

¿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-

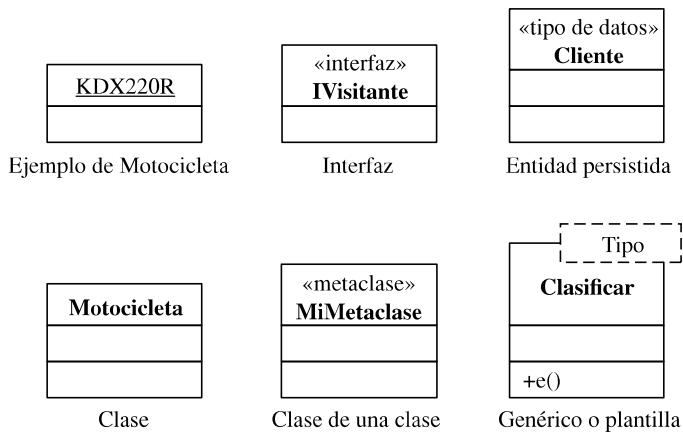


Figura 5-1 Ejemplos de clasificadores en el UML.

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 un 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 modeló 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 metaclasses

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 1º 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 atri-

Motocicleta
–motor
+ObtenerSalidaDePotencia()

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”.

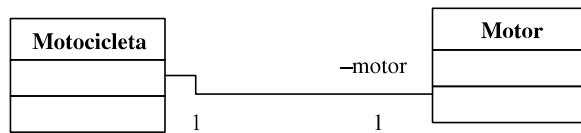


Figura 5-3 Manera de mostrar el atributo “motor” usando una asociación.

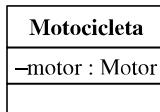


Figura 5-4 Esta figura transmite una información idéntica a la que se muestra en la figura 5-4; es decir, una motocicleta contiene un motor cuyo tipo es “Motor”.

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 Mencioné 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.

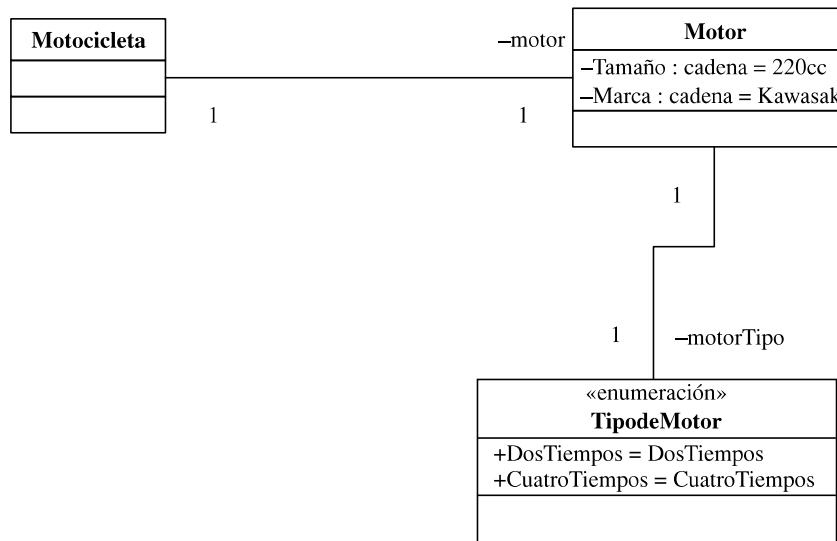


Figura 5-5 Este diagrama de clases contiene más información acerca del motor de la motocicleta al usar un atributo de asociación para el motor y un segundo atributo de asociación para los tipos posibles de motores.

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 entretenerte 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 signaturas de operaciones usadas para implementar la clase “Mano” (“Hand”).

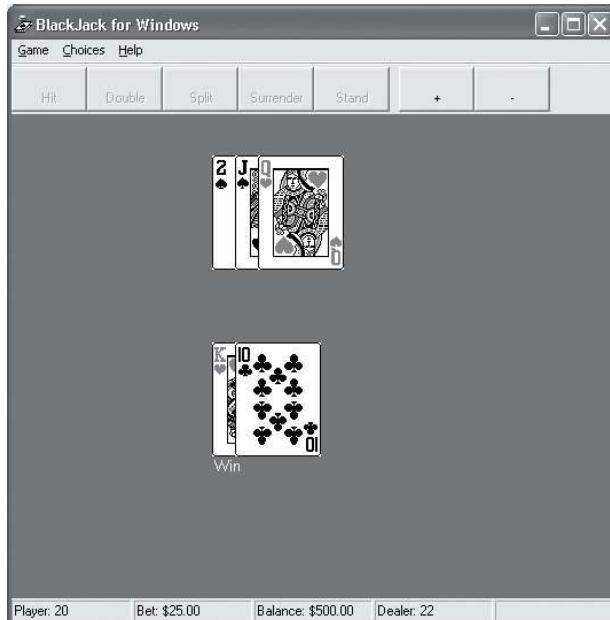


Figura 5-6 El juego “BlackJack for Windows”.

