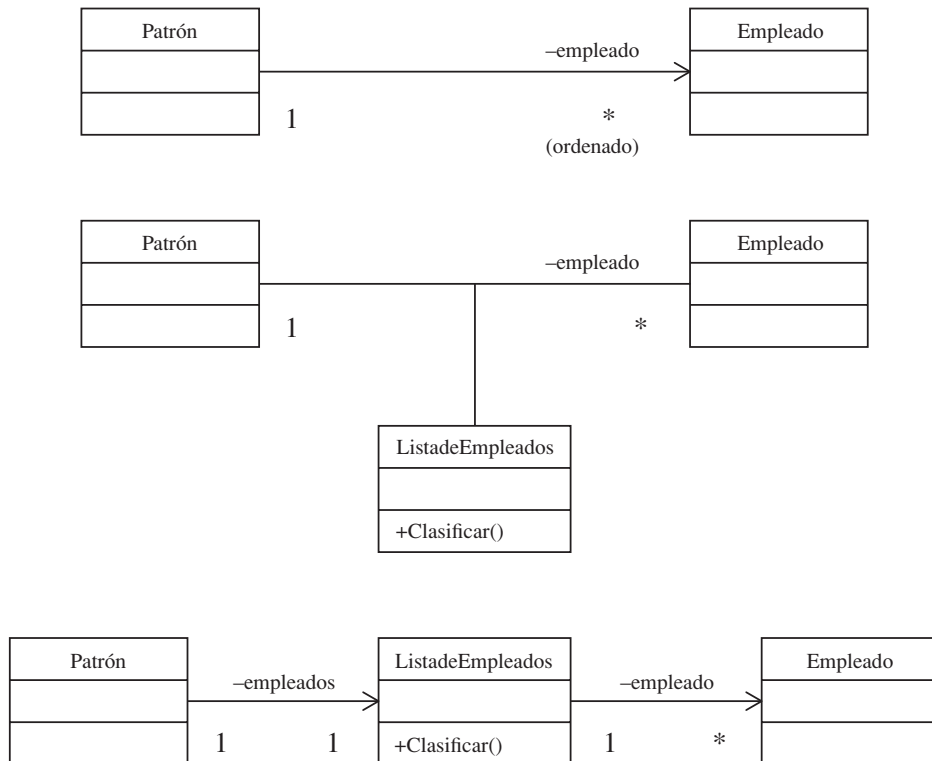
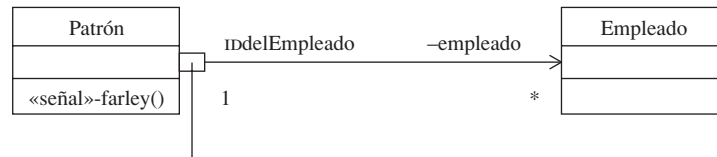


Figura 6-14 Tres variaciones que reflejan una relación uno a muchos entre un “Patrón” y “Empleados”.



El calificador es IDdelEmpleado

Figura 6-15 El calificador que da como resultado un empleado único es el “IDdelEmpleado”.

Cuando vea un calificador, esperará verlo usado como un parámetro que da como resultado un caso específico del tipo asociado. La siguiente lista de código muestra cómo podemos implementar ese código en Visual Basic.NET usando una colección con tipo de objetos “Empleado”, una clase nombrada “IDdelEmpleado” y un indexador.

```

Imports System.Collections

Public Class Patrón
    Private empleados As ListadeEmpleados

    Public Sub New()
        empleados = New ListadeEmpleados
    End Sub
End Class

Public Class ListadeEmpleados
    Inherits System.Collections.CollectionBase

    Default Public Property Item(ByVal id As IDdelEmpleado) As Empleado
        Get
            Return GetEmpleado(id)
        End Get
        Set(ByVal Value As Empleado)
            SetEmpleado(Value, id)
        End Set
    End Property

    Public Function Add(ByVal value As ListadeEmpleados) As Integer
        Return List.Add(value)
    End Function

    Private Function Indexof(ByVal value As IDdelEmpleado) As Integer
        Dim i As Integer
        For i = 1 To List.Count
            If (CType(List(i), Empleado).ID.IsEqual(value)) Then
                Return i
            End If
        Next
    End Function
End Class
  
```

```

        Throw New IndexOutOfRangeException("id not found")
    End Function

    Private Function GetEmpleado(ByVal id As IDdelEmpleado) As Empleado
        Return List(Indexof(id))
    End Function

    Private Sub SetEmpleado(ByVal value As Empleado, _
        ByVal id As IDdelEmpleado)
        List(Indexof(id)) = value
    End Sub
End Class

Public Class Empleado
    Private FName As String
    Private FID As IDdelEmpleado

    Public Property Name() As String
        Get
            Return FName
        End Get
        Set(ByVal Value As String)
            FName = Value
        End Set
    End Property

    Public Property ID() As IDdelEmpleado
        Get
            Return FID
        End Get
        Set(ByVal Value As IDdelEmpleado)
            FID = Value
        End Set
    End Property
End Class

Public Class IDdelEmpleado
    Private ssn As String

    Public Sub New(ByVal value As String)
        ssn = value
    End Sub

    Public Function IsEqual(ByVal value As IDdelEmpleado) As Boolean
        Return value.ssn.ToUpper() = ssn.ToUpper()
    End Function
End Class

```

Incluso si no está familiarizado con Visual Basic.NET, puede observar los encabezados de las clases y ver todas las clases que se muestran en la figura 6-15 (la cual incluye

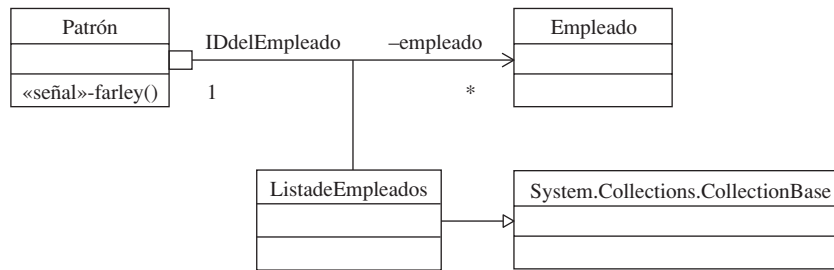


Figura 6-16 En el diagrama revisado se usa una clase asociación para introducir la generalización que muestra que “ListadeEmpleados” hereda de “System.Collections.CollectionBase”.

“Patrón”, “Empleado”, “IDdelEmpleado” y la implicada “ListadeEmpleados”). [El hecho de que “ListadeEmpleados” herede de “System.Collections.CollectionBase” es conocimiento especializado que se requiere en cualquier lenguaje o marco de referencia particular. El lector tiene la opción de mostrar la generalización de “CollectionBase” por “ListadeEmpleados”, lo cual podría usted agregar al diagrama (figura 6-16), si sus desarrolladores necesitaran que se les lleve un poco más de la mano.]

El factor de decisión que me ayuda a elegir cuánto detalle agregar es mi audiencia de programadores. Si mis compañeros programadores son muy experimentados en el lenguaje y marco de referencia de implementación que se seleccionen, entonces podría dejar fuera detalles acerca de cómo implementar la colección de empleados. Para los programadores nuevos, puede resultar de ayuda mostrar la información agregada en la figura 6-16. En la práctica, con programadores muy nuevos, suelo agregar más detalles y, a continuación, codificar un ejemplar que les muestre cómo implementar la construcción, en este caso una colección con tipo específica para Visual Basic.NET.

NOTA Incluso los diagramas UML detallados no siempre resultan claros para todos. Por esta razón, suele ser un detalle importante que los modeladores sepan cómo implementar los diagramas que crean en la plataforma objetivo que se elija o, por lo menos, que una persona del equipo pueda traducir a código los aspectos avanzados de los diagramas UML.

Examen de las relaciones de dependencia

Una *dependencia* es una relación de cliente y proveedor, también conocidos como *fuentes* y *objetivos*. La relación de dependencia es una línea punteada con una flecha de palillos en el extremo. La flecha se fija al proveedor, también llamado *objetivo*. Yo prefiero los términos *fuentes* y *objetivos*, ya que objetivo facilita recordar hacia cuál de los extremos apunta la flecha.

Una dependencia en un diagrama de clases significa que la fuente depende del objetivo de alguna manera. Si el objetivo cambia, entonces la fuente resulta afectada. Esto significa

que si cambia la interfaz del blanco, entonces resultará afectada la implementación de la fuente. Las dependencias no son transitivas. Por ejemplo, si una clase A depende de una clase B y esta última depende de la clase C, entonces si cambia la interfaz de la C, puede ser que tenga que cambiarse la implementación de la B, pero no necesariamente la interfaz de ésta. No obstante, si las dependencias son cíclicas —la clase A depende de la B, la cual depende de la C, la cual, a su vez, depende de la A— entonces los cambios a la clase C pueden tener un efecto cíclico que produzca cambios muy difíciles, lo que conduce a una implementación frágil. Como regla general, evite las dependencias complicadas y cíclicas.

Las asociaciones dirigidas, la composición y la herencia implican una dependencia. Si la clase A tiene una asociación dirigida con la B, entonces la clase A depende de la B. Si la clase B hereda de la A, entonces la B depende de la A. La asociación y la generalización son relaciones más precisas con sus propias connotaciones; use la dependencia cuando no es aplicable uno de los tipos más específicos de relaciones.

Por último, antes de que examinemos algo de los estereotipos predefinidos que se aplican a las dependencias, no trate de mostrar todas las relaciones de dependencia; sólo trace las dependencias que son importantes.

En la tabla 6-1, se muestran los estereotipos predefinidos para las dependencias. Con frecuencia, la implicación de una dependencia resulta clara por su contexto, pero estos

acceder	Referencia privada hacia otro contenido del paquete.
ligar	Describe un nuevo elemento que se crea cuando se asigna el parámetro de la plantilla.
llamar	Un método en la fuente llama a un método en el objetivo.
crear	La fuente crea un ejemplo del objetivo.
derivar	Se deriva un objeto de otro.
ejemplificar	La fuente crea un ejemplo del objetivo.
permitir	La fuente puede tener acceso a los miembros privados del objetivo (por ejemplo, implementada como una relación de amigo en algunos lenguajes).
realizar	La fuente implementa la interfaz del objetivo. (El conector de realización es una mejor selección.)
refinar	La fuente refina el objetivo. Esto se usa para tener la posibilidad de rastreo entre los modelos (por ejemplo, entre un modelo de análisis y uno de diseño).
enviar	Indica un emisor y un receptor de una señal.
sustituir	Se puede sustituir el blanco por la fuente. (Esto es semejante a cómo una subclase puede ser sustituida por su superclase.)
rastrear	Usado para vincular elementos del modelo.
usar	La fuente necesita el blanco para completar su implementación.

Tabla 6-1 Lista de estereotipos para las relaciones de dependencia definidas por el UML Versión 2.0.

estereotipos existen para que exprese con claridad el uso que usted pretende. (Después de la tabla hay una breve descripción de cada una de las relaciones de dependencia.)

A menudo, basta con trazar el conector de dependencia ocasional en el código e implementar lo que quiere usted dar a entender. Los siguientes párrafos se extienden un poco sobre las relaciones de dependencia descritas en la tabla 6-1.

La dependencia “acceder” soporta la importación de paquetes en forma privada. Algunos de estos conceptos son nuevos en el UML versión 2.0, y éste es uno que no he tenido ocasión de usar. El ejemplo más cercano que se podría aplicar aquí es la diferencia entre las cláusulas de uso de interfaz y de implementación en Delphi. En esencia, Delphi soporta importación privada en sus cláusulas de uso de la implementación.

Si alguna vez ha leído *The C++ Programming Language*, escrito por Bjarne Stroustrup, entonces habrá leído el discurso sobre las clases plantilla. En C con clases, las plantillas se originaron como una construcción semanal con tipo ideada usando la sustitución y macros. El resultado fue que el nuevo nombre creado por la concatenación del tipo cadena condujo a una nueva clase. Con las plantillas, el resultado es el mismo. Cuando define el parámetro para los tipos parametrizados —plantillas o genéricos— tiene una nueva entidad. “Ligar” existe con el propósito de modelar este caso.

“Llamar” llama de manera directa un método de la clase objetivo. “Crear” indica que la fuente crea un ejemplo del objetivo. El lector podría ver esta relación en conjunción con el patrón fábrica. El único propósito de una fábrica es realizar todos los pasos necesarios para crear el objeto correcto.

“Derivar”, “realizar”, “refinar” y “rastrear” son dependencias abstractas; existen para representar dos versiones de la misma cosa. Por ejemplo, la dependencia “realizar” implica la misma relación que una realización; es decir, la implementación de una interfaz. “Rastrear” se usa para conectar elementos del modelo conforme evolucionan; por ejemplo, usar casos para las realizaciones de casos de uso.

“Ejemplificar” también se podría usar para indicar que la fuente crea ejemplos del objetivo. Un ejemplo mejor se relaciona con la información del tipo en el tiempo de ejecución o la reflexión en .NET. Podríamos mostrar que se usa una metaclasses (o el ejemplo del objeto “Tipo” en .NET) para crear un ejemplo de una clase.

El estereotipo “permitir” se usa para indicar que la fuente puede invocar miembros no públicos del objetivo. Esta relación la soporta el modificador “Friend” (“Amigo”) en lenguajes como Visual Basic y a través de reflexión dinámica.

Una *señal* es como un evento que ocurre fuera de secuencia. Por ejemplo, cuando usted está dormido y la alarma empieza a sonar, ésta es una señal para despertar. El estereotipo “señal” se usa para indicar que ha sucedido algo que necesita una respuesta. Piense en evento.

El estereotipo “sustituir” se aplica cuando la fuente se puede sustituir con el blanco. La forma más clara de sustitución es una clase hijo en lugar de una clase padre. Por último, el estereotipo “usar” es común. “Usar” sencillamente implica que la fuente necesita que se complete el objetivo. “Usar” es una forma más generalizada de “llamar”, “crear”, “ejemplificar” y “enviar”.

Adición de detalles a las clases

Como se dice, el mal se encuentra en los detalles. Los diagramas de clases pueden incluir una gran cantidad de información que se transmite por medio de caracteres de texto, fuentes y qué es lo que se incluye, así como qué se excluye. Yo prefiero ser explícito hasta el punto que sea posible, pero no verboso, y estar presente en persona para resolver las ambigüedades en el transcurso de la implementación. En esta sección, quiero señalar unos cuantos detalles que el lector puede buscar y algunos atajos respetables que puede tomar para asegurarse de que entiende los diagramas UML creados por otros y que los otros entienden los diagramas de usted. Debido a que estas directrices básicas son más o menos cortas, se encuentran en una lista como proposiciones.

- Las características subrayadas indican características estáticas.
- Las propiedades derivadas se demarcan por medio de una diagonal antes del nombre de la propiedad. Por ejemplo, dadas las propiedades “horas trabajadas” y “salario por hora”, podemos derivar el salario, el cual aparecería como “/salario”.
- Los nombres de clases en cursivas indican clases abstractas. Una clase abstracta tiene algunos elementos sin implementación y depende de subclasses para una implementación completa.
- Campos del modelo; las propiedades se implican en lenguajes que soportan propiedades. En lenguajes que no soportan propiedades, los métodos con los prefijos “get_” y “set_” conducen al mismo resultado.
- Las restricciones especifican condiciones anteriores (pre) y posteriores (post). Use las restricciones para indicar el estado en el cual debe estar un objeto cuando se introduce un método y se hace salir otro. La construcción “aserción” soporta este estilo de programación.
- Cuando está modelando operaciones, trate de mantener un número mínimo de operaciones públicas, use campos privados y permita el acceso a los campos a través de propiedades (si se soportan) o de métodos de acceso (si no se soportan las propiedades).

Examen

1. Una subclase tiene acceso a los miembros privados de una superclase.
 - a. Verdadero
 - b. Falso

2. Si una clase hijo tiene más de una clase padre y cada padre introduce una operación con el mismo nombre,
 - a. el programador debe resolver el conflicto de nombre en forma explícita.
 - b. todos los lenguajes que soportan herencia múltiple resuelven los conflictos de manera implícita.
 - c. Ninguna de las anteriores. No se permiten los conflictos.
3. ¿Cuál(es) de las proposiciones siguientes es (son) verdadera(s)?
 - a. La generalización se refiere a subtipos.
 - b. La clasificación se refiere a subtipos.
 - c. La generalización se refiere a ejemplos de objetos.
 - d. La clasificación se refiere a ejemplos de objetos.
 - e. Ninguna de las anteriores
4. Realizar
 - a. significa heredar de una clase padre.
 - b. significa implementar una interfaz.
 - c. significa promover los miembros constituyentes en una clase compuesta.
 - d. es un sinónimo de agregación.
5. Si un lenguaje no soporta herencia múltiple, entonces se puede tener una aproximación del resultado por medio de
 - a. una asociación y la promoción de propiedades constituyentes.
 - b. realización.
 - c. composición y la promoción de propiedades constituyentes.
 - d. agregación y la promoción de propiedades constituyentes.
6. La clasificación dinámica —en donde un objeto se cambia en el tiempo de ejecución— se puede modelar usando
 - a. generalización.
 - b. asociación.
 - c. realización.
 - d. composición.

7. Una clase “asociación” se menciona como una clase de vinculación.
 - a. Verdadero
 - b. Falso
8. Un calificador de “asociación”
 - a. se usa como una precondición a una asociación.
 - b. representa el papel de un parámetro usado para retornar un objeto único.
 - c. se usa como una precondición posterior a una asociación.
 - d. es lo mismo que una asociación dirigida.
9. Seleccione las proposiciones correctas.
 - a. Una interfaz proporcionada significa que una clase implementa una interfaz.
 - b. Una interfaz requerida significa que una clase depende de una interfaz.
 - c. Una interfaz proporcionada significa que una clase depende de una interfaz.
 - d. Una interfaz requerida significa que una clase implementa una interfaz.
10. Cuando un símbolo de clasificador se encuentra en cursivas,
 - a. significa que el símbolo representa un objeto.
 - b. significa que el símbolo representa una clase abstracta.
 - c. significa que el símbolo representa una interfaz.
 - d. significa que el símbolo es un valor derivado.

Respuestas

1. b
2. a
3. a y d
4. b
5. c
6. b
7. a
8. b
9. a y b
10. b

Uso de los diagramas de esquemas de estado

Históricamente, la diferencia entre los esquemas de estado y los diagramas de actividades ha sido un embrollo. En el Unified Modeling Language (UML) versión 2.0, los esquemas de estados entran en posesión de la suya como un diagrama distinto y separado.

Los *esquemas de estados* (también conocidos como *máquinas de estado*) son buenos para mostrar el estado de un objeto sobre muchos casos de uso y para definir protocolos que describen una orquestación correcta de los mensajes, tal y como se podría necesitar para tener acceso a las bases de datos o para conectividad por el Transmission Control Protocol (TCP; Protocolo de Control de la Transmisión). De manera ideal, los esquemas de estado son adecuados para describir el comportamiento de las interfaces de los usuarios y de los controladores de dispositivos para sistemas de tiempo real. En tanto que los diagramas de interacción son buenos para la comprensión de los sistemas, los esquemas de estados son buenos para la indicación precisa del comportamiento. Si usted está trabajando en sistemas de tiempo real o con controladores de dispositivos físicos, entonces puede usar con frecuencia los esquemas de estados. Sin embargo, un número enorme de aplicaciones son

empresariales, basadas en interfaces gráficas del usuario (y las bases de datos, así como muchos programadores, usan herramientas modernas de desarrollo rápido de aplicaciones para crear prototipos de interfaces, en lugar de definir sus comportamientos usando esquemas de estados). [No estoy juzgando acerca de si los prototipos deben crearse sin los esquemas de estados, pero la creación de prototipos de interfaces gráficas del usuario (GUI) no es parte del UML.]

Parte de la desmitificación del UML es asegurarse de que usted sabe que no necesita usar todos los elementos del modelo, crear toda suerte de diagramas o modelar todos los aspectos de un sistema. Adhiérase a la modelación de elementos que sean complicados y en donde el examen del modelo puede conducir a una mejor solución. Por ejemplo, si está usando un marco de referencia bien comprendido como ADO.NET, es innecesario crear diagramas de protocolos que muestren cómo abrir una conexión, leer datos y cerrarla. Estos procesos están prescritos por el marco de referencia, y el tiempo que se consuma en la creación del modelo podría usarse mejor en otra parte. Dicho esto, en ocasiones querrá o necesitará esquemas de estados; en este capítulo le mostraré los elementos de estos esquemas y algunos ejemplos. El lector aprenderá

- Acerca de los elementos que se usan para crear esquemas de estados
- Cómo crear esquemas de estados
- La diferencia entre los esquemas de estados de comportamiento y de protocolo
- Formas comunes de implementar esquemas de estados

Elementos de un diagrama de estado

Lo más sencillo acerca del UML es que la mayoría de los diagramas se componen de símbolos y líneas sencillos. Esto se cumple en los esquemas de estados, los cuales se componen de manera significativa de símbolos llamados *estados* y líneas llamadas *transiciones*. La sencillez de los símbolos es la parte más fácil del modelado; la identificación de los problemas, la captación de las soluciones y la captura de esta comprensión son los aspectos del modelado con UML que pueden hacer que modelar sea tan complejo como programar.