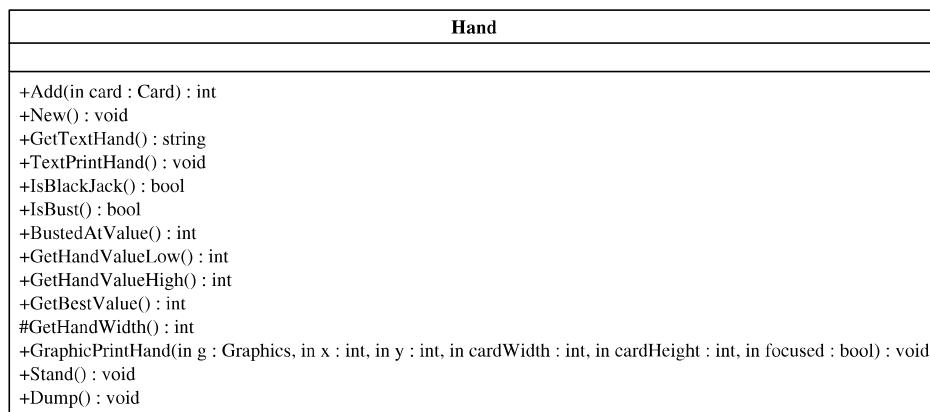


Figura 5-7 Clasificador que muestra varias de las signaturas para la clase “Mano” (“Hand”).

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 *direccional*. El extremo con la flecha es el objetivo o el objeto hacia el que se puede navegar. El extremo sin la flecha se llama *fuent*e. *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 sintáctico 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.

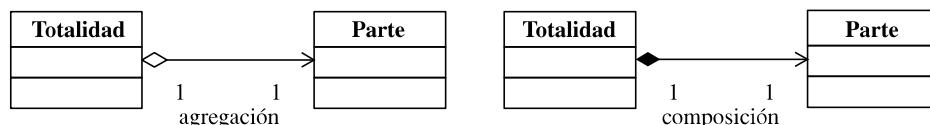


Figura 5-9 La agregaci n es sem anticamente id ntica a la asociaci n, y la composici n significa que la clase compuesta es la \'unica clase que tiene una referencia hacia la clase propietaria.

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 “Currículu”, “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 “Currículu”, “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”, “Currículu” y “Anuncio”. De manera implícita, los miembros privados son parte de “Trabajo”, “Currículu” y “Anuncio”, pero estas clases hijos —y cualesquiera otras 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.)

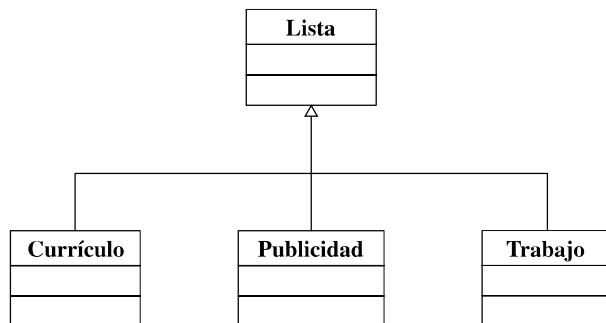


Figura 5-10 Esta figura muestra que “Currículu”, “Trabajo” y “Publicidad” se heredan de “Lista”.

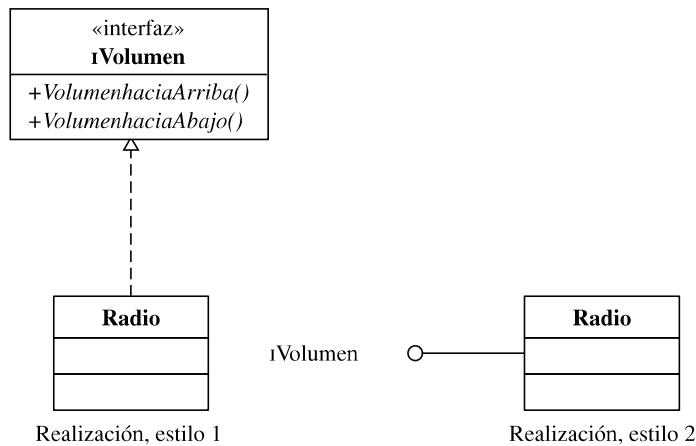


Figura 5-11 La realización, o herencia de interfaz, se puede mostrar en cualquiera de los dos estilos, como se ilustra en la figura.