Tres cosas para recordar son

- Conocer todos los símbolos y la gramática no implica que deba usarlos todos.
- Es esencial modelar los aspectos importantes del sistema, así como modelar aquellos que no son obvios.
- No necesita toda suerte de diagrama para toda suerte de problema; sea selectivo.

Dicho esto, amplíemos nuestro conocimiento del UML y consideremos los diversos símbolos para los esquemas de estados que evolucionaron a partir de su relación entremezclada con los diagramas de actividades.

Examen de los símbolos de estado

Existen varios símbolos de estados; el más común es el rectángulo con esquinas redondeadas o estado simple. De manera significativa, los esquemas de estados constan de estados y transiciones simples, pero hay otros estados que representan papeles importantes aunque menos prominentes.

En esta sección, explicaré los estados simples con actividades comunes y para hacer; estados compuestos ortogonales y no ortogonales; estados inicial, de terminación y final; las conexiones, las selecciones y los estados de historia; los estados submáquinas y los superestados, así como los puntos de entrada y de salida.

Uso de los estados inicial, final y de terminación

Recuerde que los esquemas de estados y los diagramas de actividades tienen una historia compartida; como consecuencia, aun cuando en el UML versión 2.0 sus definiciones están delineadas con mayor claridad, los esquemas de estados y los diagramas de actividades todavía tienen algunos símbolos en común. Tres estados—inicial, final y terminal—usan los mismos símbolos que se encuentran en los diagramas de actividades, pero desempeñan papeles adecuados para los esquemas de estados.

El estado inicial es un círculo relleno que representa un pseudoestado en las máquinas de estados de protocolo—vea “Creación de máquinas de estado de protocolo” más adelante—en el UML versión 2.0. Puede usar el estado inicial en los esquemas de estados en general, pero no es de uso común. En el estado final se usa el mismo símbolo—un círculo relleno con un contorno circular—como actividad final en los diagramas de actividades y se usa para indicar el fin de un esquema de estados; vea “Creación de máquinas de estado de comportamiento” más adelante. Los estados finales no tienen transiciones salientes; no tienen actividades de entrada, salida o para hacer; no hacen referencia a submáquinas, y no están divididos en regiones. (Estos conceptos se describen abajo.) El estado final es un punto extremo sin elaboración. El estado de terminación es una X usada en las máquinas de estados de protocolos; piense en él como en un extremo muerto.

Uso de la conexión y los estados de selección

Un estado de selección es un pseudoestado que se usa en las máquinas de estado de protocolos. Una selección se parece a un diamante de decisión y desempeña un papel semejante al de una decisión en los diagramas de actividades. Una selección tiene una sola transición entrante y más de una saliente. Las transiciones salientes se toman dependiendo de cuál condición guardián evalúa lo que es verdadero. Si más de un guardián evalúa lo que es verdadero, entonces se toma una transición arbitraria, pero por lo menos un guardián debe evaluar lo verdadero (figura 7-1).

Una conexión es un círculo relleno, como el del estado inicial, y se usa para combinar varias transiciones entrantes en una sola saliente, o para dividir una sola transición entrante en múltiples transiciones salientes (figura 7-2).

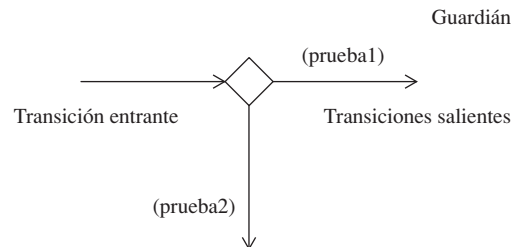


Figura 7-1 Estado de selección en el que se muestran una sola transición entrante y dos salientes, cada una con una condición guardián.

El mayor problema al usar notaciones y símbolos de estilo antiguo es que si usted trata de generar código, es posible que la herramienta le informe de un error. Sin embargo, el estado de los generadores de código todavía es incierto, y cada herramienta tiene algunas limitaciones relativas a la especificación formal del UML.

En virtud de la historia compartida del esquema de estados con los diagramas de actividad, el lector podría ver conexiones modeladas usando los símbolos de bifurcación y de unión que se emplean en los diagramas de actividad. Tanto la bifurcación/unión como la conexión con sus transiciones entrantes y salientes indican con claridad el intento de transiciones que se dividen o combinan.

Uso de los estados de historia superficial y profunda

Una historia superficial se indica por medio de un círculo con una H, y una profunda se indica por medio de un círculo con una H*. Las historias se usan en las máquinas de estados de protocolos. Se usa una historia superficial para representar un subestado reciente para un estado compuesto, y se usa una profunda para representar una historia recursiva de subestados. (Para obtener más información, consulte la siguiente sección sobre los estados compuestos.)

Si una máquina de estados realiza una transición hacia un estado de historia, entonces se activa y ejecuta el estado más reciente. Piense en los estados de historia como en un medio de modelar deshacer, rehacer o hacer pausa en los comportamientos y reanudarlos.

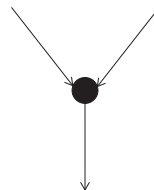


Figura 7-2 Conexión en la que se muestran múltiples transiciones entrantes con una sola saliente.

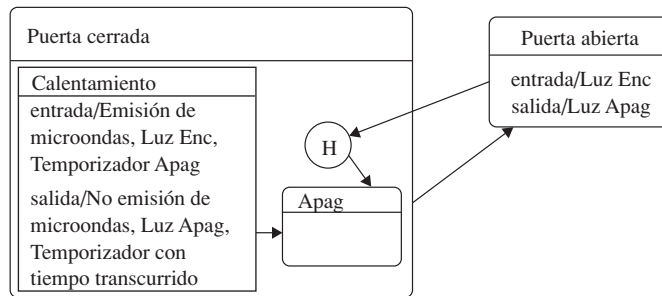


Figura 7-3 Estado compuesto en el que se muestra una historia superficial —círculo H— la cual indica que el estado del microondas se almacena cuando la puerta se abre.

En la figura 7-3, se muestra un estado compuesto —vea “Comparación de los estados simples y compuestos” más adelante— que representa un horno de microondas. Cuando la puerta está cerrada, podríamos estar calentando, o el horno sólo podría estar apagado (off). Cuando estamos calentando, un temporizador, una luz y el emisor de microondas están encendidos (on); cuando salimos del modo de calentamiento, el temporizador, la luz y el emisor se apagan. Si la puerta se abre, entonces la luz se enciende y se almacena una historia antes de la transición al estado de APAGADO. Se pretende que la historia permita reanudar, en el punto del tiempo transcurrido en el temporizador, si arrancamos de nuevo el horno.

En la primera parte del siglo pasado, se descubrió que las microondas podían rebotar de los objetos y se usaron para detectar la dirección y el alcance. Se intentó que la aplicación original fuera para detectar los *messerschmitts* alemanes durante la Segunda Guerra Mundial. El doctor Percy Spencer, en Raytheon, descubrió de manera accidental que el emisor de microondas fundió algo de chocolate que tenía en su bolsillo. A continuación, Spencer probó con algunas semillas de maíz para producir “palomitas” en una bolsa de papel, y se descubrió el horno de microondas. Debido a su aplicación original como radar, al horno se le llamó “estufa de radar” y, finalmente, el nombre se cambió por *horno de microondas*. La primera estufa de radar tenía 6 pies (1.80 m) de altura y costó 5,000 dólares.

Uso de las actividades de estado

Los estados son activos o inactivos. Un estado se vuelve *activo* cuando se ejecuta su actividad de entrada. Un estado se vuelve *inactivo* después de que se ejecuta su actividad de salida. (En la figura 7-3, el lector puede ver ejemplos de actividades de entrada y salida.) Se puede ver una buena demostración de una implementación de actividades de entrada y salida en eventos escritos para cuando un control aumenta foco y pierde foco. Por ejemplo, cuando abrimos la puerta de un refrigerador, se enciende una luz y, cuando cerramos la puerta, la luz se apaga.

Los estados pueden contener actividades adicionales. Éstas se dividen en categorías: comunes y de hacer. Una *actividad común* es algo que sucede de manera instantánea. Una

actividad con el prefijo “hacer/” se conoce como *actividad de hacer*. Las actividades de hacer suceden durante un tiempo. Por ejemplo, una actividad común se podría completar en unas cuantas instrucciones de máquina que no se pueden interrumpir, o quizás podría durar más, si ocurriera dentro de una sección crítica de camino. Una actividad de hacer sucede en el curso de muchas instrucciones y puede ser interrumpida, por ejemplo, por un evento.

Considere la aplicación Visual SourceSafe de la figura 7-4. Si hace clic en un nodo de alto nivel y elige la opción “Get Latest Version” (“Obtener la versión más reciente”), entonces podría estar esperando un tiempo, porque copiar cientos o miles de archivos de un almacén de código fuente, a través de una red, hasta una estación de trabajo toma tiempo. En forma concienzuda, esa operación de larga ejecución debe ser susceptible de interrumpirse. Con el uso de una simple actividad de hacer en un estado, se indica que ésta es una parte que se pretende del diseño.

Comparación de los estados simples y compuestos

Un *estado simple* es aquél sin subestructura. No tener subestructura significa que el estado no está dividido en regiones y que no hay subestados. Un *estado compuesto* (también llamado *superestado*) tiene una estructura interna que puede incluir regiones y sí incluye subestados. El estado “Puerta cerrada” de la figura 7-3 es compuesto; también es un estado *no ortogonal*.

Estado compuesto *no ortogonal* significa que hay subestados anidados y sólo uno está activo en un momento. Por ejemplo, en la figura 7-3, sólo “Calentamiento” o “Apag” está activo en un momento. Un estado compuesto ortogonal está dividido en regiones que se ejecutan en forma concurrente. En cada región, sólo un subestado está activo en un momento.

La figura 7-3 es un ejemplo de un estado compuesto no ortogonal. Para crear un estado compuesto ortogonal, divida el símbolo de estado en regiones y coloque subestados en sus regiones respectivas. La figura 7-5 muestra un estado compuesto ortogonal que representa el congelador de un refrigerador. El enfriamiento y la congelación suceden en forma concurrente en compartimientos separados, pero el encendido de la luz sucede cuando se abre cualquiera de las dos puertas.

SUGERENCIA *Visio no realiza un gran trabajo de administración de estados compuestos; soporta subestados compuestos agregando un esquema de estados hijo vinculado cuan-*

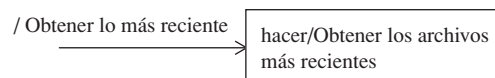


Figura 7-4 Estado con una “hacer/actividad Obtener la versión más reciente”; el “hacer/” significa que este estado se puede interrumpir.

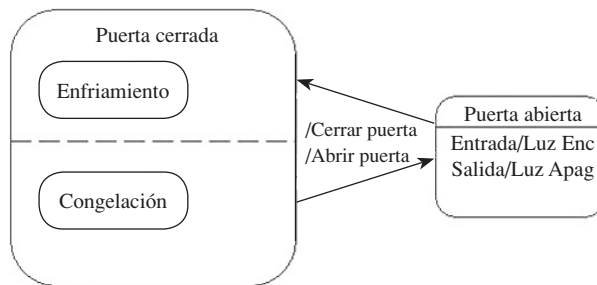


Figura 7-5 Estado compuesto ortogonal que representa enfriamiento y congelación simultáneos.

do usted agregue un estado compuesto a un diagrama. Las características avanzadas, como los subestados compuestos ortogonales, tienen soporte en las herramientas más avanzadas (y más caras). Vale la pena pagar el precio de admisión a ese tipo de herramientas, si usted usa con frecuencia características avanzadas que no encuentra en herramientas como Visio.

La figura 7-3 se creó laboriosamente usando MS-Paint, en tanto que la 7-5 se creó con mucha mayor rapidez usando Rational XDE. Yo realizo una gran cantidad de modelado, de modo que vale la pena el precio de admisión para usar este paquete, pero en algunos proyectos, he usado Visio y funciona bien para el modelado cotidiano. Las herramientas buenas son la marca de un buen artesano, pero gastar una gran cantidad de dinero no es garantía de éxito.

Uso de las actividades internas

Las actividades internas son como autotransiciones; consulte “Examen de las transiciones” más adelante. Una *actividad interna* es una respuesta que sucede en forma interna y que dispara una actividad sin la ejecución de una actividad de entrada o salida. En las actividades internas se usa el mismo evento, guardián y actividad que en las transiciones. Más adelante hablaré más acerca de las transiciones.

Vinculación con las submáquinas

En lugar de repetir los diagramas de esquemas de estados (máquina de estados), usted quiere volver a usar los diagramas. Esto se aplica a las máquinas de estados. El UML soporta el modelado de submáquinas mediante el nombramiento de la máquina de subestados después del nombre del estado, separada por un nombre de clase. (Esto se parece a la sentencia de declaración de nombre de clase variable en C++.) Por ejemplo,

```
mystate : MyStateMachine
```

indica que “MyState” (MiEstado) es un ejemplo de la máquina de estados nombrada “MyStateMachine”.

Si está usando Visio, entonces puede usar la notación de nombre, dos puntos y máquina de estados para hacer referencia a una máquina de subestados. Otras herramientas —como Rational XDE, mencionada con anterioridad— soportan un símbolo especial para las submáquinas y vinculará en forma dinámica la submáquina de referencia.

Examen de las transiciones

Las *transiciones* son líneas dirigidas que conectan estados. Las transiciones pueden ocurrir con base en algún mecanismo de disparo —por lo común, implementado como eventos— y pueden procesarse o no con base en una condición guardián, lo que da como resultado algún efecto. Esta suerte de relación de causa y efecto ilustra por qué las máquinas de estados pueden resultar útiles para modelar interfaces del usuario. En esta sección, se examinarán los mecanismos de disparo, ejemplos de condiciones guardianes y la manera de especificar los efectos. También completaremos la exposición acerca de las transiciones internas y externas introducidas en la sección “Uso de actividades internas”.

Especificación de los disparadores

Una transición tiene un estado fuente, un evento de transición, un guardián, un efecto y un estado objetivo. Antes de salir del estado fuente, ocurre la actividad de salida. Cuando ocurre el disparo de la transición, se puede realizar una prueba con una condición guardián para determinar si se toma la transición. Una transición tomada da como resultado un efecto. Por último, se ejecuta la actividad de entrada del objetivo. La línea dirigida que representa la transición se rotula con el evento opcional de disparo, el guardián y el efecto. Si finalizan las actividades en un estado, entonces el resultado se conoce como *transición sin disparo* o *de compleción*.

A los disparadores, o eventos, que significan una transición se les da la categoría de eventos de *llamada*, de *cambio*, de *señal* y *temporizador*. Un evento de llamada especifica una llamada síncrona de un objeto. Un evento de cambio representa un cambio en el resultado de una expresión booleana. Un evento de señal indica un mensaje explícito, nombrado síncrono, y un evento temporizador es un disparador que ocurre después de un intervalo específico de tiempo. El disparador es el primer elemento, si está presente, fijo a una transición.

SUGERENCIA Algunas herramientas pueden colocar prefijos a tipos específicos de transiciones, con etiquetas como “cuándo”, en el caso de Visio y eventos de cambio.

Especificación de las condiciones guardianes

Las condiciones guardianes se colocan entre corchetes y deben evaluarse para una condición booleana susceptible de probarse. (He visto la notación para las condiciones guardianes en otros diagramas, como los diagramas de actividad y de interacción.) Si está

presente una condición guardián, entonces se evalúa y debe conducir a un valor verdadero para que se complete la transición.

Las condiciones guardianes deben ser relativamente sencillas y no deben conducir a efectos secundarios. Por ejemplo “[$x > 0$]” es una buena condición guardián, pero “incrementar x durante la evaluación como [$x++ > 0$]” es un guardián con efectos secundarios porque se cambia el valor de x cada vez que se ejecuta el guardián.

NOTA El modelado formal evolucionó después de prácticas formales de codificación. Muchas buenas prácticas, como no escribir código condicional con efectos secundarios, las prácticas especulares deseables en el código y, en general, los modelos, finalizan como código.

Especificación de los efectos

Los disparadores, los guardianes y los efectos son opcionales. El último elemento de un símbolo de transición es el efecto opción (o actividad). El efecto es alguna actividad que se debe realizar cuando se dispara la transición. La signature de una transición, incluyendo un disparador, un guardián y el efecto, es

Event [Guard] / Effect

También podría ver eventos mencionados como *disparadores* y efectos a los que se les da el nombre de *actividades*. Aunque muchos sinónimos pueden ser confusos, estas palabras son suficientemente cercanas como para transmitir su finalidad.

Según la especificación formal, puede haber muchos disparadores, un guardián y una actividad. Soportar cero para muchos disparadores significa que más de un evento pue-

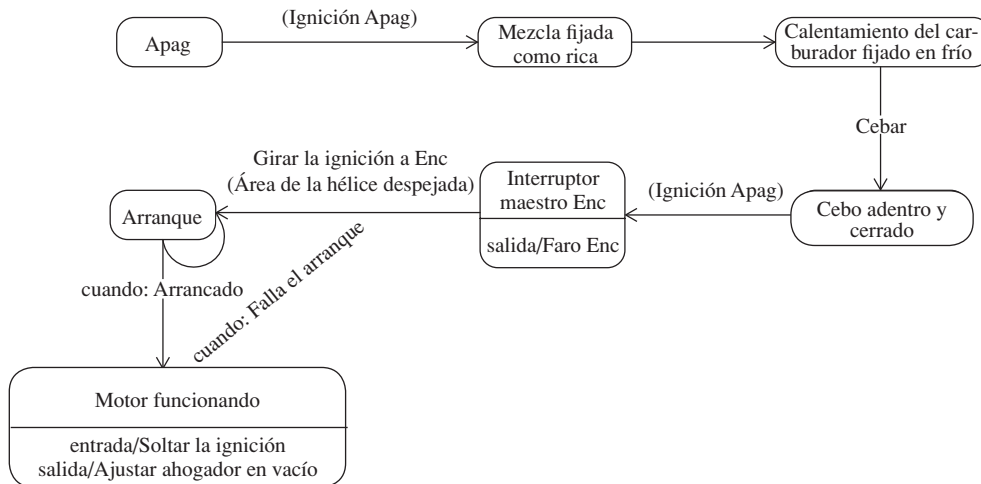


Figura 7-6 Diversas transiciones que muestran elementos opcionales.

de dar como resultado una transición. Soportar un solo guardián no significa que éste no pueda tener múltiples predicados (subexpresiones que conducen a un resultado booleano), y un solo efecto no significa que éste no puede ser un efecto compuesto. (Además, el estado objetivo también puede realizar muchas actividades.) En la figura 7-6, se muestran varias transiciones con algunos de los elementos descritos en esta sección, o todos ellos.

En la figura, estamos mostrando una máquina de estados que refleja el estado de un avión monomotor entre los estados de apagado y de marcha en vacío. La máquina de estados modela el motor como un sistema complejo con una progresión de transiciones y estados, siendo el estado final que el motor se encuentra funcionando y marchando en vacío.

NOTA *En un sistema digital, resulta fácil hacer que se ejecuten cosas como “la ignición debe estar en posición de apagado antes de que el interruptor maestro se haga girar a la de encendido”, pero en un sistema analógico, podríamos con facilidad hacer girar una hélice en un Cuisinart humano. Como modeladores, nuestro trabajo es captar las reglas; a veces, no se puede hacer que se ejecuten las reglas, en especial en sistemas analógicos.*

Revisión de los tipos de transición

Hablé de varias clases de transiciones. Déme un momento para revisarlas aquí.

Una transición de entrada ocurre cuando se entra primero a un estado, antes que suceda cualquier otra cosa en ese estado. Una transición de salida es lo último que sucede antes de salir de un estado. Una transición externa puede ser una autotransición o una transición hacia otro estado. Una autotransición ocurre cuando se sale de un estado y se vuelve a entrar al mismo. En la figura 7-6, se muestra una autotransición cuando falla el estado de “Arranque” y regresamos al mismo estado para hacer otro intento. Por último, una transición interna es una respuesta a un evento que no da como resultado un cambio de estado. Las transiciones internas no causan la ejecución de una actividad de entrada o de salida.

Creación de máquinas de estado de comportamiento

Las máquinas de estados de comportamiento son para modelar el comportamiento preciso y se implementan como código. Como consecuencia, en las máquinas de estados de comportamiento se usan la mayoría de los elementos de los que se dispone para la creación de esquemas de estados (o diagramas de máquinas de estados). En el UML versión 2.0 se definen con precisión los elementos que se pretende se usen en las máquinas de estados de protocolo y las que están dirigidas a las máquinas de estados de comportamiento; sin embargo, si necesita un elemento en una de comportamiento, entonces úsela, incluso si no está dirigida de manera específica para una máquina de este tipo.

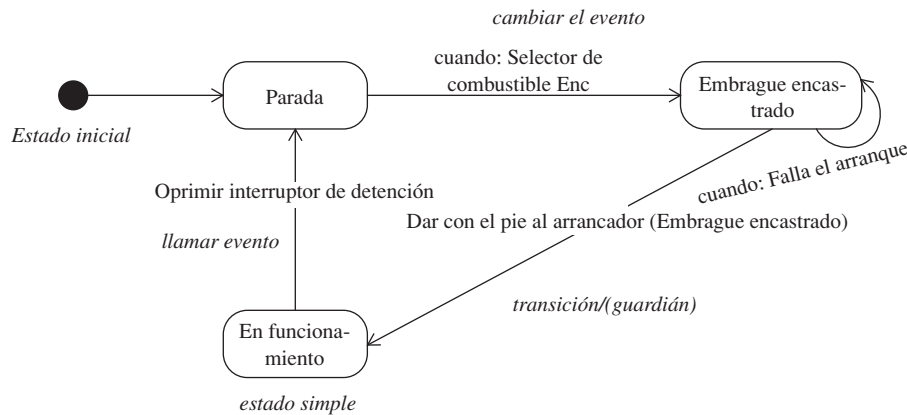


Figura 7-7 Máquina de estados de comportamiento en la que se realiza un ciclo a través de los estados de parada y funcionando en mi motocicleta.

En la figura 7-7, se ponen juntos muchos de los elementos y se describe una máquina de estados de comportamiento. Esta máquina empieza con una motocicleta en el estado de parada y las transiciones hacia los estados previo al arranque y en funcionamiento, incluyendo un camino para regresar al estado de parada. (El texto en cursivas rotula varios elementos del diagrama.)

Una clave para construir una máquina de estados de comportamiento es determinar cuánta información poner en su modelo. El modelo de la figura 7-7 podría describir información suficiente para un arranque estando montado en una motocicleta, pero si necesitaríamos entender también cómo funcionaron los sistemas de combustible, transmisión e ignición, entonces este diagrama sería insuficiente. Como con la programación, la sentencia “*Divide et impera*” (*Divide y manda*) también se aplica aquí. Lo que quiero decir por dividir y conquistar es que es posible que modeláramos los diversos subsistemas —ignición, combustible y transmisión— por separado y usáramos referencias a las máquinas de subestados para incorporar esos elementos en el diagrama de la figura 7-7. La premisa es que nuestro diagrama es un buen punto de partida, pero agregar demasiados elementos, lo que conduce a un solo diagrama monolítico, posiblemente sea más complejo que aquello que se puede captar de una sola mirada. Los diagramas complejos contrarrestan el valor del modelado.

Creación de máquinas de estado de protocolo

Las máquinas de estados de protocolo tienen que ver con una serie de secuencias lógicas predecibles. No quiere decir que estas máquinas tengan que implementarse, pero son para describir el orden de las transiciones y los estados. Por esta razón, las máquinas de estados de protocolo se usan para describir interfaces. Debido a que las interfaces no tienen definiciones, muchos de los elementos que usted usa en las máquinas de estados de comportamiento sencillamente no se necesitan en las de protocolo.

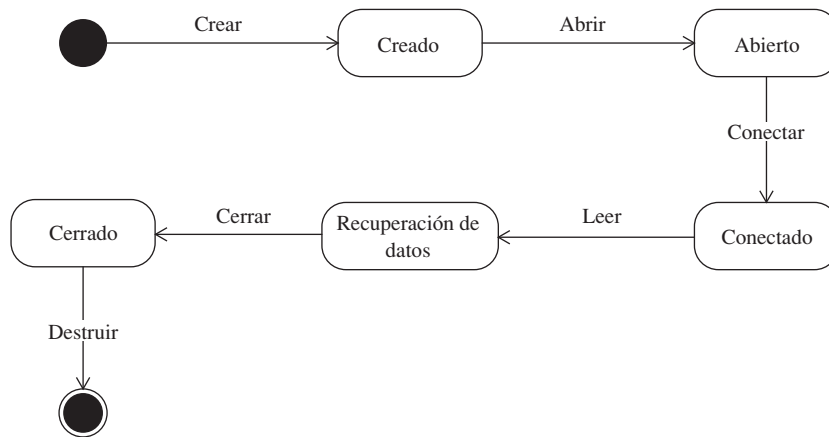


Figura 7-8 Máquina de estados de protocolo en la que se muestra la secuencia lógica y confiable de eventos que tienen que ocurrir para usar de manera correcta todas las veces una conexión a una base de datos.

Considere el uso ordenado de una base de datos. Podemos decir que se crea una conexión hacia esa base, se abre la conexión, se recuperan los datos y se cierra la conexión. Esto describe un protocolo que se puede implementar como una interfaz (o interfaces) para tener una secuencia lógica predecible y confiable de pasos —un protocolo— con el fin de garantizar que una conexión se usa con corrección todas las veces. En la figura 7-8, se muestra la máquina de estados de protocolo que se describe aquí.

Se puede usar una máquina de estados de protocolo para mostrar a los desarrolladores, en un nivel alto, cómo usar en forma correcta todas las veces las partes del sistema. Mediante la definición de una interfaz con estos elementos, les daría un medio de seguir el protocolo. La máquina de estados que se muestra en la figura 7-8 se podría usar como una ayuda de adiestramiento para garantizar que un recurso valioso, como la conexión a una base de datos, no se use en forma incorrecta.

Implementación de diagramas de estado

Los diagramas de actividad muestran cómo se soporta un solo caso de uso. Los diagramas de interacción muestran el ordenamiento en el tiempo de la creación de objetos y mensajes enviados, pero no son buenos para mostrar cómo se implementan los objetos. Las máquinas de estados muestran un objeto conforme cubre varios casos de uso y están diseñadas para mostrar cómo se deben implementar los objetos. Quizás una de las razones por las que parece que estas máquinas se usan con menos frecuencia que los diagramas de interacción es porque aquéllas están más cercanas al código que los otros tipos de

diagramas, y cuanto más próximos se encuentran al código, más tentados se sienten los programadores a empezar a codificar.

En el desarrollo de software de alta ceremonia, puede haber un mandato que imponga el número y variedad de diagramas por crear. (He trabajado en un par de ellos, pero son raros.) En virtud de que las máquinas de estados están cercanas a las líneas del código, yo sólo crearía estas máquinas para tipos riesgosos, complicados o raros de subsistemas. La producción de prototipos GUI funciona de maravilla para la mayoría de las aplicaciones y tiene un efecto apaciguador sobre los usuarios. Las máquinas de estados que representan GUI no parecen satisfacer la necesidad para tener una evidencia tangible del progreso, así como para prototipos interactivos, visualmente estimulantes.

Dicho esto, Fowler (2000) expresa que una máquina de estado se puede implementar en una de tres maneras: conmutador anidado, el patrón de comportamiento de los estados y las tablas de estados. Una *sentencia de conmutador anidado* es exactamente como suena: se evalúa algún valor semántico constante, y una serie de sentencias si... condicionales, se selecciona la caja o conmutador de sentencia que determina cuál bloque de ramales del código se debe ejecutar. Usar un conmutador anidado es la manera menos orientada a objetos de implementar una máquina de estados. La segunda elección que se da en la lista es el patrón de estados. El *patrón de estados* define comportamientos abstractos, y la máquina de estados se implementa llamando ejemplos específicos de subclases de la clase de estados abstractos. Ésta es una manera poderosa orientada a objetos de implementar el comportamiento de estados. Por último, podemos usar tablas externas de estados. En una *tabla de estados* se almacena la fuente, el disparador, el guardián, el efecto y la información del objetivo en una base de datos, archivo XML o algo semejante. Aun cuando no es un procedimiento orientado a objetos, es el más flexible porque podemos cambiar la tabla de estados sin modificar, reestructurar y desplegar el código.

En la lista siguiente se muestra cómo podríamos implementar el comportamiento del horno de microondas (de la figura 7-3), usando una sentencia de conmutador. Aunque este código es funcional, puede ser el más difícil de implementar, leer y mantener.

```
using System;
namespace HornodeMicroondas
{
    public enum EstadodelaPuerta{ Cerrada, Abierta };
    public enum EstadodelaLuz{ Apag, Enc };
    public enum EstadodelEmisordeMicroondas{ Off, On };
    public enum EstadodelTemporizador{ Apag, EnPausa, Enc };

    class Class1
    {
        private EstadodelaPuerta puerta;
        private EstadodelaLuz luz;
        private EstadodelEmisordeMicroondas emisor;
        private EstadodelTemporizador temporizador;
```



```
        case EstadodelTemporizador.Enc:
            throw new Exception("su cerebro está siendo escalfado");
    }
    break;

case EstadodelaPuerta.Abierta:
    switch(temporizador)
    {
        case EstadodelTemporizador.Apag:
            break;
        case EstadodelTemporizador.Enc:
            emisor = EstadodelEmisordeMicroondas.Apag;
            temporizador = EstadodelTemporizador.EnPausa;
            break;
        case EstadodelTemporizador.EnPausa:
            break;
    }
    luz = EstadodelaLuz.Enc;
    puerta = EstadodelaPuerta.Abierta;
    break;
    }
}
}
```

Podríamos implementar las reglas en una tabla y leerla para cada transición (tabla 7-1). Aunque sería improbable que cambiáramos los estados del microondas después del despliegue, este procedimiento es de uso común en los portales de aplicación de la Web, como dotnetnuke o IBUYSPY.

La lista anterior de código funciona bastante bien porque podemos codificar con facilidad las relaciones anidadas que reflejan los subestados de “Calentamiento” y “Apag”. La tabla 7-1 no es por completo satisfactoria, porque tenemos que llevar a la superficie los subestados anidados con el fin de captar los comportamientos deseados cuando la puerta está cerrada y reanudamos el ataque nuclear a los alimentos. (El significado es bastante claro en la tabla; podríamos agregar una columna adicional para indicar con claridad los subestados.) Vea el capítulo 9, en relación con un ejemplo del patrón de comportamiento de los estados.

Vale la pena hacer notar que el patrón de estados, un conmutador o una tabla externa no implementarán una máquina completa de estados. Estas tres opciones representan un procedimiento general, pero el código básico y otros patrones también son útiles aquí. Por ejemplo, podemos usar el patrón de comportamiento Memento para facilitar la captura y restablecer el estado interno de un objeto. Vea el capítulo 9, para obtener más información sobre los patrones y consiga un ejemplar de *Design Patterns*, escrito por Erich Gamma *et al.*

Fuente	Disparador	Guardián	Efecto	Objetivo
Puerta cerrada	Abrir puerta		Luz Enc	Puerta abierta
Puerta abierta	Cerrar puerta		Luz Apag	Puerta cerrada
Calentamiento	Abrir puerta		Luz Enc, Emisor Apag, Temporizador en pausa	Puerta abierta
Apag	Abrir puerta		Luz Enc	Puerta abierta
Puerta abierta	Cerrar puerta		Luz Enc, Emisor Enc, Temporizador Enc	Calentamiento
Puerta abierta	Cerrar puerta		Luz Apag	Apag

Tabla 7-1 Esta tabla se podría exteriorizar en una base de datos o en un archivo XML, para permitir que se cambien los comportamientos después del despliegue.

Examen

- Los esquemas de estado (o diagramas de máquinas de estado) son buenos para
 - trazar diagramas de sistemas.
 - trazar diagramas de objetos y mensajes para un solo caso de uso.
 - comprender un solo caso de uso.
 - especificar el comportamiento de un objeto a través de varios casos de uso.
- Las máquinas de estado son especialmente útiles en el examen de las GUI y de los controladores de tiempo real.
 - Verdadero
 - Falso
- Se usa una conexión para
 - combinar varias transiciones entrantes en una sola transición saliente.
 - dividir una sola transición entrante en varias transiciones salientes.
 - Tanto a como b
 - Ninguna de las anteriores
- Se usan los pseudoestados de historia para restablecer los estados anteriores.
 - Verdadero
 - Falso

5. Una actividad común se ejecuta
 - a. en el transcurso de un tiempo, y una de hacer se ejecuta de inmediato, pero se puede interrumpir.
 - b. de inmediato, y una de hacer se ejecuta en el transcurso de un tiempo y se puede interrumpir.
 - c. en el transcurso de un tiempo y se puede interrumpir, y una de hacer se ejecuta en el transcurso de un tiempo.
 - d. en el transcurso de un tiempo, y una de hacer se ejecuta también en el transcurso de un tiempo; sólo que esta última se puede interrumpir.
6. Las transiciones son líneas dirigidas rotuladas con
 - a. un evento disparador opcional, un guardián y un efecto.
 - b. un evento disparador, un guardián opcional y un efecto.
 - c. un evento disparador, un guardián y un efecto opcional.
 - d. opcionalmente, un evento disparador, un guardián y un efecto.
7. Las transiciones internas hacen que se ejecuten una actividad de entrada y una de salida.
 - a. Verdadero
 - b. Falso
8. Las autotransiciones hacen que se ejecuten una actividad de entrada y una de salida.
 - a. Verdadero
 - b. Falso
9. Un estado compuesto ortogonal
 - a. está dividido en regiones, y sólo se puede activar una de ellas a la vez.
 - b. está dividido en regiones, y sólo se puede activar un subestado a la vez.
 - c. está dividido en regiones, y sólo se puede activar un subestado por región a la vez.
 - d. está compuesto de una sola región, y se pueden activar múltiples subestados en forma simultánea.
10. Un estado compuesto no ortogonal
 - a. está compuesto de regiones, y sólo se puede activar una de ellas a la vez.
 - b. no está dividido en regiones, y sólo se puede activar un subestado a la vez.

- c. no está dividido en regiones, y se pueden activar múltiples subestados a la vez.
- d. está dividido en regiones, y se puede activar un subestado por región a la vez.

Respuestas

- 1. d
- 2. a
- 3. c
- 4. a
- 5. b
- 6. a
- 7. b
- 8. a
- 9. c
- 10. b



CAPÍTULO

8

Modelado de componentes

Cuanto tenía 15 años, compré mi primer auto por 325 dólares. Adelante; ríase; en 1981, un auto de 325 dólares era tan malo como usted pueda imaginar. Por supuesto, siendo industrioso, empecé a hallar maneras de restaurarlo y hacerlo tan respetable para el camino en tanto yo supiera cómo hacerlo. Una de las primeras cosas de las que me di cuenta acerca de este cubo oxidado Cutlass Oldsmobile 1974 —además de que el asiento delantero no quedaba fijo, lo que causaba que se fuera por completo hasta delante, cuando me detenía, y por completo hasta atrás, cuando aceleraba, del agujero del tamaño de un balón de futbol en el radiador y de los neumáticos de cuatro tamaños diferentes— era que se necesitaba reemplazar la banda del motor, con trayectoria en serpentina. Pensé que reemplazar una banda con estas características era una tarea que podía manejar.

Después de llevarme el auto a casa, me enfrasqué en el reemplazo de la banda. Empecé por quitar el radiador, la bomba de agua y el alternador. El lector se imagina el cuadro. Me di cuenta de que éste era un trabajo más grande de lo que podría ser capaz de hacer y resolví llevar el auto al taller de reparaciones de la Firestone que estaba en la carretera. El muchacho del taller aflojó el alternador, lo hizo girar hacia dentro, deslizó la banda sobre el ventilador y el alternador, regresó el alternador a su lugar, apretó los pernos y terminó en 10 minutos. Sólo recibí mi primera lección, con un costo de 35 dólares, de lo que vale el conocimiento.

¿Por qué le relaté esta historia? La respuesta es que cuando le digo que es posible que repase de modo superficial este capítulo y tal vez no necesite los diagramas de componentes, créame.

Para modelar componentes, usamos muchos de los mismos símbolos y conectores que hemos expuesto en los capítulos anteriores, pero existe una diferencia. Los *componentes* son trozos autónomos de código —piense en subsistema— que se pueden volver a usar desplegándolos de manera independiente. (Los componentes no tienen que ser grandes, pero en general, son mucho más que una sola clase o un par de clases vagamente relacionadas.) En general, los componentes tienen múltiples interfaces suministradas y requeridas y se encuentran en aplicaciones grandes y complejas con docenas o cientos de clases del dominio. Por tanto, si está estructurando una simple aplicación cliente-servidor, un sitio web básico o una aplicación de Windows para un solo usuario, entonces es posible que no necesite diagramas de componentes. Si está estructurando una solución empresarial con cientos de clases del dominio y elementos susceptibles de volver a usarse, entonces podría necesitar diagramas de componentes.

No toda clase es de dominio. Las clases de arreglos, colecciones e interfaces gráficas de los usuarios (GUI) no son clases del dominio. Las clases del dominio son las que captan el problema de este último: estudiante, registro, clases en una aplicación de matrícula; reservaciones, personas, procesos, tiempo servido en la aplicación de administración de una prisión, y depósitos, retiros y cuentas en un sistema bancario. Si tiene cientos de estos tipos de clases, entonces puede ser que necesite diagramas de componentes.

Ejemplos obvios de componentes muy complejos incluyen cosas como aplicaciones de Microsoft Office, Enterprise Java Beans, COM+ y CORBA. Quizás componentes menos complejos podrían incluir el componente de persistencia en la base personalizada de datos del lector.

Dicho esto, lo aliento a que sólo vea superficialmente este capítulo, pero debe leerlo por completo si sabe que está estructurando un sistema grande o está intentando organizar los esfuerzos de un equipo grande; un panorama general del sistema le ayudará a orquestrar los esfuerzos de todos los desarrolladores. En este capítulo, se mostrará la mecánica directa de creación de los diagramas de componentes. Para obtener directrices excelentes sobre las circunstancias de estructuración de los diagramas de componentes, consulte *The Object Primer: Agile Model-Driven Development with UML 2.0*, de Scott Ambler, 3a. edición. En este capítulo aprenderá

- Cómo describir los componentes
- Cómo especificar las interfaces suministradas y requeridas
- A alternar las maneras para especificar un componente con base en el detalle que quiere transmitir

Introducción del diseño basado en componentes

Existen dos métodos generales para derivar componentes: el método componentes-interfaz, y el método que privilegia el desarrollo de clases. Cualquiera de ellos es útil. Permítame explicar cómo funcionan y el porqué de su utilidad.

Diseño componentes-interfaz

Conocido también como método de arriba hacia abajo, es el más recomendado por algunos especialistas. Este enfoque implica que primero se definen los componentes —es decir, las grandes piezas del sistema— y después las interfaces correspondientes. Una vez que los componentes y las interfaces se han definido, es posible dividir la implementación del sistema entre los participantes, organizándolos en varios grupos o equipos encargados de construir cada componente. Como todos los involucrados están de acuerdo respecto de cómo construir las interfaces, los desarrolladores son libres de implementar como deseen las partes internas del componente.

Considero que este método puede resultar si el equipo está utilizando muchos componentes bien establecidos con interfaces de dominio público. Sin embargo, definir todos los nuevos componentes desde esta perspectiva puede constituir todo un reto.

Por otro lado, utilizar el enfoque “de arriba hacia abajo” implica el compromiso de instaurar un estilo de implementación complejo, ya que los sistemas basados en componentes constan de hasta tres o cinco interfaces de soporte, y pasan por clases para todas las clases dominio. (Ésta es la razón por la que los componentes representan interfaces discretas y bien definidas, resultantes de las clases por las que pasan.)

En consecuencia, el problema del método componentes-interfaz radica en que es necesario diseñar (e implementar) cinco clases de soporte para cada clase dominio, razón por la cual los sistemas basados en componentes pueden resultar caros, riesgosos y muy demandantes por lo que se refiere al tiempo de desarrollo.

Diseño a partir de las clases

El método a partir de las clases (conocido también como método de abajo hacia arriba) significa que antes que nada se definen las clases —por ejemplo, las que resuelven el pro-

blema del negocio— y después la estructura. El resultado es que se dedica una importante cantidad de esfuerzo a la resolución del problema, en lugar de dedicarlo al diseño de una arquitectura complicada.

Utilizando las clases dominio y el método a partir de las clases se tiende más a la resolución del problema, aunque siempre es posible derivar componentes de las clases dominio en caso de que la complejidad de la solución se incremente o se identifique un grupo de clases que pueda depurarse y reutilizarse con más facilidad si se les encapsula en componentes.

Cualquiera de los métodos descritos puede funcionar. En el caso de aplicaciones pequeñas o medianas es probable que no se requieran muchos componentes, de manera que un diseño a partir de las clases daría buenos resultados. Por lo que se refiere a las aplicaciones de tipo empresarial, que demandan una guía experimentada, tal vez sería mejor el diseño componentes-interfaz.

Vale la pena considerar que es más fácil cambiar de decisión en aquellos modelos que estén codificados. Por lo tanto, si usted crea modelos podrá explorar y cambiar rápidamente sus decisiones en materia de diseño. Esta premisa es válida también por lo que respecta a los diagramas de componentes.

Modelado de un componente

En el Unified Modeling Language (UML), el símbolo de componente se cambió del símbolo difícil de manejar de la figura 8-1 a uno de clasificador —un rectángulo— con el estereotipo de «componente» (figura 8-2) o un pequeño icono que luce como el de la figura 8-1, en la esquina superior derecha del propio símbolo.

Tenemos que acomodarnos con algunas herramientas UML que no son por completo compatibles con el UML versión 2.0. El clasificador de la figura 8-2 muestra las secciones de atributos y operaciones del propio símbolo. Esto es aceptable.