Para ayudarle a familiarizarse con la herencia de interfaz, agregué un estilo alterno 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.

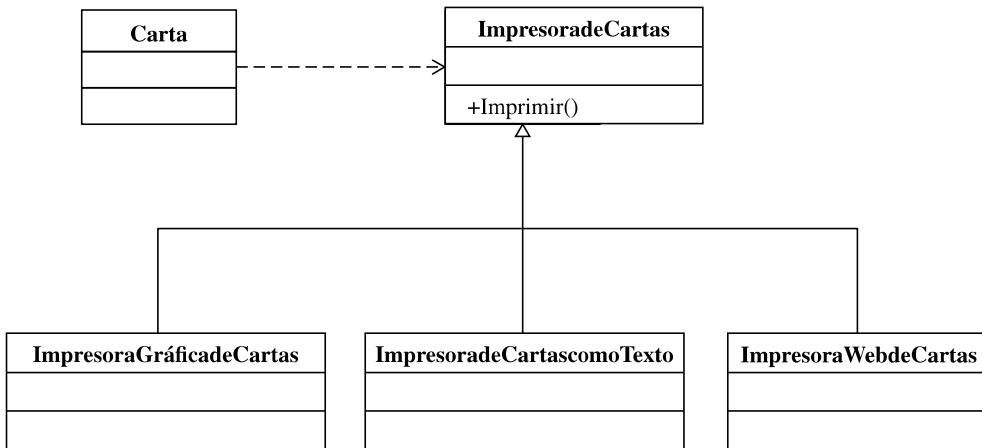


Figura 5-13 La relación de dependencia ahora incluye generalización que muestra tipos específicos de objetos “ImpresoradeCartas”.

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.

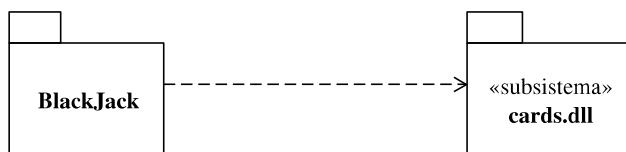


Figura 5-14 El diagrama muestra que el paquete “BlackJack” depende del paquete “cards.dll”, en el cual se usa el estereotipo «subsistema».

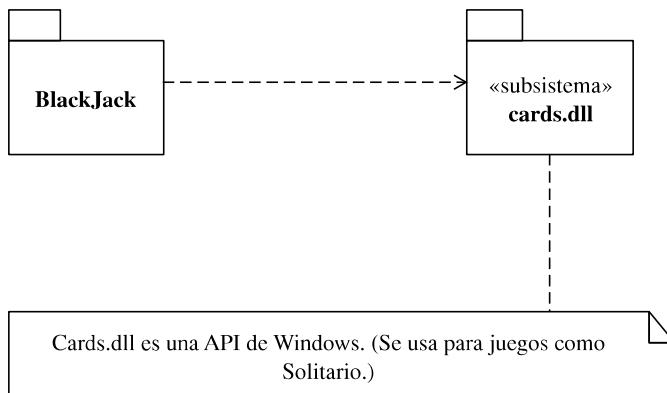


Figura 5-15 Se usa el rectángulo con la punta doblada para agregar notas o comentarios a los elementos de los diagramas UML.