Si su herramienta soporta el símbolo de estilo antiguo (mostrado en la figura 8-1), entonces también puede usarlo. Aparentemente, la razón para el cambio de símbolo es que los rectángulos sobresalientes del estilo antiguo dificultaban el dibujo y la fijación de conectores.

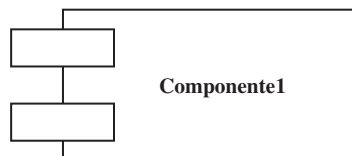


Figura 8-1 Símbolo de componente del estilo antiguo del UML.

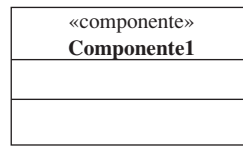


Figura 8-2 Símbolo revisado de componente en el UML versión 2.0.

Especificación de las interfaces proporcionadas y requeridas

En el capítulo 6, introdujimos las interfaces proporcionadas y requeridas. Una *interfaz proporcionada* se representa por la paleta de caramelo que se extiende desde la interfaz, y una *interfaz requerida* se representa por media paleta que se extiende desde la interfaz. En términos sencillos, una interfaz proporcionada es aquella que el componente define, y una interfaz requerida es la que necesita que se complete. En la figura 8-3, se ilustra parte de un sistema financiero que muestra el componente de administración de cuentas y la capa de persistencia (por lo común, base de datos).

No se quede atascado en las limitaciones de su herramienta de modelado. Es más que probable que, si su herramienta genera código, entonces lo generará con base en el uso correcto de los símbolos para el subconjunto de la versión del UML que su herramienta soporta. Por ejemplo, en la figura 8-3, vemos los rectángulos sobresalientes más pequeños, y tuvimos que fabricar la imagen de rótula para las interfaces requerida y suministrada, lo cual, para esta versión del UML, en realidad funciona para frustrar la herramienta.

Si su herramienta tiene la misma limitación que Visio 2003 —la cual no soporta la mitad de la paleta— entonces podría indicar las relaciones de interfaz suministrada y requerida usando el conector de dependencia (figura 8-4).

NOTA A los conectores de mitad de paleta y de paleta completa y a los clasificadores se les menciona en forma metafórica como diagrama de alambrado. Si alguna vez el lector ha visto un diagrama de alambrado, entonces podría ver las semejanzas.

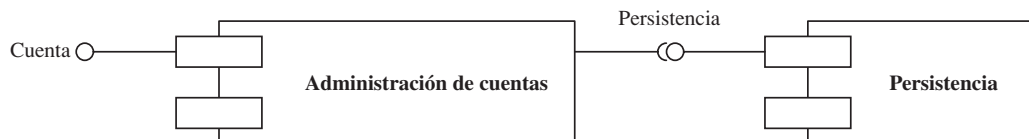


Figura 8-3 El componente “AdministradordeCuentas” proporciona la interfaz “Cuenta” y requiere la interfaz “Persistencia”.

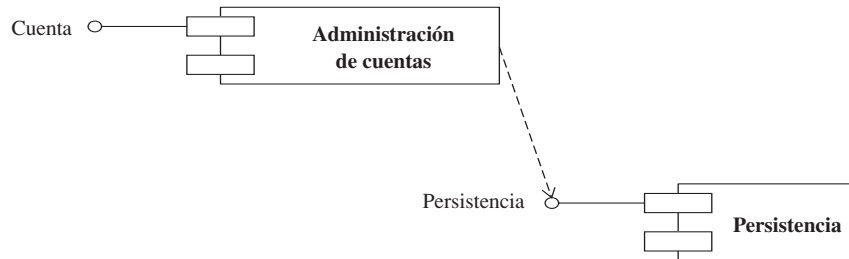


Figura 8-4 Uso de una dependencia, en lugar de la mitad de la paleta de caramelo, para modelar una interfaz requerida cuando el UML versión 2.0 no está por completo soportado por su herramienta de modelado.

Examen de los estilos de modelado de componentes

Existen diferentes maneras de trazar el diagrama del mismo componente con base en la información que queremos mostrar. Si un diagrama es para un implementador, entonces tal vez usted quiera mostrar un diagrama de caja blanca —con los detalles internos mostrados— de un componente. Si el diagrama es para un consumidor, entonces sólo necesita mostrar las interfaces proporcionadas y requeridas. Si quiere mostrar la implementación de las interfaces proporcionadas, entonces puede usar un clasificador y dependencias, porque los clasificadores son mejores para mostrar los detalles de implementación de las interfaces.

En esta sección, revisaremos algunas variaciones de los diagramas de componentes, incluyendo diagramas con más elementos. (Para esta sección del capítulo, cambié a Poseidon para el UML versión 3.1, que tiene mejor soporte para los diagramas de componentes del UML versión 2.0 que cualquiera de las copias de Rational XDE o Visio. Cuando se modela una aplicación o un sistema real, le aliento a que use la herramienta y la notación más fácilmente disponibles. Sin embargo, en un formato de libro, el cambio de herramientas le da a usted una idea de algo de la variedad que hay por ahí.)

Trazado de los diagramas de componentes para consumidores

Cuando está creando diagramas de componentes para consumidores —otros programadores que usarán los componentes— todo lo que necesitará mostrarles es una vista de caja negra del componente. Una vista de caja negra de un componente proporciona los detalles de las interfaces proporcionadas y requeridas. Si su herramienta lo soporta, puede usar un símbolo de componente y hacer una lista de las interfaces proporcionadas y requeridas, incluyendo las firmas expuestas de los métodos, o puede mostrar los clasificadores con el estereotipo «interfaz». La mayoría de las herramientas soportan las realizaciones, las dependencias y los clasificadores, de modo que este último estilo es el más fácil de crear.

Una interfaz proporcionada es aquella en la que el componente realiza; por tanto, al usar clasificadores, la paleta de caramelo se convierte en el conector de realización. Una interfaz requerida es aquella en la cual el componente depende; por tanto, al usar clasificadores, la media paleta de caramelo se convierte en el conector de dependencia con un estereotipo «usar». En la figura 8-5, está un diagrama de caja negra en el que se muestran las interfaces proporcionadas “iExceptionXMLPublisher” (“iPublicadorXMLdeExcepciones”) e “iExceptionPublisher” (“iPublicadordeExcepciones”) y la interfaz requerida “iConfigSectionHandler” (“iManejadordeConfigSecciones”). (Éste es un diagrama parcial de componentes del Exception Management Application Block; Bloc de aplicación de administración de excepciones para .NET ofrecido por Microsoft y que se usa en Motown-jobs.com.)

En la figura 8-5, el lector sabe que el componente “ExceptionManagement” realiza “iExceptionXMLPublisher” e “iExceptionPublisher”, los cuales son elementos que el consumidor será capaz de usar. El lector también sabe que algo llamado “iConfigSectionHandler” es algo que el componente necesita.

NOTA Si usted está interesado en .NET y los bloques de aplicaciones, entonces puede obtener más información en www.microsoft.com. Los bloques de aplicaciones son básicamente componentes que resuelven problemas susceptibles de volver a usarse, en un nivel más alto de abstracción que sencillamente clases en un marco de referencia.

Si el contexto es desconocido, entonces este diagrama no proporciona información suficiente, pero una vez que colocamos el componente en un contexto —en este caso, en el marco de referencia .NET—, los tipos de datos y las interfaces requeridas quedan a disponibilidad del consumidor.

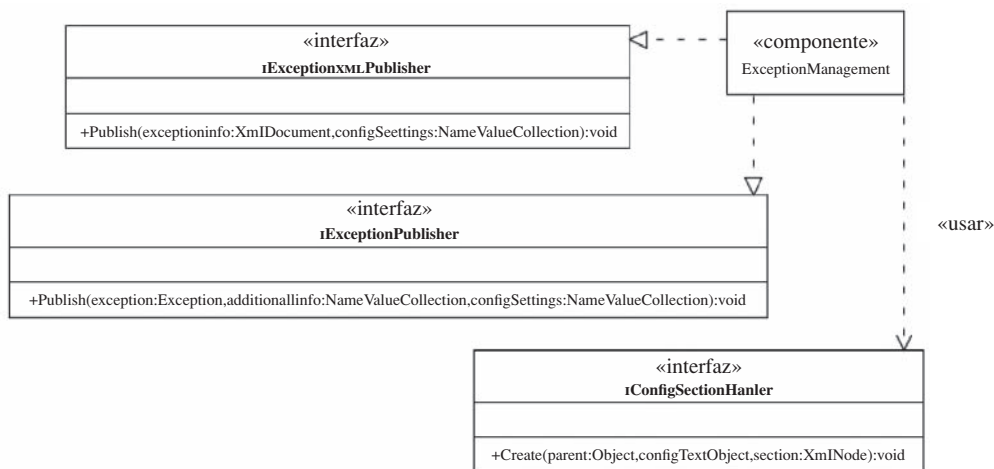


Figura 8-5 Interfaces suministradas y requeridas modeladas con el uso de conectores de realización y de dependencia, así como clasificadores, para elaborar la definición.

Trazado de los diagramas de componentes para productores

Si estamos trazando diagramas de componentes para productores —aquellos que implementarán el componente— entonces necesitamos más información. Para los productores, necesitamos mostrar los componentes, las clases y las relaciones internas que el implementador de los componentes deberá crear como código. A esto es a lo que me estoy refiriendo como vista de caja blanca, o detalles internos.

Podemos desarrollar el diagrama de componentes de la figura 8-5 y agregar detalles internos acerca del componente “ExceptionManagement”. En la figura 8-6, se muestran las interfaces proporcionadas y requeridas como paletas de caramelo y se amplía el enfoque sobre los elementos internos del componente.

En la figura 8-6, se muestran las mismas interfaces proporcionadas y requeridas, pero nuestra vista original de caja blanca muestra ahora cómo soportamos algunos de los elementos externos. Aunque puede ser que esta vista todavía no proporcione todos los detalles necesarios para implementar el componente “ExceptionManagement”, podríamos agregar atributos y operaciones a los clasificadores y combinar el diagrama de componentes con otros diagramas, como los esquemas de estados, los diagramas de clases y las secuencias. En forma colectiva, los diversos diagramas explicarían la manera de implementar el componente.

Vale la pena hacer notar que estamos expresando la misma suerte de relaciones que hemos visto antes en los diagramas de clases. También vale la pena hacer notar que los componentes, como las clases, pueden contener elementos anidados, como los componentes anidados.

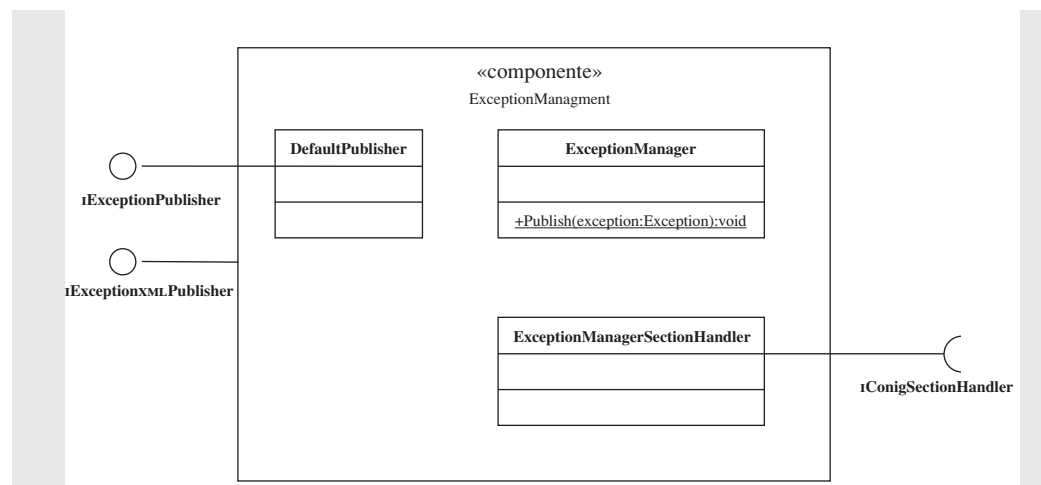


Figura 8-6 En esta figura, se cambia el enfoque para destacar la vista interna, o caja blanca, del componente.

Para experimentar con el modelado de componentes, encuentre un dominio con el que esté familiarizado o una solución existente, como la base de datos de muestra de Northwind. Vea si puede describir una vista de arriba hacia abajo de una versión en componentes de los elementos de un sistema para la plena satisfacción de los pedidos de los clientes. (Por supuesto, puede usar cualquier dominio muestra con el cual esté familiarizado.)

Examen

1. Todo modelo debe contener por lo menos un diagrama de componentes.
 - a. Verdadero
 - b. Falso
2. Un método de arriba hacia abajo para los diagramas de componentes significa que usted
 - a. define primero los componentes y, a continuación, descompone esos componentes en sus partes constituyentes.
 - b. define las partes constituyentes y, a continuación, coloca los componentes en la parte superior de esas partes constituyentes.
 - c. Ninguna de las anteriores
3. Un método a partir de clases o de abajo hacia arriba para diseñar puede ser valioso porque (seleccione todo lo que sea aplicable)
 - a. los componentes en realidad no se necesitan.
 - b. usted logra más tracción al resolver primero los problemas del dominio.
 - c. la estructuración de infraestructura es cara y tardada.
 - d. en un momento posterior, las clases del dominio siempre pueden organizarse en componentes.
4. Los símbolos de los componentes se pueden representar usando un clasificador con el estereotipo «componente».
 - a. Verdadero
 - b. Falso
5. Una interfaz proporcionada se puede representar por medio de una paleta de caramelo con nombre
 - a. o por la mitad de una paleta de caramelo.
 - b. o por una dependencia en un clasificador con el estereotipo «interfaz».
 - c. o por un estereotipo «interfaz» en un clasificador con un conector de realización.
 - d. sólo usando la paleta de caramelo.

6. Una interfaz requerida es aquella que realiza el componente.
 - a. Verdadero
 - b. Falso
7. Una interfaz requerida se puede representar por medio de la mitad de una paleta de caramelo con nombre
 - a. o por una paleta de caramelo.
 - b. o por una dependencia en un clasificador con el estereotipo «interfaz».
 - c. o por un estereotipo «interfaz» en un clasificador con un conector de realización.
 - d. sólo usando la mitad de una paleta de caramelo.
8. Los componentes pueden contener componentes anidados.
 - a. Verdadero
 - b. Falso
9. Como regla general, usted sólo usa componentes y diagramas de componentes para sistemas con 100 o más clases del dominio.
 - a. Verdadero, pero ésta es una directriz general. Los componentes pueden ayudarle a organizar una solución y a estructurar elementos susceptibles de volverse a usar que se pueden vender por separado.
 - b. Falso, porque la estructuración de componentes siempre es más barata a largo plazo.
10. Para cada clase del dominio en una arquitectura basada en componentes, usted puede necesitar de tres a cinco clases de soporte.
 - a. Verdadero
 - b. Falso

Respuestas

- | | |
|-------------|-------|
| 1. b | 6. b |
| 2. a | 7. b |
| 3. b, c y d | 8. a |
| 4. a | 9. a |
| 5. c | 10. a |

Ajuste y finalización

He trabajado en proyectos con presupuestos desde menos de 5 millones de dólares que incluían 20,000 horas-hombre, hasta proyectos con presupuestos de más de 1,000 millones de dólares y cientos de miles de horas-hombre. En algunos de estos proyectos casi no se usó modelado y diseño formales y, en otros, se usó tanto modelado y diseño que todo el ímpetu llegó a perderse. La lección es que demasiada poca formalidad puede dar por resultado un producto mediocre, de mala calidad, y demasiada formalidad puede hacer que un proyecto se atore o se cancele.

También vale la pena mencionar que he trabajado para compañías enormes que no modelan en absoluto, pero entregan software constantemente. Uno tiene que preguntarse si el éxito de esos proyectos está relacionado con cuánto dinero tuvieron que arrojar al problema esas empresas y también si el software hubiera sido mejor, más rápido y más barato si se hubiera modelado y diseñado un poco.

La respuesta se encuentra en alguna parte entre los dos extremos. En general, los modelos de software necesitan ser tan completos y precisos como lo que se está diseñando. Por ejemplo, si está construyendo algo tan complejo como una casa para un perro, entonces es probable que no necesite mucho en el camino de los modelos. Para algo tan complejo como una casa, es posible que necesite modelos tan complejos como un plano azul. Conocer el tamaño, el número de habitaciones y los materiales de construcción le ayudará a dimensionar y presupuestar un proyecto, dejando al mismo tiempo algo de espacio para la invención. Por ejemplo, las variaciones en los artefactos luminosos, el color de la pintura, el estilo del alfombrado y la colocación precisa de las tomas de electricidad se pueden dejar (dentro de lo razonable) al ingenio de los especialistas. Para las casas, los especialistas son los

carpinteros, los electricistas, los colocadores de techos y los plomeros; para el software, los especialistas son los programadores, los probadores, los DBA (database administrators; administradores de bases de datos) y los diseñadores de la interfaz gráfica para los usuarios (GUI).

La realidad es que casi todo el software es más complejo que una casa, y mucho de él se está estructurando sin planos azules (modelos UML) que describirían de manera adecuada la casa de un perro. La razón es que el modelado del software es nuevo y difícil. Además, el código se puede compilar, depurar, ejecutar y probar con resultados superficialmente mensurables. Como contraste, los modelos no se compilan y sólo se pueden “depurar” en forma manual, y no se ejecutan, y no hay manera sencilla de ponerlos a prueba. Yo apostaría a que muy pocas empresas de software están realizando revisiones técnicas del código con éxito; olvídense acerca de las revisiones técnicas del Unified Modeling Language (UML).

Lo que todo esto significa es que si usted está leyendo esto, entonces se encuentra adelante de muchos de sus semejantes en términos de la práctica del modelado del software; también significa que la definición de un proceso y encontrar un equilibrio entre demasiado mucho y demasiado poco modelado son importantes. En este libro, he dado algunos apuntadores prácticos, en el contexto del modelado y diseño con UML, que me han ayudado en el pasado. Estos apuntadores se basan en algunos proyectos que han tenido éxito y en algunos que han fallado. Para ayudarle a imaginarse cómo completar sus modelos, hablaré acerca de

- Unas cuantas cosas básicas que hacer y no hacer
- El uso de patrones y refactorizaciones conocidos
- Cuándo y cómo agregar documentación de soporte
- La validación de sus modelos

Modelado de los hacer y los no hacer

Pensé en nombrar esta sección “Las mejores prácticas de modelado”, pero los hacer y los no hacer me parecen más desmitificadores.

A principio de la década de 1990, inicié el modelado usando lo notación booleana. En aquellos días, había unos cuantos lugares en donde usted podía aprender un lenguaje como C++ de un profesional experimentado y casi ninguna parte en donde aprender modelado. Esto significa que, al principio, los pocos libros que pude conseguir y mis propios errores fueron los únicos profesores de los que dispuse. Después de más de 12 años, he mejorado, pero todavía existen unos cuantos expertos auténticos en modelado y, hasta donde puede decir, muchas universidades todavía no están ofreciendo currículos para arquitectos en software (o incluso modeladores en UML); este conocimiento todavía

es nuevo. Como consecuencia, el consejo que puedo darle se basa en mi propio estudio intenso y en muchos años de sentir mi camino. Es evidente que esto sugiere que el experto de su comunidad puede no estar de acuerdo con mi opinión. Usted conoce mejor a su propia gente que yo; si piensa que algo no funcionará o que mi consejo es cuestionable, entonces busque a esos pocos viejos sabios a quienes todos reconocen como expertos: James Rumbaugh, Ivar Jacobson, Grady Booch, Erich Gamma y Martin Fowler. Hoy unos cuantos otros, pero ya tiene el panorama. Cuando tengo preguntas acerca del modelado con UML, éstos son los amigos hacia quienes también recurro.

No tenga esperando a los programadores

La primera regla es: *no tenga a los programadores esperando los modelos*. Esto significa que debe realizar una gran cantidad de diseño antes de armar su equipo principal de programación. Resultará útil que disponga de unos cuantos programadores para ayudarle en la creación de su prototipo, pero no cree el equipo por completo hasta que tenga bien encaminados un plan del proyecto y algo del análisis y el diseño.

Por desgracia, la mayoría de los proyectos no se organizan de este modo. Llega el equipo completo y, de inmediato, empieza la presión para que todos trabajen, incluyendo los programadores. Intente crear modelos con el detalle suficiente como para tener trabajando a los programadores, pero no tan detallados que los tenga atascados esperando. Esto es difícil de hacer.

Trabaje de una macrovista hacia una microvista

Trabaje primero sobre aspectos del “gran panorama”. Por ejemplo, identifique primero las partes grandes del sistema —GUI web, macrolenguaje personalizado, servicios web y persistencia de las bases de datos— antes de trabajar sobre las clases y las líneas del código. Si quiere concebir las partes y la forma en que se ajustan entre sí, entonces el trabajo se puede dividir en subsistemas. Éste es un procedimiento de arriba hacia abajo, pero soporta una división del trabajo y le da un contexto para el trabajo más pequeño y más detallado.

Documento en forma económica

La mayor parte de la documentación es parte de la microvista. Al modelar, tenga presente que el UML es un lenguaje taquigráfico para el texto. (Usted podría diseñar un sistema completo en texto llano, ¿correcto?) Analice y diseñe una solución tan completa como se necesite, sin agregar una cantidad de notas y documentación. A menudo, los diagramas adicionales pueden aclarar un diagrama con tanta rapidez como un texto largamente desarrollado.

También puede guardar algo de la documentación para el final del proyecto, si sus modelos son difíciles de entregar. Si su cliente (interno o externo) no está pagando por los modelos, entonces consumir recursos para pulirlos puede ser un desperdicio de su tiempo y de su dinero.

Encuentre un editor

Ser un buen modelador con UML no es lo mismo que ser un buen escritor. Además de tener un segundo par de ojos para mirar sus diagramas UML, tenga escondido un buen conocimiento del español para revisar su documentación. Una vez más, sólo haga esto si los modelos son difíciles de entregar.

Sea selectivo acerca de los diagramas que elige crear

¿Por qué la pollita cruzó la carretera? Es posible que la respuesta sea porque pudo. No cree diagramas porque puede; sólo cree aquellos que resuelven problemas interesantes y sólo aquellos que en realidad se necesitan. Este enfoque también le ayudará a eliminar el problema de los programadores en espera.

No dependa de la generación del código

James McCarthy advierte acerca de dejarse llevar por la imaginación —dejar salir nuestro alocamiento—, pero si alguien le dice que debe modelar, modele, modele y déle un golpecito al interruptor para generar un ejecutable, entonces *déjese llevar*. Nos encontramos a una década o dos de que el apoyo a la tecnología y a la educación generó aplicaciones. Nunca he visto este trabajo de aproximación y he hablado con varios consultores de Rational, quienes están de acuerdo conmigo. La generación de código es una buena idea, pero nos falta mucho por recorrer para automatizar la generación de software.

Modele y structure disminuyendo el riesgo

El software suele tener unas cuantas cajas muy importantes de la empresa y un montón de cajas de soporte de esa empresa. El principio guía es estructurar primero las partes más difíciles y más importantes del software. El ataque a los problemas difíciles le ayuda a evitar sorpresas desagradables y, con frecuencia, se puede embarcar el software si las cajas importantes de la empresa están apoyadas, incluso cuando los adornos adicionales no son tan grandes. En mi experiencia, éste es uno de los errores más grandes que se cometen en los proyectos: estructurar primero las cosas fáciles.

Si es obvio, no lo modele

Los bloques, los componentes, las herramientas de terceras partes y los marcos de referencia de las aplicaciones se encuentran fuera de su control; todo lo que puede hacer es usarlos; a menos que usted también posea esos elementos, lo cual es raro. No desperdicie el tiempo modelando lo que no posee. Si debe modelar herramientas de terceras partes para ayudar a los desarrolladores a usarlas, entonces modélelas como cajas negras: todo lo que necesita es modelar su presencia e interfaces, y sólo necesita modelar las interfaces que en realidad está usando. Si, por ejemplo, sus desarrolladores pueden usar ADO.NET o el Data Access Application Block (Bloc de aplicación de acceso a datos), entonces sencillamente indique que lo está usando. Eso es suficiente.

Haga hincapié en la especialización

Otra equivocación es la generalización de los miembros del equipo. Los equipos de software constan de personas con aptitudes y conocimientos variados, pero existe una cantidad tremenda de documentación y evidencia históricas acerca de que la especialización es algo bueno: *Wealth of Nations* de Adam Smith, las líneas de montaje de Henry Ford y la antigua frase latina “*Divide et impera*”. Considerar en primer lugar la división del problema, la intensidad del enfoque, la especialización y la estructuración de los elementos críticos le llevará a recorrer gran parte del camino al éxito.

Uso de patrones de estado conocidos

Los patrones no son una idea original o nueva. La aplicación de los patrones en software parece tener su origen en un libro de 1977 titulado *A Pattern Language*, escrito por Christopher Alexander *et al.* Lo extraño es que este libro es acerca del diseño de ciudades y poblados pequeños, y patrones como *espacios verdes*. El patrón de espacios verdes significa que los poblados deben tener parques.

Evidentemente, es una extrapolación inteligente convertir un libro acerca del diseño de ciudades en un concepto que revolucione el software —esto no sucede de cualquier otra manera que no sea por extrapolación— pero se ha demostrado que el buen uso de patrones ayuda a lograr un buen software. La pregunta es: dado que los patrones de software están documentados, ¿necesita usted agregarlos a sus diagramas UML cuando los use en sus diseños? La respuesta es posiblemente.

Los patrones de software son plantillas, pero existe cierta latitud en cómo implementarlos. Cada vez que se emplea un patrón, el lector tendrá nombres diferentes de clases basados en el dominio de la solución, y muchos patrones se pueden implementar de

maneras diferentes. Por ejemplo, los eventos y los manejadores de eventos son una implementación del patrón observador, pero ésta no es precisamente la manera en que está documentado el observador. Microsoft considera las páginas ASP y detrás del código para ASP.NET como una implementación del controlador de visión de modelos (MVC, model-view-controller), pero usted no verá ASP.NET mencionado en la definición del patrón. Por consiguiente, la respuesta es sí, en muchos casos; si usa un patrón, entonces debe incorporarlo en sus modelos para colocarlo en el contexto de su dominio del problema. Sin embargo, si tiene un equipo muy experimentado, entonces sencillamente podría decirles a los desarrolladores que usen aquí el patrón MVC, de observador o de estado.

SUGERENCIA Una buena sugerencia es identificar los patrones cuando los use. La identificación de los patrones de diseño bien documentados eliminará o por lo menos mitigará la necesidad de que duplique esa documentación en sus diseños.

Una buena regla empírica es que el buen software se basa en patrones. La clave es aprender acerca de los patrones de diseño, concebir las áreas clave en donde ayudarán al diseño de usted y, a continuación, incorporarlos en sus diseños.

En la figura 9-1, se demuestra cómo podemos modelar el patrón de comportamiento de estados, pidiendo prestado del horno de microondas del capítulo 7. En este ejemplo, se demuestra cómo podemos modelar un patrón conocido en donde sólo los nombres cambian. En la figura 9-2, se muestra el modelo clásico de patrón de observador, y en la figura 9-3 se muestra una variación de este patrón que refleja las variaciones en el modelo clásico pero, no obstante, de observador.

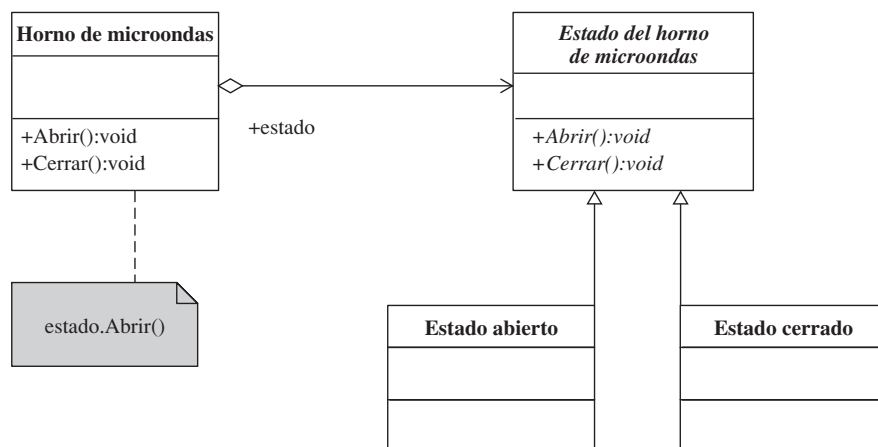


Figura 9-1 Esta figura es una implementación clásica del patrón de comportamiento de estados para el ejemplo del horno de microondas del capítulo 7.

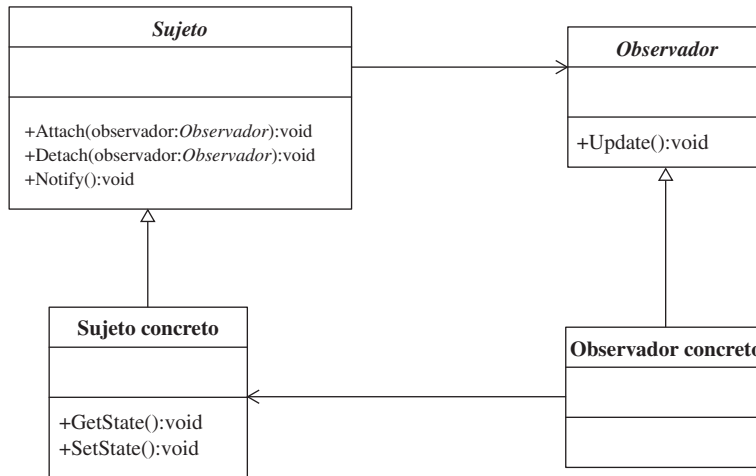


Figura 9-2 Diagrama clásico del patrón de observador, también conocido como *publicar-suscribir*.

Advierta que en el ejemplo clásico del observador (vea la figura 9-2), no se usa una interfaz; sin embargo, en la figura 9-3 usé una interfaz. El resultado es que cualquier cosa puede implementar “iEscucha” y desempeñar el papel de escucha. Esta implementación es útil en los lenguajes de herencia sencilla y también es útil para mover mensajes por una aplicación de una manera unificada. La razón para agregar este modelo e indicar que es una implementación de observador es que se trata de una implementación diferente a la clásica, pero la documentación para el observador todavía ayuda a aclarar la razón fundamental para su uso.

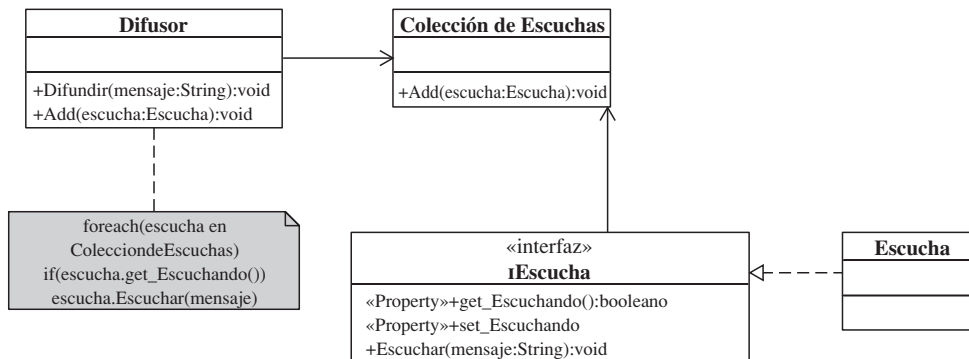


Figura 9-3 Variación del patrón de comportamiento de observador que menciono como *difundir-escuchar*, lo cual está muy próximo a la noción de publicar-suscribir del observador.

Refactorización de su modelo

Este libro no es el mejor foro para enseñar los patrones de diseño o la refactorización. El UML es distinto a los patrones, pero éstos se describen con el uso del UML y el texto de otros libros. La refactorización es diferente tanto del UML como de los patrones. Aun cuando existe cierto traslape entre los patrones y las refactorizaciones —por ejemplo, tanto Singleton como Factory son patrones de creación así como refactorizaciones—, la refactorización es algo que, en general, se realiza después de que se ha escrito el código, para mejorar el diseño del código existente. Dicho esto, no hay razón para que usted no pueda refactorizar sus modelos.

Suponga, por ejemplo, que tiene una signature de mensaje en un diagrama de interacción que tiene varios parámetros. Antes de liberar el diagrama a sus programadores, podría aplicar la refactorización “IntroduceParameterObject” (“IntroducirParámetroObjeto”). Esta refactorización sencillamente dice convertir una signature larga de un método en una corta, mediante la introducción de una clase que contenga todos los parámetros necesarios para un método en particular y cambiar ese método para que acepte un caso de esa clase.

No hay necesidad de hacer otra cosa que no sea introducir la clase parámetro y cambiar la signature del método, pero tendría que saber acerca de la refactorización y la justificación para hacer este cambio. Para aprender más acerca de la refactorización, lea *Refactoring: Improving the Design of Existing Code*, escrito por Martin Fowler, y obtenga así más información sobre este tema.

Los patrones y las refactorizaciones no son parte del UML, pero le ayudarán a crear mejores diagramas UML. Los buenos diseños no necesitan tener un UML gramáticamente perfecto, pero los patrones y las refactorizaciones harán que sus diseños sean mejores.

Modo de agregar documentación de soporte

Muchas herramientas de modelado aceptarán cualquier documentación que usted cree y combine con sus diagramas y producirán documentación de alta calidad para el modelo —en general, HyperText Markup Language (HTML)— con referencias cruzadas e indexado. Sin embargo, si usa una herramienta como Excel, Word, Bloc de notas o algo junto con la herramienta UML para crear su documentación, entonces está anulando esta característica de la mayoría de las herramientas.

Le recomiendo que transmita con imágenes tanto significado como sea posible. La razón sencilla es que las imágenes transmiten más información en un formato conciso que resmas de texto. Si necesita texto, entonces intente con restricciones y notas en el modelo, pero manténgalas en un mínimo. Por último, si debe agregar mucha documentación, no retrase a los programadores mientras la escribe. Tendrá suerte si los programadores incluso leen sus modelos —la verdad hierde— dejando en paz el texto largamente desarro-

llado. Por desgracia, muchos programadores se sienten perfectamente felices si codifican lo que les viene a la mente o cualquier cosa que hayan codificado en su último proyecto. Los modelos complicados pueden finalizar siendo ignorados.

En general, para la posteridad, me gusta incluir un panorama general arquitectónico escrito, en un documento separado, que describe el sistema en un alto nivel. Algunas personas sencillamente no pueden o no leerán los modelos —piense en los administradores o, incluso, en futuros programadores—, pero yo creo estos documentos cerca del final del proyecto cuando todos los demás están ocupados depurando y haciendo pruebas.

Tenga presente que el UML y el modelado son sólo una faceta del desarrollo del software. El modelado debe ayudar, no obstaculizar, el proceso en su conjunto.

Validación de su modelo

Muchas herramientas validarán los modelos en forma automática. Por desgracia, cada herramienta es diferente y cada una de ellas parece soportar aspectos diferentes del UML. El lector puede volverse loco tratando de eliminar los errores de los que informen las herramientas de validación de los modelos UML. Yo no consumiría mi tiempo aquí. Punto.

Su tiempo será mejor empleado codificando ejemplos que muestren a los desarrolladores cómo implementar el modelo, enseñando a los desarrolladores la manera de leer los modelos y recorriendo los modelos con los desarrolladores para ver si tienen sentido y se pueden implementar. En general, en el momento en que usted y los desarrolladores se encuentren felices con un diagrama en particular, el programa tiene la mayor parte de lo que su diagrama describe codificado de todos modos.

Por último, precisamente como no embarcaría código con advertencias y errores, tampoco quiero embarcar modelos con advertencias o errores. Si la validación de un modelo informa de un error, esto suele significar que estoy usando una característica de manera incoherente con la implementación del UML que mi herramienta específica soporta. Antes de ponerle un listón a mi modelo y seguir con alguna otra cosa, trataré de resolver las discrepancias de las que informan las herramientas de validación. No obstante, históricamente, los clientes por lo común no han estado dispuestos a pagar por este esfuerzo.

Examen

1. Un modelo sólo está completo cuando contiene por lo menos uno de cada tipo de diagrama.
 - a. Verdadero
 - b. Falso

2. Los diagramas de componentes son absolutamente necesarios.
 - a. Verdadero
 - b. Falso
3. Debo seleccionar un procedimiento de arriba hacia abajo o a partir de clases para modelar, pero no puedo combinar las técnicas.
 - a. Verdadero
 - b. Falso
4. Se ha argumentado que la especialización conduce a ganancias en la productividad.
 - a. Verdadero
 - b. Falso
5. Los patrones de diseño son parte de la especificación UML.
 - a. Verdadero
 - b. Falso
6. La refactorización no es parte de la especificación UML.
 - a. Verdadero
 - b. Falso
7. La mayoría de los expertos están de acuerdo en que los patrones y las refactorizaciones mejorarán la implementación del software.
 - a. Verdadero
 - b. Falso
8. El UML es un estándar y todos están de acuerdo en que debe usarse.
 - a. Verdadero
 - b. Falso
9. Es esencial usar una herramienta para validar los modelos.
 - a. Verdadero
 - b. Falso
10. Todas las herramientas de modelado del UML son capaces de generar de manera eficaz aplicaciones enteras, completas.
 - a. Verdadero
 - b. Falso



Respuestas

1. b
2. b
3. b
4. a
5. b
6. a
7. a
8. b
9. b
10. b

Visualización de su topología de despliegue

Topología de despliegue significa, sencillamente, la forma en que lucirá su sistema cuando lo ponga en uso. Para esto, puede construir un *diagrama de despliegue*. Este tipo de diagramas muestran al lector los elementos lógicos, sus ubicaciones físicas y cómo se comunican estos elementos, así como el número y variedad de elementos físicos y lógicos.

Use diagramas de despliegue para mostrar en dónde está su servidor web y si tiene más de uno; úselos para mostrar dónde está su servidor de bases de datos y si tiene más de uno, así como cuál (cuáles) es (son) la(s) relación (relaciones) del (de los) servidor(es) con los otros elementos. Este tipo de diagramas pueden mostrar cómo están conectados estos elementos, cuáles protocolos están usando para comunicarse y cuáles sistemas operativos o dispositivos físicos, incluyendo las computadoras y otros dispositivos, están presentes.

Resulta clara la implicación de que, si no cuenta con la mayoría de estos elementos, entonces es posible que no necesite crear un diagrama de despliegue. Si está creando una aplicación simple que es única o, incluso, una simple aplicación de base de datos para un solo usuario, un sitio web, una aplicación de consola o un servicio, entonces puede pasar por alto la creación de un diagrama de despliegue.

Los diagramas de despliegue no son difíciles de crear, en general no contienen un gran número de elementos y sólo se necesitan para aplicaciones de complejidad mediana a grande. Estos diagramas son buenos para la visualización del panorama de su despliegue, para sistemas con múltiples elementos. Desde luego, usted es libre para crear un diagrama de despliegue para todo modelo, pero ésta es un área en donde podría economizar.

Modelado de nodos

Los *nodos* son cajas tridimensionales que representan dispositivos físicos que pueden ser computadoras, aunque no necesariamente, o entornos de ejecución que pueden ser computadoras, sistemas operativos o entornos de autocontención, como COM+, IIS o un servidor Apache.

Los dispositivos físicos por lo común incluyen computadoras, pero pueden incluir cualquier dispositivo físico. Al trabajar en un proyecto para Lucent Technologies, hace algunos años, estuve escribiendo software para teléfonos de hoteles: mover los ajustes telefónicos de teléfono a teléfono y controlar los sistemas de conmutación. En mi diagrama de despliegue, mostré las computadoras, los teléfonos y los conmutadores telefónicos. Más recientemente, estuve trabajando en un proyecto para Pitney Bowes. Estaba escribiendo un almacén de embarque multinacional para soportar el concepto de transportador universal. En gran parte de ese almacén se usó MSMQ —formación de cola de mensajes con COM+—, de modo que el diagrama de despliegue reflejó nodos que representaban un entorno de ejecución COM+.

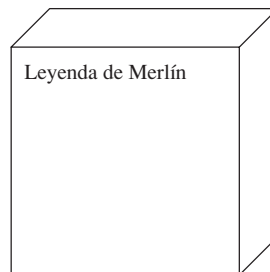


Figura 10-1 Nodo con un solo nombre en un diagrama de despliegue del Unified Modeling Language (UML).

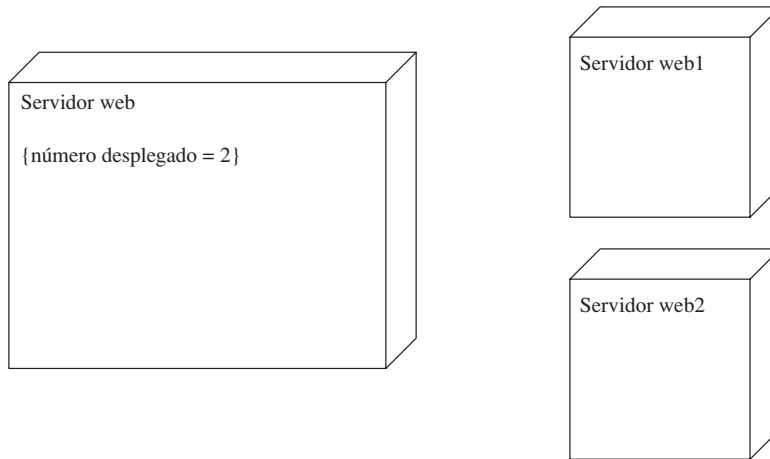


Figura 10-2 En este diagrama, se muestra una etiqueta que indica que existen dos servidores web (izquierda) y dos servidores web de nodo físico a la derecha.

El símbolo básico para un nodo es un cubo tridimensional con el nombre en el mismo (figura 10-1). Si usted quisiera modelar varios nodos del mismo tipo, entonces podría usar una etiqueta que indique el número de casos de ese nodo, o podría agregar múltiples nodos al diagrama. En la figura 10-2, se muestra cómo podría modelar una granja web con el uso de la etiqueta de nodos múltiples a la izquierda y los símbolos de esos nodos a la derecha.

Además de usar etiquetas para indicar la multiplicidad de los nodos, podemos usarlas para indicar información acerca del nodo. Por ejemplo, en nuestro ejemplo del servidor web, podríamos indicar que todos los nodos están ejecutando el servidor IIS y Windows 2003. En la figura 10-3, se muestran estas etiquetas adicionales.

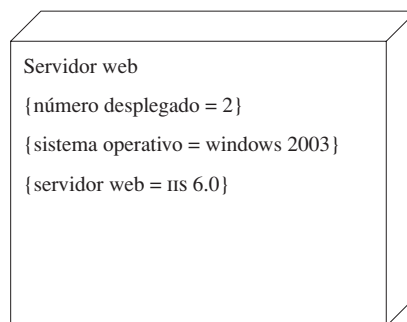


Figura 10-3 Diagrama de despliegue parcial en el que se muestran nodos múltiples y detalles acerca del sistema operativo y de la versión del servidor web.

SUGERENCIA La PC virtual es una herramienta que uso para tener múltiples computadoras lógicas en una sola computadora. Es una manera excelente para probar software beta o tener una máquina limpia para despliegue local, como el de una aplicación web para hacer la prueba acerca de dependencias y el montaje y la configuración apropiados.

Por último, podemos agregar uno o dos estereotipos a un nodo —«dispositivo» o «entorno de ejecución»— para indicar si estamos hablando acerca de un dispositivo físico o de un entorno de ejecución. En la figura 10-4, se ilustra un diagrama alternativo en el que se muestra un solo servidor web que ejecuta un entorno de ejecución en un ejemplo de PC virtual.

NOTA Un reto interesante y recurrente es que, en los proyectos a largo plazo, los desarrolladores vienen y se van. En general, el resultado de una transición es que alguien que ha permanecido en el proyecto tiene que dedicar una tarde o todo un día ayudando al recién llegado a configurar su máquina. Un proyecto de instalación o un diagrama de despliegue para el entorno de desarrollo podrían ser tan útiles como un diagrama de despliegue para un sistema de producción. (Si tiene un poco de tiempo adicional, inténtelo y vea cómo funciona.)

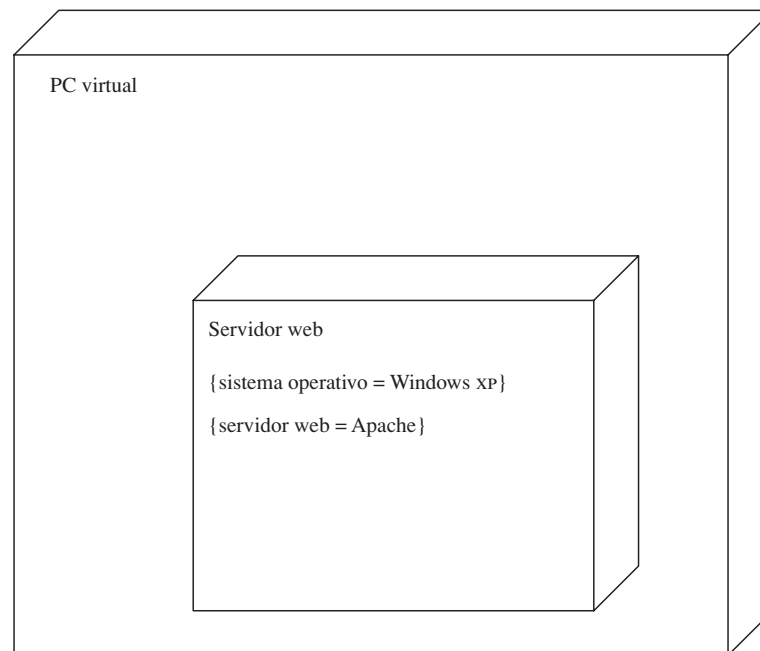


Figura 10-4 Nodo en el que se muestra la PC virtual usada como un entorno de ejecución.

Manera de mostrar artefactos en nodos

Los *artefactos* son las cosas que está desplegando. (Si está combinando el desarrollo de hardware y software, entonces también podría desplegar sus propios nodos, pero sólo estoy hablando acerca de software.) Los artefactos se modelan usando el símbolo de clase y un estereotipo «artefacto». Los artefactos pueden ser EXE, DLL, archivos HTML, documentos, archivos .JAR, ensamblajes, guiones (scripts), archivos binarios o cualquier otra cosa que despliegue como parte de su solución. Por lo común, los artefactos binarios son componentes y podemos usar una etiqueta para especificar cuál componente representa un artefacto. En la figura 10-5, se muestra un artefacto que representa un .DLL y, en la 10-6, se muestra cómo colocaríamos ese artefacto en un nodo.

De manera tradicional, podría usted encontrar cierto traslape entre los diagramas de componentes y los de despliegue. Por ejemplo, si un artefacto implementa un componente, puede mostrar el componente implementado como una etiqueta, o puede agregar el componente al nodo que muestra la dependencia entre el artefacto y el componente. En la figura 10-7, se muestra la etiqueta de componente usada para indicar que el artefacto mostrado implementa el componente usado “AdministracióndeExcepciones”, y en la figura 10-8 se muestra lo mismo con el uso de la dependencia más verbosa fija al símbolo de un componente. (El estereotipo «manifestar» significa que el artefacto es una manifestación del componente.)

NOTA También se pueden usar dependencias entre los artefactos, para indicar que un artefacto depende de un segundo. Esto apoya la noción de referencias en .NET, se usa en Delphi e incluye C++. Por ejemplo, el “AdministracióndeExcepciones.dll” tiene una dependencia del “Sistema.dll” (no mostrada) que contiene la clase “RegistrodeEventos” en .NET.

Como una alternativa para colocar varios diagramas anidados de clases en un solo nodo, el UML permite hacer listas de artefactos como texto. Por ejemplo, un sitio web basado en ASP.NET contendrá un binario, varios archivos .ASPX que contienen HTML y ASP, y quizás otros documentos o elementos, como un guión. Usar el símbolo de clase para más de un par de artefactos dará por resultado que el nodo sea ridículamente grande. Haga una lista de los artefactos como texto, si hay muchos de ellos. En la figura 10-9, se muestra cómo podemos hacer una lista de varios artefactos en un solo nodo.



Figura 10-5 Artefacto que representa un binario que es el ejecutable que soporta un sitio web.

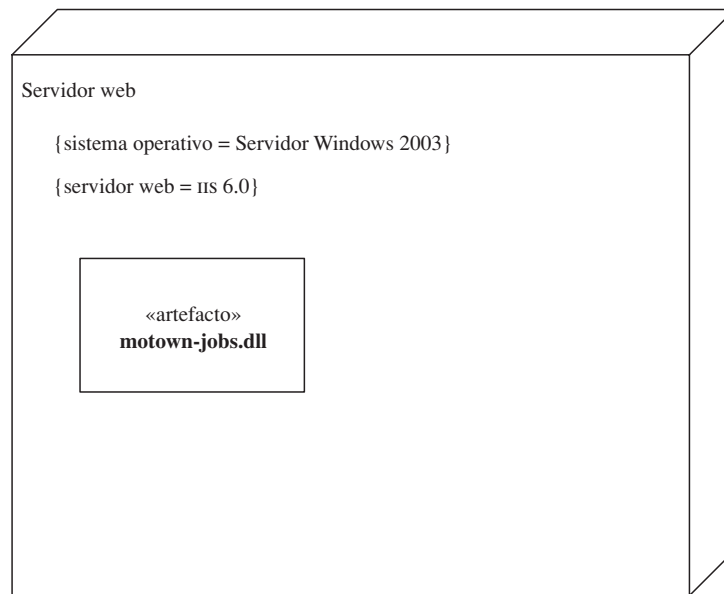


Figura 10-6 En los diagramas de despliegue, los artefactos se despliegan hacia los nodos, de modo que podemos mostrar un artefacto anidado en un nodo.

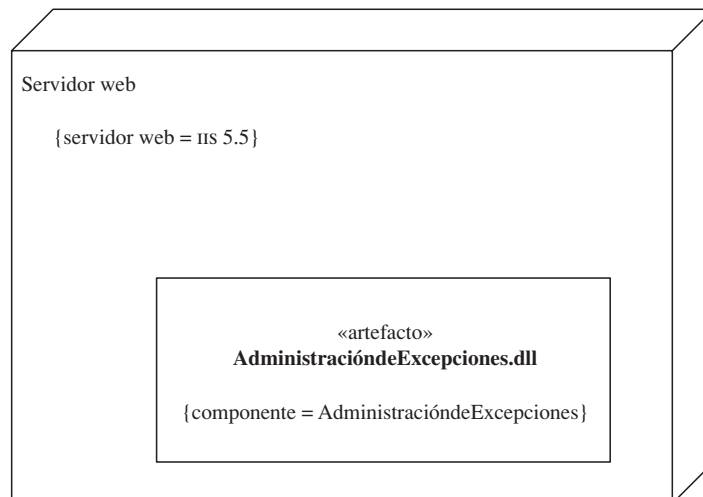


Figura 10-7 La especificación del componente en un artefacto implementa el uso de una etiqueta.

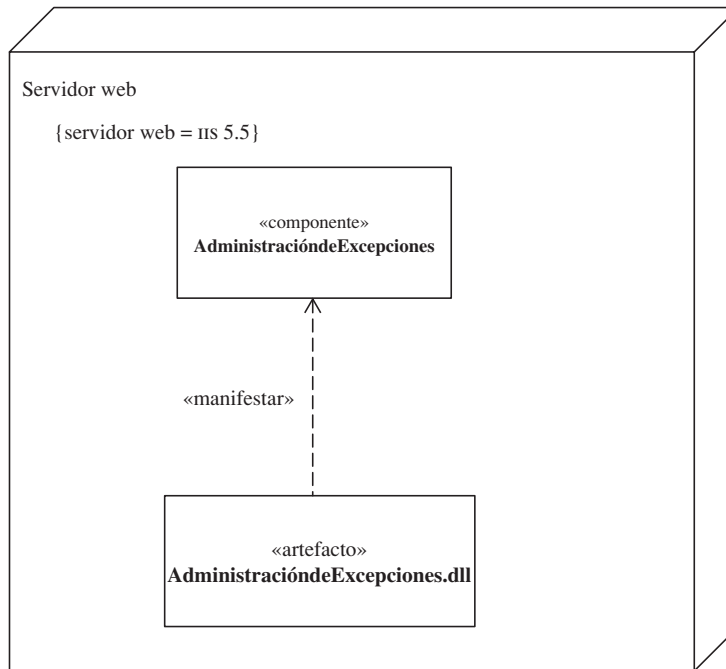


Figura 10-8 Especificación de la dependencia de un componente mediante un símbolo de componente.

Si estuviéramos desplegando el archivo .DLL del sitio web en una granja web, entonces cada nodo de servidor web sería idéntico. En este caso, sería más fácil usar la etiqueta de número desplegado, en un solo nodo, en lugar de repetir cada nodo y trazar los diagramas de nodos idénticos.

Técnicamente, puede agregar la combinación de nodos, componentes y artefactos que necesite, y puede variar los estilos —texto o símbolos— con base en cuántos elementos tiene un nodo. Sin embargo, tenga presente que si tiene demasiados elementos, entonces el diagrama puede volverse difícil de leer. Si tiene un diagrama complicado de despliegue, entonces intente la implementación de una macrovista con nodos, artefactos y conectores, y una microvista que amplíe los aspectos importantes del macrodiagrama. Muestre los detalles en una o más microvistas asociadas con el macrodiagrama de despliegue. Por ejemplo, considere mostrar los artefactos en el servidor web y, si quiere expandir la relación entre el artefacto “AdministracióndeExcepciones.dll”, el componente “AdministracióndeExcepciones” y la clase “RegistrodeEventos”, entonces cree una vista separada de este aspecto del sistema.

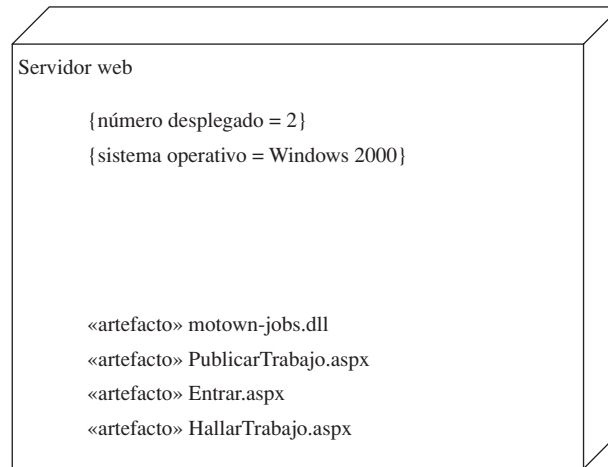


Figura 10-9 El UML también permite el uso de texto para hacer listas de artefactos.

Adición de trayectorias de comunicación

Si usted sólo tiene un nodo, entonces no necesita un diagrama de despliegue; si tiene más de uno, entonces posiblemente quiera un diagrama de despliegue y querrá mostrar cómo se conectan y comunican esos nodos.

Hay dos tipos de conectores que se usan entre los nodos y los artefactos en un diagrama de despliegue. La asociación representa una trayectoria de comunicación entre los nodos; muestra los nodos que se comunican y se puede usar un rótulo sobre esa asociación para mostrar los protocolos de comunicaciones entre nodos. Además, se puede dibujar un artefacto fuera de un nodo (un buen procedimiento para Visio, el cual no permite anidar artefactos en los nodos) y fijarlo a ese nodo con una dependencia y un estereotipo «desplegar». La dependencia desplegar entre un artefacto y un nodo significa lo mismo que un artefacto anidado o un artefacto en una lista como texto: que ese tipo de artefacto se despliega en ese tipo de nodo.

En la figura 10-10, se demuestra cómo podemos externar los artefactos como una manera alternativa de mostrar en dónde se despliegan esos artefactos y, de manera adicional, también se muestran los nodos y las trayectorias de comunicación entre éstos. Las trayectorias de comunicación se rotulan si existe algún lugar interesante de comunicación entre los nodos.

Como ocurre con todos los diagramas, puede agregar notas, restricciones y documentación. También puede agregar tanto o tan poco detalle como prefiera. He encontrado que, con cualquier diagrama, cuando ha pasado el punto en donde el significado puede

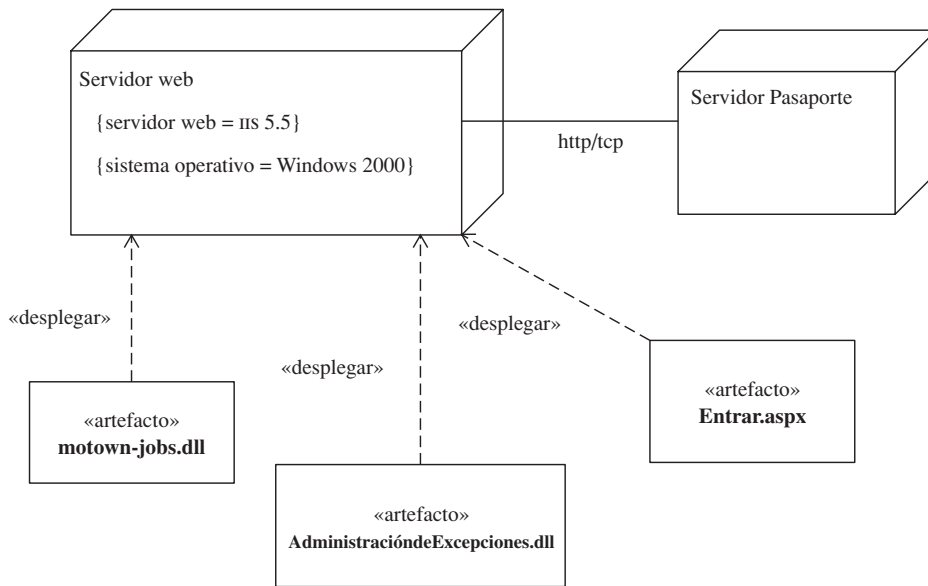


Figura 10-10 En esta figura, se muestra que los tres artefactos están desplegados en el servidor web y que el nodo de este servidor se comunica con el servidor pasaporte a través de HTTP/TCP.

ser entendido a primera vista, ese diagrama empieza a perder su valor para el lector. Una buena práctica es mantener algo de enfoque. Si quiere mostrar el sistema entero desplegado, entonces muestre los nodos y las conexiones. Si quiere desarrollar un solo nodo, entonces cree un nuevo diagrama y agregue detalles para ese nodo. ¿Puede imaginar lo difícil que sería leer un sencillo mapa del mundo si contuviera información acerca de la navegación aérea, así como los estados, las ciudades, los pueblos, los caminos, las vías férreas, los ríos, las veredas, las sendas y la topografía? Piense en los diagramas UML como mapas de su software, con niveles variables de detalle: diferentes tipos de mapas proporcionan diferentes tipos y niveles de detalle.

Ahora, habiendo dicho todo esto, debe haber una manera para que, como modelador de un sistema, pueda usted articular estos pasos en el proceso. Los diagramas de despliegue son una faceta para un entorno de *despliegue de aplicaciones vivientes*. El monitoreo de la salud y las pruebas de rendimiento proporcionan al modelador retroalimentación continua acerca de que su trabajo está funcionando. Podría seguir adelante un largo trecho acerca de esto, pero siento que dejar caer una pequeña sugerencia podría invitarle a pensar más acerca del producto final en lugar de sólo dibujar imágenes. La integración de esos artefactos con el código real y ver los frutos de su labor es absolutamente gratificante. Esto es interesante, pero no está relacionado en forma directa con el UML; tiene que ver con la incorporación de “otras” herramientas en un proceso.

Examen

1. Un nodo siempre representa un dispositivo físico.
 - a. Verdadero
 - b. Falso
2. Un nodo puede representar (seleccione todo lo que sea aplicable)
 - a. una computadora.
 - b. cualquier dispositivo físico.
 - c. un contexto de ejecución, como un servidor de aplicaciones.
 - d. Todo lo anterior
3. Los estereotipos que se aplican a los nodos pueden ser (seleccione todo lo que sea aplicable)
 - a. «dispositivo».
 - b. «componente».
 - c. «entornodeejecución».
 - d. «manifestar».
4. Se usan etiquetas para agregar detalles a un nodo.
 - a. Verdadero
 - b. Falso
5. Un servidor de bases de datos es un ejemplo de un nodo.
 - a. Verdadero
 - b. Falso
6. ¿Cuál símbolo usan los artefactos?
 - a. De paquete
 - b. De clase
 - c. De actividad
 - d. De objeto
7. Un artefacto se puede representar como texto en un nodo, una clase en un nodo y con un conector de realización y un símbolo externo de clase.
 - a. Verdadero
 - b. Falso

8. El conector y el estereotipo para un artefacto mostrado fuera de un nodo es
 - a. de realización y manifestar.
 - b. de dependencia y desplegar.
 - c. de asociación y desplegar.
 - d. de dependencia y manifestar.

9. Cuando se muestra un artefacto conectado a un componente, ¿cuál estereotipo se aplica?
 - a. «desplegar»
 - b. «usar»
 - c. «manifestar»
 - d. «extiende»

10. ¿Cuál conector se usa para mostrar comunicación entre los nodos?
 - a. De dependencia
 - b. De generalización
 - c. De asociación
 - d. De vínculo

Respuestas

1. a
2. d
3. a y c
4. a
5. a
6. b
7. b
8. b
9. c
10. d



APÉNDICE

Examen final

1. ¿Qué significa el acrónimo UML?
 - a. Uniform Model Language
 - b. Unified Modeling Language
 - c. Unitarian Mock-Up Language
 - d. Unified Molding Language
2. El UML sólo se usa para modelar software.
 - a. Verdadero
 - b. Falso
3. ¿Cuál es el nombre del proceso más íntimamente asociado con el UML?
 - a. El proceso de modelado
 - b. El Rational Unified Process
 - c. eXtreme Programming
 - d. Los métodos Agile

4. ¿Cuál es el nombre del cuerpo de normas que define el UML?
 - a. Unified Modeling Group
 - b. Object Modeling Group
 - c. Object Management Group
 - d. Los cuatro amigos
5. Los diagramas de casos de uso se usan para captar las macrodescripciones de un sistema.
 - a. Verdadero
 - b. Falso
6. Diferencie entre los diagramas de secuencia y los de colaboración (elija todo lo que sea aplicable).
 - a. Los diagramas de secuencia son diagramas de interacción; los diagramas de colaboración no lo son.
 - b. Los diagramas de secuencia representan un ordenamiento en el tiempo; los de colaboración representan clases y mensajes, pero no se implica el ordenamiento en el tiempo.
 - c. El orden en el tiempo está indicándose al numerar los diagramas de secuencia.
 - d. Ninguno de las anteriores
7. Un diagrama de clases es una visión dinámica de las clases de un sistema.
 - a. Verdadero
 - b. Falso
8. Un buen modelo UML contendrá por lo menos un diagrama de cada tipo.
 - a. Verdadero
 - b. Falso
9. ¿Cuál es el apodo del grupo de científicos que se asocia de manera más notable con el UML?
 - a. La pandilla de los cuatro
 - b. Los tres mosqueteros
 - c. Los tres amigos
 - d. El dúo dinámico
10. Los diagramas de secuencia son buenos para mostrar el estado de un objeto a través de muchos casos de uso.
 - a. Verdadero
 - b. Falso



11. ¿Qué símbolo representa un actor?
 - a. Una línea
 - b. Una línea dirigida
 - c. Una figura de palillos
 - d. Un óvalo que contiene texto
12. Un actor puede ser una persona o algo que actúa sobre un sistema.
 - a. Verdadero
 - b. Falso
13. ¿Qué símbolo representa una asociación? (Seleccione la mejor respuesta.)
 - a. Una línea.
 - b. Una línea con un triángulo que apunta hacia el elemento dependiente.
 - c. Una línea punteada con una flecha que apunta hacia el elemento dependiente.
 - d. Una línea punteada con una flecha que apunta hacia el elemento del que se depende.
14. Los estereotipos son más comunes en
 - a. los actores.
 - b. los conectores.
 - c. los casos de uso.
 - d. Ninguno de los anteriores
15. Se usa una relación de inclusión para modelar características opcionales en las que se reutiliza el comportamiento modelado por otro caso de uso.
 - a. Verdadero
 - b. Falso
16. Se usa una relación de extensión para modelar el comportamiento captado por otro caso de uso.
 - a. Verdadero
 - b. Falso
17. Generalización es sinónimo de
 - a. polimorfismo.
 - b. agregación.
 - c. herencia.
 - d. interfaces.

18. Toda capacidad de un sistema debe representarse por un caso de uso.
 - a. Verdadero
 - b. Falso
19. En una relación de inclusión, la flecha apunta hacia el
 - a. caso de uso básico.
 - b. caso de uso de inclusión.
20. Es importante implementar primero los casos de uso difíciles para mitigar pronto el riesgo.
 - a. Verdadero
 - b. Falso
21. Sinónimos para transición son conector y flujo.
 - a. Verdadero
 - b. Falso
22. En general, los diagramas de actividades constan de (elija todo lo aplicable)
 - a. nodos.
 - b. transiciones.
 - c. decisiones.
 - d. estímulos.
23. Las excepciones no se permiten en los diagramas de actividades.
 - a. Verdadero
 - b. Falso
24. En un nodo de unión y en uno de fusión se usan
 - a. símbolos diferentes.
 - b. símbolos idénticos.
 - c. símbolos idénticos o diferentes, dependiendo del contexto.
 - d. Todos los símbolos de nodos son los mismos
25. Los flujos múltiples que entran a un nodo de acción no son
 - a. una fusión implícita.
 - b. una unión implícita.

26. Los flujos esperan en una fusión hasta que
 - a. todos los flujos hayan llegado.
 - b. el primer flujo haya llegado.
 - c. usted le dice que salga.
 - d. Depende
27. La metáfora de carril todavía se usa
 - a. Verdadero
 - b. Falso
28. Pueden existir acciones sólo en una partición de la actividad al mismo tiempo.
 - a. Verdadero
 - b. Falso
29. Un nodo de unión y bifurcación se representa por
 - a. un óvalo.
 - b. un círculo.
 - c. un rectángulo.
 - d. un diamante.
30. Los diagramas de actividades son idénticos a los de flujo.
 - a. Verdadero
 - b. Falso
31. Un diagrama de colaboración es un ejemplo de
 - a. un diagrama de secuencia.
 - b. un diagrama de clases.
 - c. un diagrama de actividad.
 - d. un diagrama de interacción.
32. Un diagrama de colaboración muestra cómo evoluciona el estado de un objeto sobre muchos casos de uso.
 - a. Verdadero
 - b. Falso

33. A los diagramas de colaboración se les dio el nuevo nombre de diagramas de comunicación en el UML versión 2.0.
 - a. Verdadero
 - b. Falso
34. Los diagramas de secuencia no se pueden usar para modelar comportamiento asíncrono y de encaminamiento múltiple.
 - a. Verdadero
 - b. Falso
35. En los marcos de interacción se usa un (o una) _____ (llene el espacio en blanco) para controlar cuándo y cuál fragmento del cuadro ejecutar.
 - a. fusión
 - b. unión
 - c. guardián
 - d. mensaje asíncrono
36. El marco de interacción alt se usa para
 - a. modelar un comportamiento opcional.
 - b. modelar un comportamiento de encaminamiento múltiple.
 - c. modelar lógica condicional.
 - d. captar condiciones de error.
37. Los diagramas de secuencia y los de comunicación muestran vistas complementarias.
 - a. Verdadero
 - b. Falso
38. Un símbolo de activación muestra
 - a. la duración de un objeto en un diagrama de comunicación.
 - b. la creación de un objeto.
 - c. la duración de un objeto en un diagrama de secuencia.
 - d. la destrucción de un objeto.
39. Un esquema de numeración anidada es UML válido que se usa en
 - a. diagramas de secuencias.
 - b. diagramas de actividades.
 - c. casos de uso.
 - d. diagramas de comunicación.

40. Los diagramas de secuencias son perfectos para modelar líneas de código.
 - a. Verdadero
 - b. Falso
41. Se usa el mismo símbolo básico para las enumeraciones y las interfaces.
 - a. Verdadero
 - b. Falso
42. Al agregar clases a un diagrama, usted debe mostrar campos y
 - a. métodos.
 - b. sólo campos.
 - c. propiedades.
 - d. propiedades y métodos.
43. Una propiedad se puede modelar como una característica de una clase y
 - a. una subclase.
 - b. una clase de asociación.
 - c. una clase dependiente.
 - d. una interfaz.
44. Al modelar atributos, se
 - a. requiere que modele métodos atributos.
 - b. recomienda que no muestre métodos atributos.
 - c. recomienda que muestre los campos subyacentes para esos atributos.
 - d. Ninguno de los anteriores
45. Los tipos simples se deben modelar como características y los complejos como (seleccione la mejor)
 - a. características también.
 - b. clases asociación.
 - c. atributos.
 - d. características o clases asociación.
46. Una asociación unidireccional tiene una flecha en uno de los extremos, conocido como la fuente. El otro extremo se conoce como el objetivo.
 - a. Verdadero
 - b. Falso

47. Una agregación es lo más semejante a una
- herencia.
 - asociación.
 - composición.
 - generalización.
48. ¿Cuál es la diferencia más importante entre una agregación y una composición?
- Composición significa que la clase totalidad, o compuesta, será responsable de la creación y destrucción de la parte o clase contenida.
 - Agregación significa que la clase agregada totalidad será responsable de la creación y destrucción de la parte o clase contenida.
 - Composición significa que la clase totalidad, o compuesta, es la única clase que puede tener un caso de la clase parte en cualquier momento dado.
 - Agregación significa que la clase totalidad, o agregada, es la única clase que puede tener un caso de la clase parte en cualquier momento dado.
 - a y c
 - b y d
49. Realización significa
- polimorfismo.
 - asociación.
 - herencia de interfaz.
 - composición.
50. Una asociación nombrada se modela como un(a)
- método.
 - propiedad.
 - campo y una propiedad.
 - dependencia.
51. Una subclase tiene acceso a los miembros protegidos de una superclase.
- Verdadero
 - Falso
52. Una clase hijo sólo puede tener una clase padre.
- Verdadero
 - Falso

53. ¿Cuál de las proposiciones siguientes es falsa?
- a. La generalización se refiere a subtipos.
 - b. La clasificación se refiere a subtipos.
 - c. La generalización se refiere a ejemplos de objetos.
 - d. La clasificación se refiere a ejemplos de objetos.
 - e. Ninguna de las anteriores
54. La realización se refiere a
- a. herencia de clase.
 - b. herencia de interfaz.
 - c. promover los miembros constituyentes en una clase compuesta.
 - d. agregación.
55. Se puede tener una aproximación de herencia múltiple a través de
- a. una asociación y la promoción de propiedades constituyentes.
 - b. una realización.
 - c. una composición y la promoción de propiedades constituyentes.
 - d. una agregación y la promoción de propiedades constituyentes.
56. La clasificación dinámica —en donde un tipo de objeto se cambia en el tiempo de ejecución— se puede modelar usando
- a. generalización.
 - b. asociación.
 - c. realización.
 - d. composición.
57. A una clase asociación no se le menciona como una clase de vinculación.
- a. Verdadero
 - b. Falso
58. Un parámetro usado para retornar a un tipo único se conoce como
- a. una realización.
 - b. un calificador asociación.
 - c. una condición posterior a una asociación.
 - d. una asociación dirigida.

59. Seleccione las proposiciones correctas.
 - a. Una interfaz proporcionada significa que una clase implementa una interfaz.
 - b. Una interfaz requerida significa que una clase depende de una interfaz.
 - c. Una interfaz proporcionada significa que una clase depende de una interfaz.
 - d. Una interfaz requerida significa que una clase implementa una interfaz.
60. Cuando un símbolo de clasificador está subrayado, significa que
 - a. el símbolo representa un objeto.
 - b. el símbolo representa una clase abstracta.
 - c. el símbolo representa una interfaz.
 - d. el símbolo es un valor derivado.
61. Los esquemas de estados (o diagramas de máquinas de estados) son buenos para
 - a. trazar diagramas de sistemas.
 - b. trazar diagramas de objetos y mensajes para un solo caso de uso.
 - c. comprender un solo caso de uso.
 - d. especificar el comportamiento de un objeto a través de varios casos de uso.
62. No se deben usar las máquinas de estados para examinar las interfaces gráficas de los usuarios (GUI) y los controladores de tiempo real.
 - a. Verdadero
 - b. Falso
63. Se usa una conexión para
 - a. combinar varias transiciones entrantes en una sola transición saliente.
 - b. dividir una sola transición entrante en varias transiciones salientes.
 - c. Tanto a como b
 - d. Ninguna de las anteriores
64. Se usan los pseudoestados de historia para restablecer los estados anteriores.
 - a. Verdadero
 - b. Falso
65. Una actividad de hacer se ejecuta
 - a. en el transcurso de un tiempo, y una común se ejecuta de inmediato, pero se puede interrumpir.
 - b. de inmediato, y una común se ejecuta en el transcurso de un tiempo y se puede interrumpir.

- c. en el transcurso de un tiempo y se puede interrumpir, y una común se ejecuta de inmediato.
 - d. en el transcurso de un tiempo, y una común se ejecuta de inmediato, pero no se puede interrumpir.
66. Las transiciones son líneas dirigidas rotuladas con
- a. un disparador opcional, un evento y un efecto.
 - b. un disparador, un evento opcional y un efecto.
 - c. un disparador, un evento y un efecto opcional.
 - d. opcionalmente, un disparador, un evento y un efecto.
67. Las transiciones externas hacen que se ejecuten una actividad de entrada y una de salida.
- a. Verdadero
 - b. Falso
68. Las autotransiciones hacen que se ejecuten una actividad de entrada y una de salida.
- a. Verdadero
 - b. Falso
69. Un estado compuesto ortogonal
- a. está dividido en regiones y sólo se puede activar una de ellas a la vez.
 - b. está dividido en regiones y sólo se puede activar un subestado a la vez.
 - c. está dividido en regiones y sólo se puede activar un subestado por región a la vez.
 - d. está compuesto de una sola región, y se pueden activar múltiples subestados en forma simultánea.
70. Un estado compuesto no ortogonal
- a. está compuesto de regiones, y sólo se puede activar una de ellas a la vez.
 - b. no está dividido en regiones, y sólo se puede activar un subestado a la vez.
 - c. no está dividido en regiones, y se pueden activar múltiples subestados a la vez.
 - d. está dividido en regiones, y se puede activar un subestado por región a la vez.
71. Todo modelo debe contener por lo menos un diagrama de componentes.
- a. Verdadero
 - b. Falso

72. Un método de abajo hacia arriba para los diagramas de componentes significa que usted
 - a. define primero los componentes y los descompone en partes constituyentes.
 - b. define las partes constituyentes y coloca los componentes en la parte superior de esas partes constituyentes.
 - c. Ninguno de las anteriores
73. Un método de abajo hacia arriba para diseñar puede ser valioso porque (seleccione todo lo que sea aplicable)
 - a. los componentes en realidad no se necesitan.
 - b. usted logra más tracción al resolver primero los problemas del dominio.
 - c. la estructuración de infraestructura es cara y tardada.
 - d. las clases del dominio siempre se pueden organizar en componentes en un momento posterior.
74. Los símbolos de los componentes se pueden representar usando un clasificador con el estereotipo «componente».
 - a. Verdadero
 - b. Falso
75. Una interfaz requerida se puede representar por medio de la mitad de una paleta de caramelo con nombre
 - a. o por una paleta de caramelo.
 - b. o por una dependencia en un clasificador con el estereotipo «interfaz».
 - c. o por la conexión a una interfaz con una dependencia.
 - d. sólo usando la mitad de la paleta de caramelo.
76. Una interfaz proporcionada es aquella que realiza un componente.
 - a. Verdadero
 - b. Falso
77. Una interfaz requerida se puede representar por medio de la mitad de una paleta de caramelo con nombre y es equivalente a una dependencia entre un componente y una interfaz.
 - a. Verdadero
 - b. Falso
78. Los componentes pueden no contener componentes anidados.
 - a. Verdadero
 - b. Falso

79. Como regla general, usted sólo usa componentes y diagramas de componentes para sistemas con 100 o más clases del dominio.
 - a. Verdadero, pero ésta es una directriz general. Los componentes pueden ayudarle a organizar una solución y a estructurar elementos susceptibles de volver a usarse que se pueden vender por separado.
 - b. Falso, porque la estructuración de componentes siempre es más barata a largo plazo.
80. Para cada clase del dominio en una arquitectura basada en componentes, usted puede necesitar de dos a tres clases de soporte.
 - a. Verdadero
 - b. Falso
81. Un modelo sólo está completo cuando contiene por lo menos uno de cada tipo de diagrama.
 - a. Verdadero
 - b. Falso
82. Los diagramas de componentes sólo son necesarios para los sistemas grandes.
 - a. Verdadero
 - b. Falso
83. Debo seleccionar un método de arriba hacia abajo o de abajo hacia arriba para modelar, pero no puedo combinar las técnicas.
 - a. Verdadero
 - b. Falso
84. Se ha argumentado que la especialización conduce a ganancias en la productividad.
 - a. Verdadero
 - b. Falso
85. Los patrones de diseño no son parte de la especificación UML.
 - a. Verdadero
 - b. Falso
86. La refactorización es parte de la especificación UML.
 - a. Verdadero
 - b. Falso
87. Unos cuantos expertos están de acuerdo en que los patrones y las refactorizaciones mejorarán la implementación del software.
 - a. Verdadero
 - b. Falso

88. El UML es un estándar, y todos están de acuerdo en que debe usarse.
 - a. Verdadero
 - b. Falso
89. Es esencial usar una herramienta para validar los modelos.
 - a. Verdadero
 - b. Falso
90. Todas las herramientas de modelado del UML son capaces de generar de manera eficaz aplicaciones enteras, completas.
 - a. Verdadero
 - b. Falso
91. Un nodo siempre representa un dispositivo físico.
 - a. Verdadero
 - b. Falso
92. Un nodo puede representar (seleccione todo lo que sea aplicable)
 - a. una computadora.
 - b. cualquier dispositivo físico.
 - c. un contexto de ejecución, como un servidor de aplicaciones.
 - d. Todo lo anterior
93. Los estereotipos que se aplican a los nodos son (seleccione todo lo que sea aplicable)
 - a. «dispositivo».
 - b. «componente».
 - c. «entorno de ejecución».
 - d. «manifestar».
94. No se usan etiquetas para agregar detalles a un nodo.
 - a. Verdadero
 - b. Falso
95. Un servidor de bases de datos es un ejemplo de un nodo.
 - a. Verdadero
 - b. Falso
96. ¿Cuál símbolo usan los artefactos?
 - a. De paquete
 - b. De clase



- c. De actividad
- d. De objeto
- 97. Un artefacto se puede representar como texto en un nodo, una clase en un nodo y con un conector de realización y un símbolo externo de clase.
 - a. Verdadero
 - b. Falso
- 98. El conector y el estereotipo para un artefacto mostrado fuera de un nodo es
 - a. de realización y manifestar.
 - b. de dependencia y desplegar.
 - c. de asociación y desplegar.
 - d. de dependencia y manifestar.
- 99. Cuando se muestra un artefacto conectado a un componente, ¿cuál estereotipo se aplica?
 - a. «desplegar»
 - b. «usar»
 - c. «manifestar»
 - d. «extiende»
- 100. ¿Cuál conector se usa para mostrar comunicación entre los nodos?
 - a. De dependencia
 - b. De generalización
 - c. De asociación
 - d. De vínculo

Respuestas

- | | | |
|------|-------|-------|
| 1. b | 7. b | 13. a |
| 2. b | 8. b | 14. b |
| 3. b | 9. c | 15. b |
| 4. c | 10. b | 16. b |
| 5. a | 11. c | 17. c |
| 6. b | 12. a | 18. b |

- | | | |
|-----------|--------------|-----------|
| 19. b | 47. c | 75. c |
| 20. a | 48. ? | 76. a |
| 21. b | 49. c | 77. a |
| 22. a y d | 50. c | 78. b |
| 23. b | 51. a | 79. a |
| 24. a | 52. b | 80. b |
| 25. a | 53. b y c | 81. b |
| 26. a | 54. b | 82. b |
| 27. a | 55. c | 83. b |
| 28. b | 56. b | 84. a |
| 29. c | 57. b | 85. a |
| 30. b | 58. b | 86. b |
| 31. d | 59. a y b | 87. b |
| 32. b | 60. a | 88. b |
| 33. a | 61. d | 89. b |
| 34. b | 62. b | 90. b |
| 35. c | 63. c | 91. b |
| 36. c | 64. a | 92. d |
| 37. b | 65. c | 93. a y c |
| 38. c | 66. d | 94. b |
| 39. d | 67. a | 95. a |
| 40. b | 68. a | 96. b |
| 41. a | 69. c | 97. b |
| 42. a | 70. b | 98. b |
| 43. b | 71. b | 99. c |
| 44. b | 72. b | 100. c |
| 45. d | 73. b, c y d | |
| 46. a | 74. a | |



BIBLIOGRAFÍA SELECCIONADA

- Ambler, Scott. *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3a. ed. New York: Wiley, 2004.
- Booch, Grady. *Object Solutions*. Reading, MA: Addison-Wesley, 2005.
- Booch, Grady, Ivar Jacobson y James Rumbaugh. *The Unified Modeling Language*, 2a. ed. Reading, MA: Addison-Wesley, 2005.
- Eriksson, Hans-Erik, Magnus Penker, Brian Lyons y David Fado. *UML 2 Toolkit*. Indianapolis: Wiley, 2004.
- Fowler, Martin. *UML Distilled Third Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 2004.
- Love, John F. *McDonald's: Behind the Arches*. New York: Bantam, 1995.
- Övergaard, Gunnar y Karen Palmkvist. *Use Cases: Patterns and Blueprints*. Reading, MA: Addison-Wesley, 2005.

ÍNDICE



Símbolos

- símbolo (menos), 107
- # símbolo (número), 107
- + símbolo (más), 107
- « y » (comillas angulares), 24

A

- A Pattern Language* (Christopher Alexander), 189
- Acción, nodos de (diagramas de actividades), 52, 53, 56-62
 - adición de condiciones previas/condiciones posteriores, 58-62
 - modelado de las subactividades, 62
 - nombramiento de las acciones, 57-58
- Aceptar, señal de (diagramas de actividades), 67
- Activación de las líneas de vida, 84-85
- Actividad, nodo final de (diagramas de actividades), 71-72
- Actividad, particiones de la (*V. Carriles* [diagramas de actividades])
- Actividades (diagramas de esquemas de estados), 165
- Actividades comunes, 161
- Actividades de hacer, 161-162
- Actividades internas (diagramas de esquemas de estados), 163

- Actor, símbolos de, 7, 21
 - definición, 36-39
 - Esaw
 - función de los, 18
 - líneas de vida fijas a los, 83, 84
- Agile, proceso, 14, 92
- Agregación, relaciones de, 112-114, 143 (*V. también* Composición, relaciones de)
- Alexander, Christopher, 189
- Ambler, Scott, 176
- Anotaciones:
 - diagramas de casos de uso, 27-28 (*V. también* Documentación [diagramas de casos de uso])
 - diagramas de clases, 118
- Antepasado (como término), 132
- Arreglos:
 - de asociaciones, 145-147
 - de atributos, 110, 111
- Artefactos (diagramas de despliegue), 201-204, 205
- Asociación calificada, 147
- Asociación dirigida, 146, 151
- Asociación dirigida/direccional, 113
- Asociación, relaciones de, 145-150
 - casos de uso, 22
 - diagramas de clases, 108-109, 112, 113

Atributos, 103

- arreglos y multiplicidad, 110, 111
- con asociación, 108-109
- declaración de, 107-108, 107-109
- decoración de los, 106-107
- en un símbolo de clasificador, 106-107
- primitivos como, 121
- unicidad de los, 110-111
- uso de, 107-111

B

- Babbage, Charles, 4
- Balancing Agility and Discipline* (Barry Boehm), 14
- Behind the Golden Arches*, 73
- Bifurcación, nodos de (diagramas de actividades), 56, 63
- Boehm, Barry, 14
- Booch, Grady, 4, 187
- Broccoli, Albert, 136
- Bucle, marco de, 89-90
- Búsqueda, motor de, 92-93

C

- C++, 4
- Cambio, eventos de, 164
- Campos, 107, 145
- Característica(s):
 - como procesos, 48
 - de clases, 103
 - significado del término, 106-107
 - símbolos de identificación de, 107
- Carriles (diagramas de actividades), 63-68
- Casos de uso de diagramas múltiples, 39-43
- Casos de uso, diagramas de (casos de uso), 7, 17-44
 - adición de documentación de soporte a los, 28-29
 - anotación de los, 27-32
 - como listas de *cosas por hacer*, 19
 - comunicación con los, 20

- conectores en los, 22-25
- creación de los, 32-34
- decisión sobre el número de, 34
- definición de los actores en los, 36-39
- división de los, en múltiples diagramas, 39-43
- ejemplo usando los, 34-43
- escenarios de éxito y falla en los, 92
- establecimiento de prioridades de las capacidades con los, 19-20
- finalidad de los, 17
- impulsión del diseño con, 43-44
- inserción de notas en los, 27-28
- objetivo de los, 42
- sencillez de los, 18-19
- sencillez engañosa de las, 18-19
- símbolo de caso de uso, 21
- símbolos de actor en los, 21
- texto con los, 13, 19
- uso de directrices para documentar, 29-32
- y la documentación de sus ideas, 42
- Casos de uso, óvalos de los, 7, 18
- Casos de uso, símbolos de (en los diagramas de casos de uso), 21
- Chitty Chitty Bang Bang* (película), 136-137
- CLAB (crear, leer, actualizar y borrar), comportamiento, 51
- Clase base, 114
- Clases asociación, 146-150
 - calificadas, 147
 - dirigidas, 146, 151
- Clases entidades, 124-127
- Clases fronteras, 124-125, 128
- Clases, diagramas de, 8-9, 101-128
 - adición de detalles a las clases, 153
 - adición de operaciones a las clases, 111-112
 - atributos, 107-111
 - características, 103
 - clases de control, 127
 - clases entidades, 126-127

- clases fronteras, 128
- clasificador, 103-107
- comentarios, 118
- decoración de las clases, 106-107
- enumeraciones, 121-122
- espacios de nombres, 122-123
- estereotipos, 117
- identificación de las clases necesarias, 123-128
- interfaces, 104-105
- metaclases, 105-106
- notas, 118-119
- objetos, diagramas de, 104
- paquetes, 118
- primitivos, 120-121
- relaciones en los, 111, 113-117
- restricciones, 118-120
- tipos de datos, 105
- tipos genéricos, 105
- tipos parametrizados, 105
- Clases:
 - adición de detalles a las, 153
 - asociación, 146-150
 - características de las, 103
 - clasificadores, 103-107
 - de control, 124, 125, 127
 - decoración de las, 106
 - descubrimiento de las, 102
 - dominio, 176
 - en los diagramas de clases, 8-9
 - entidades, 124-127
 - fronteras, 124-125, 128
 - línea de vida que representa, 84
 - metaclases, 105-106
 - relaciones entre (V. Relaciones)
- Clasificación dinámica, 133-136
- Clasificación, algoritmos de, 105
- Clasificación:
 - dinámica, 133-136
 - generalización en comparación con, 132-133
- Clasificador, papel del (diagramas de colaboración), 94
- Clasificadores, 103-107
 - atributos en los, 106-107
 - operaciones en los, 111
 - paleta de caramelo, media y completa, 179
- Clavijas (en los diagramas de actividades), 55-56
- Clientes, comunicación con los, 20
- Cobol, 4
- Codificación, iniciación de la, 96-97
- Colaboración (comunicación), diagramas de, 9, 11, 82, 94-95
- Comentarios (diagramas de clases), 118
- Comillas angulares (« y »), 24
- Compleción, transición de, 164
- Componentes, 176
- Componentes, diagramas de, 11-12, 175-182
 - especificación de las interfaces, 179-180
 - método de diseño de abajo hacia arriba, 178
 - método de diseño de arriba hacia abajo, 177
 - para los consumidores, 180-181
 - para los productores, 182-183
- Comportamiento de estado, patrón de, 133-136, 190-191
- Comportamiento, máquinas de estados de, 159, 166-167
- Comportamientos, 103
- Composición, relaciones de, 112-114, 143-145
- Computadoras, historia de las, 4
- Comunicación, diagramas de (V. Colaboración, diagramas de)
- Comunicación, trayectorias de (diagramas de despliegue), 204-205
- Condiciones guardianes:
 - diagramas de actividades, 52-54, 58-62
 - diagramas de esquemas de estados, 164-165
 - diagramas de secuencia, 90
- Condiciones posteriores (diagramas de actividades), 56, 58-62

Condiciones previas (diagramas de actividades), 56, 58-62

Conectores:

- diagramas de casos de uso, 22-25
- diagramas de clases, 112-117
- diagramas de colaboración, 94
- diagramas de despliegue, 204
- estereotipos asociados con los, 117
- paleta de caramelo, media y completa, 179

Conexión, estado de, 159-160

Conmutador anidado, declaración de, 169

Consumidores, diagramas de componentes para los, 180-181

Control, clases de, 124-125, 127

Control, flujo de (diagramas de actividades), 52-53

Controlador de visión del modelo (MVC), 127

CRC, fichas (*V.* Responsabilidad y colaborador de las clases, fichas de)

Creación, patrones de, 133

Crear, leer, actualizar y borrar (CLAB), comportamiento, 51

D

Datos, tipos de, 105

De arriba hacia abajo, procedimiento de diseño de, 177

Decisión, diamantes de (diagramas de flujo), 56, 62-63

Decisión, nodos de (diagramas de actividades), 56, 62-63

Decoración de las clases, 106-107

Dependencia, relaciones de, 150-152

- casos de uso, 22, 25-27, 32
- diagramas de clases, 112, 116-117
- estereotipos ampliar, 26-27
- estereotipos incluir, 25-26
- estereotipos para las, 151-152
- inserción de referencias a las, 32

Desarrollo del software, llevar al exterior el, 5

Design Patterns (Erich Gamma), 97, 127, 133, 171

Despliegue de aplicaciones vivientes, entorno de, 205

Diagramas de actividades, 7-8, 47-77

- acciones que se extienden sobre las particiones, 67

- bifurcaciones, 63

- carriles, 63-68

- condiciones guardianes, 52-54

- condiciones previas y condiciones posteriores, 58-62

- creación de los, 72-73

- determinación del número de, 77

- diagramas de flujo en comparación con los, 48, 51

- ejemplos de, 51

- en la reingeniería de procesos, 73-76

- flujo de control, 52-53

- manera de mostrar las excepciones en los, 70-71

- manera de mostrar los flujos en los, 54-56
- meta de los, 47

- nodo inicial, 52

- nodos de acción, 56-62

- nodos de decisión, 62-63

- nodos de fusión, 62-64

- nombramiento de las acciones, 57-58

- parámetros de entrada, 70

- partición de la responsabilidad, 63-68

- particiones multidimensionales, 67-68

- señal de tiempo, 67-69

- subactividades, 62

- terminación de los, 71-72

- uniones, 63

- usos de los, 48-51

Diagramas de flujo:

- diagramas de actividades en comparación con los, 48, 51

- diamantes de decisión en los, 62

Diagramas estáticos, 101, 131 (*V. también* Clases, diagramas de)

- Diagramas redundantes, 81-82
- Diagramas, 7-12
- cuándo crear los, 12
 - de actividades, 7-8
 - de casos de uso, 7
 - de clases, 8-9
 - de componentes, 11-12
 - de estados, 10-11
 - de interacción, 9-10
 - de topología del despliegue, 9-10
 - elección de los, 188
 - tamaño y complejidad de los, 13
 - texto que suplementa los, 13
- Directrices (como documentación de los casos de uso), 29-32
- Diseño impulsado por los casos de uso, 43-44
- Diseño por contrato, 58-59
- Diseño, patrones de, 127, 133-134
- Diseño:
- de abajo hacia arriba, 178
 - de arriba hacia abajo, 177
 - impulsado por casos de uso, 43-44
- Disparadores, 164, 165
- Documentación (diagramas de casos de uso), 13, 28-32, 42
- de necesidades primarias y secundarias, 20
 - directrices para la, 29-32
 - formas de la, 28
 - notas como, 27-28
- Documentación (en general):
- (V. *también* Comentarios; Notas)
 - cantidad de, 187-188
 - con los modelos, 192-193
 - edición de la, 188
- Dominio, clases del, 176
- Dominio, expertos del, 49, 125
-
- E**
- Eckert, Presper, 4
- Edición de la documentación, 188
- Enumeraciones (diagramas de clases), 121-122
- Es un(a)*, relaciones de, 114-115, 132-133
- Esaw, 1-2, 10-11
- Espacios de nombres (diagramas de clases), 118, 122-123
- Especialización (equipos para desarrollo de software), 189
- Esquemas de numeración anidada, 95
- Establecimiento de prioridades de las capacidades, con los diagramas de casos de uso, 19-20
- Estado final, 159
- Estado inicial, 159
- Estados activos, 161-162
- Estados compuestos, 162-163
- Estados inactivos, 161
- Estados no ortogonales, 162
- Estados ortogonales, 162-163
- Estados simples, 162
- Estados, esquemas de, diagramas de (diagramas de estados/máquinas de estados), 10-11, 157-171
- actividades internas, 163
 - estado de conexión, 159-160
 - estado de historia profunda, 160-161
 - estado de historia superficial, 160-161
 - estado de selección, 159, 160
 - estado de terminación, 159
 - estado final, 159
 - estado inicial, 159
 - estados activos/inactivos, 161-162
 - estados de historia, 160-161
 - estados simples/compuestos, 162-163
 - implementación de los, 168-171
 - máquinas de estados de comportamiento, 166-167
 - máquinas de estados de protocolo, 167-168
 - símbolos para los, 158-159
 - transiciones, 164-166
 - vinculación con las submáquinas, 163
- Estados, patrones de, 169, 171, 189-191
- Estados, símbolo de, 158

Estereotipos, 24-25
 diagramas de clases, 117
 extender, 26-27
 incluir, 25-26
 para dependencias, 151-152
 tipos de datos, 105
 Estímulo (V. Control, flujo de)
 Eventos temporizadores, 164
 Excepciones (diagramas de actividades),
 70-71
 Excepciones, manejador de (diagramas de
 actividades), 70-71
 Extensión, casos de uso de, 25-27
 eXtreme Programming (XP), 6, 14

F

Fleming, Ian, 136
 Flujo (V. Control, flujo de)
 Flujo, nodo final del (diagramas de actividades),
 71-72
 Fowler, Martin, 97, 187, 192
 Fragmentos combinados (V. Interacciones,
 marcos de)
 Fuente (del conector), 113, 150
 Fusión, nodos de (diagramas de actividades),
 62-64

G

Gamma, Erich, 97, 127, 133, 171, 187
 Gates, Bill, 3
 Generalización, relaciones de:
 (V. también Herencia, relaciones de)
 casos de uso, 22-23
 diagramas de clases, 112, 114, 115
 Genéricos, 105
 Gramática, 102-103

H

Herencia de interfaz, 139-143
 Herencia múltiple, 135-138
 Herencia simple, 132

Herencia, 104-105
 múltiple, 135-138
 simple, 132, 135
 Herencia, relaciones de, 132-143
 (V. también Generalización, relaciones de)
 diagramas de clases, 112, 114-115
 herencia de interfaz, 139-143
 herencia múltiple, 135-138
 herencia simple, 132
 patrón de comportamiento de estado,
 133-135

Hijo (como término), 114, 132
 Historia profunda, estado de, 160-161
 Historia superficial, estado de, 160-161
 Historia, estados de, 160-161
 Hopper, Grace, 4
 Hornos de microondas, 161

I

Idiomas, 10
 Inclusión, caso de uso de, 25-26
 Ingeniería automovilística, 4
 Interacción, diagramas de, 9-10, 81-97
 diagramas de colaboración (comunicación),
 82, 94-95
 diagramas de colaboración, 9-11
 diagramas de secuencia, 9-10, 82-94
 y escritura del código, 96-97
 Interacción, marco alternativo de, 90-91
 Interacciones, marcos de (fragmentos
 combinados), 87-91
 Interfaces proporcionadas, 141, 179
 Interfaces requeridas, 141, 179, 180
 Interfaces:
 diagramas de clases, 104-105
 implementación de las, 142
 proporcionadas, 141, 179
 requeridas, 141, 179-180
 Interfaz, herencia de, 139-143
 (V. también Generalización, relaciones de)
 interfaces proporcionadas, 141

interfaces requeridas, 141
 modelado en pizarrón blanco, 139-140
 reglas para la, 141-143

J

Jacobson, Ivar, 4, 187

L

Lectura-escritura, comportamiento de, 51
 Líneas de vida (diagramas de secuencia), 83-84
 activación de las, 84-85
 escalonamiento de las, 91
 Listas de *cosas por hacer*, diagramas de casos de
 uso como, 19
 Llamada anidada, 85
 Llamada, eventos de, 164

M

Macrofase (modelado), 111
 Macroprocedimiento, 19, 97
 Mansfield, Richard, 124
 Más (+), símbolo, 107
 Mauchly, John, 4
 McCarthy, James, 188
 McDonald's, 73
 Menos (-), símbolo, 107
 Mensajes (diagramas de secuencia):
 definición, 85
 descubrimiento de los, 92-94
 envío de, 85-87
 hallados, 85
 perdidos, 85
 Mensajes hallados, 85
 Mensajes perdidos, 85
 Metaclases, 105-106
 Metadatos, 106
 Métodos:
 comportamientos como, 103
 decoración de los, 106-107
 descubrimiento de los, 102
 uso del término, 111

Microfase (modelado), 111
 Microprocedimiento, 19, 187
 Microsoft:
 SOA, 14
 y el costo del software, 6
 Modelado, herramientas para el, 5-6, 13
 Modelado, lenguajes de:
 desarrollo de los, 4
 proceso en comparación con los, 14
 Modelado:
 (V. también los temas específicos)
 actividades primarias asociadas con el, 139
 expertos en, 187
 los hacer y los no hacer para el, 186-189
 macrofases y microfases en el, 111
 meta del, 10
 razones para el, 5
 uso de patrones conocidos de estados,
 189-191
 y el desarrollo futuro del software, 5
 Modelos:
 adición de documentación a los, 192-193
 definición de, 2
 evaluación de la compleción del, 12
 notas en los, 118
 refactorización, 192
 texto que suplementa a los, 13
 uso de los, 6
 validación de los, 193
 valor de los, 2
 Motown-jobs.com (ejemplo), 34-44
 búsqueda del diseño para, 92-94
 condiciones guardianes, 53-54
 definición de los actores, 36-39
 diagrama de secuencia para, 86
 división en diagramas múltiples, 39-43
 MSDN, 6
 Multiplicidad:
 atributos, 110
 conectores, 24
 MVC (controlador de visión del modelo), 127

N

Navegación, 113
Nieto (como término), 132
Nodo inicial (diagramas de actividades), 52
Nodos (diagramas de despliegue), 198-200
Nodos conectores (diagramas de actividades), 54-55
Nombramiento de las acciones (diagramas de actividades), 57-58
Notas:
 diagramas de casos de uso, 27-28
 (V. *también* Documentación [diagramas de casos de uso])
 diagramas de clases, 118-119
 diagramas de secuencia, 87-88
Numeración, esquemas de, 95
Número (#), símbolo, 107

O

Object Constraint Language (OCL), 87, 118
Object Management Group (OMG), 3
Objetivo (del conector), 150-151
Objetos, diagramas de, 104
Objetos, líneas de vida de (diagramas de secuencia), 83-84, 84-85
Objetos:
 Descubrimiento de, con los diagramas de secuencia, 92-94
 En los diagramas de actividades, 55
OCL (V. Object Constraint Language)
OMG (Object Management Group), 3
Opción, efecto (diagramas de esquemas de estados), 165-166
Opdike, William, 97
Operaciones, 111
Orientados a objetos, análisis y diseño:
 principio básico del, 48
 reto para el, 124
 soporte UML para el, 17
Outsourcing del desarrollo de software, 5
Óvalos (V. Casos de uso, óvalos de los)

P

Padre (como término), 114
Paleta de caramelo, completa, 179
Paleta de caramelo, media, 179
Papel, especialización del, 93
Paquete, símbolo de, 118
Particiones (V. Carriles [diagramas de actividades])
Particiones multidimensionales de la actividad, 67-68
Patrones estructurales, 133
Plantillas (C++)
Polimorfismo, 115, 132
Posibilidad de ser sustituido, 115
Primitivos (diagramas de clases), 120-121
Problema, dominio del, 48
Proceso unificado de modelado, 4
Proceso(s):
 características como, 48
 lenguajes de modelado en comparación con el, 14
Proceso, reingeniería del, 73
Productores, diagramas de componentes para, 182-183
Propiedad, 107
Protocolo, máquinas de estado de, 159-160, 167-168

R

Radar, estufa de, 161
Rational Unified Process (RUP), 14, 92
Realización, relaciones de:
 diagramas de clases, 112, 115-116
Refactoring (Martin Fowler), 97, 192
Refactorización, 97, 192
Reingeniería del proceso, 73-76
Relaciones:
 de agregación, 112-114, 143
 de asociación, 22, 108-109, 112-113, 145-150
 de composición, 112-114, 143-145

de dependencia, 25-27, 32, 112, 116-117, 150-152
 de *es un(a)*, 114-115, 132-133
 de generalización, 22-23, 112, 114-115
 de herencia, 112, 114-115, 132-143
 de realización, 112, 115-116
 en los casos de uso, 22-23, 25-27, 32
 en los diagramas de clases, 111, 113-117
 Responsabilidad y colaborador de las clases (CRC), fichas de, 125-126
 Restricciones:
 diagramas de clases, 118-120
 diagramas de secuencia, 87
 Retroalimentación, 13
 Reutilización de los diagramas, 163
 Roomba®, aspiradora, 33
 Rose XDE, 6
 Rumbaugh, James, 4, 187
 RUP (*V. Rational Unified Process*)

S

Secuencia, diagramas de, 9-10, 82-94
 activación de las líneas de vida, 84-85
 descubrimiento de objetos/mensajes con los, 92-94
 envío de mensajes, 85-87
 líneas de vida de objetos, 83-84
 marcos de interacciones, 87-91
 notas, 87-88
 restricciones, 87
 utilidad de los, 91-92
 Selección, estado de, 159-160
 Señal a ruido, razón baja, 125
 Señal a ruido, razón, 125
 Señal de enviar (diagramas de actividades), 67, 69
 Señal, 152
 Señal, eventos de 164
 Service Oriented Architecture (SOA), 14
 Smalltalk, 4
 SOA (Service Oriented Architecture), 14

Sobrecarga de operadores, 121
 Software, diseño de:
 (*V. también Modelado*)
 complejidad del, 186
 evolución del, 3-5
 Solución, dominio de la, 48
 Spencer, Percy, 161
 Stroustrup, Bjarne, 4, 152
 Subactividad(es):
 en los diagramas de actividades, 62
 reingeniería, 74-76
 Subclase, 114
 Submáquinas, vinculación con las, 163
 Superclase, 114
 Superestado (*V. Estados compuestos*)

T

Tabla de estados, 169
 Terminación, estado de, 159
 Texto (diagramas de casos de uso), 13, 19, 28, 32
 (*V. también Documentación [diagramas de casos de uso]*)
The C++ Programming Language (Gjarne, Stroustrup), 152
The Object Primer (Scott Ambler), 176
 Tiempo, señales de (diagramas de actividades), 67-69
 Tipos parametrizados, 105
 Together, 6
 Topología del despliegue, diagramas de, 12, 197-205
 artefactos, manera de mostrar en los, 201-205
 nodos en los, 198-200
 trayectorias de comunicación, 204-205
 Transición sin disparador, 164
 Transiciones (diagramas de esquemas de estados), 158, 164-166
 Trazo de diagramas sobre pizarrón blanco, 139-140
 Turing, Aolan, 4

U

Unified Modeling Language (UML), 2
 como un lenguaje, 2-3, 82
 comunicación precisa en el, 102
 desarrollo del, 4
 descomposición/recomposición de
 problemas con, 49
 gramática del, 103
 y evolución del diseño de software, 3-5
Unified Process, 14
Uniform Resource Locators (URL), 28
Unión, nodos de (diagramas de actividades), 63
Usuarios, comunicación con los, 20

V

Validación de los modelos, 193
Visio, 3, 6
 adición de documentación, 29

 estados compuestos, 162
 estereotipo extender, 25
 marco de interacción, 89
 nodos conectores con, 54-55
 simulación de la señal de tiempo, 68
 simulación de partición multidimensional,
 67
 subactividades, 62
 vinculación con las submáquinas, 163
 y la mitad de una paleta de caramelo, 179

X

XP (V. eXtreme Programming)