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

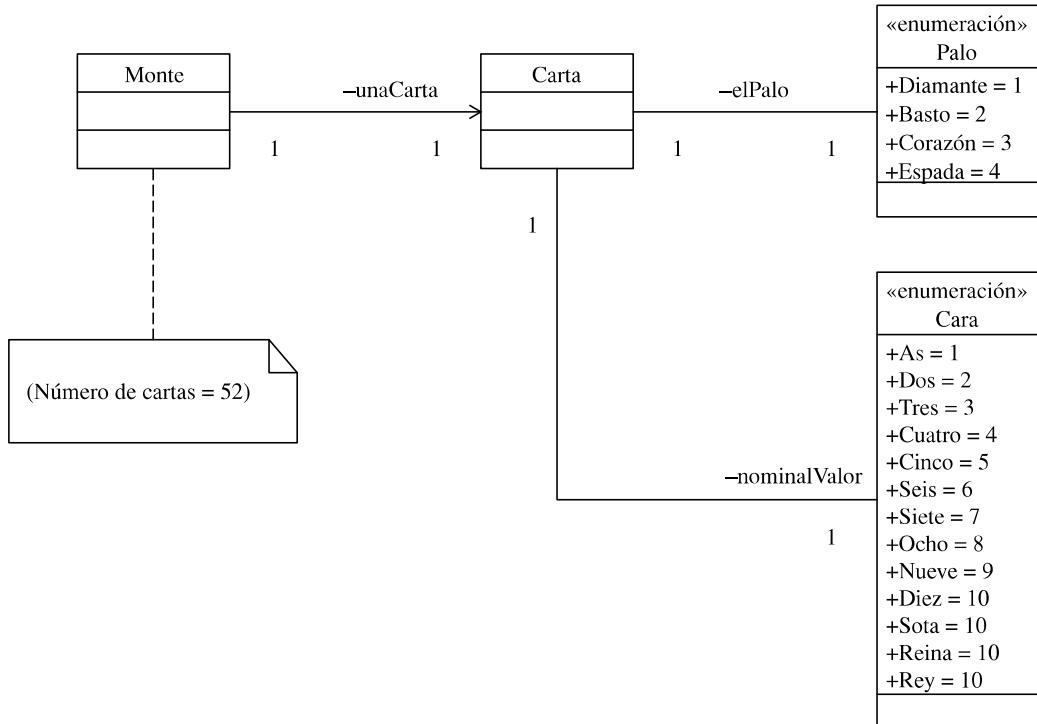


Figura 5-16 En esta figura se ilustra cómo podemos mezclar las restricciones —“Número de cartas = 52” en la figura— con otros elementos del diagrama para aumentar su precisión.

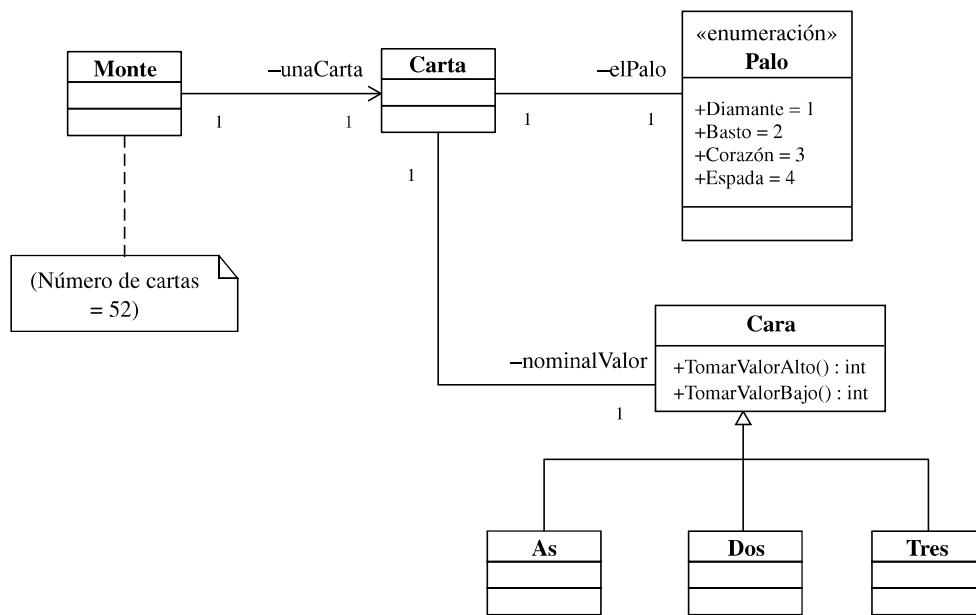


Figura 5-17 Diagrama de clases de la figura 5-16 modificado para captar el hecho de que las cartas pueden tener valores nominales dinámicos (int = entero).

exprese 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 “NaturalIlimitado”) 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.

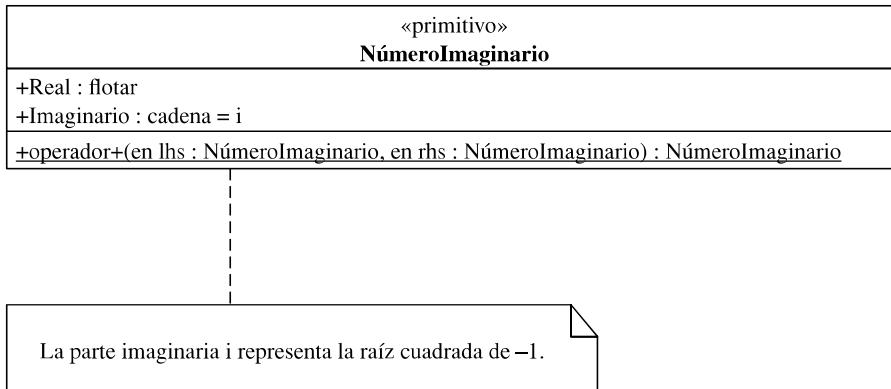


Figura 5-18 Los números imaginarios son números reales multiplicados por el número imaginario i , el cual representa la raíz cuadrada de -1 .

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 mante de cartas de juego, pero una enumeración tipo “Palo” que contiene cuatro valores nombrados transmite más significado (vea la figura 5-17).

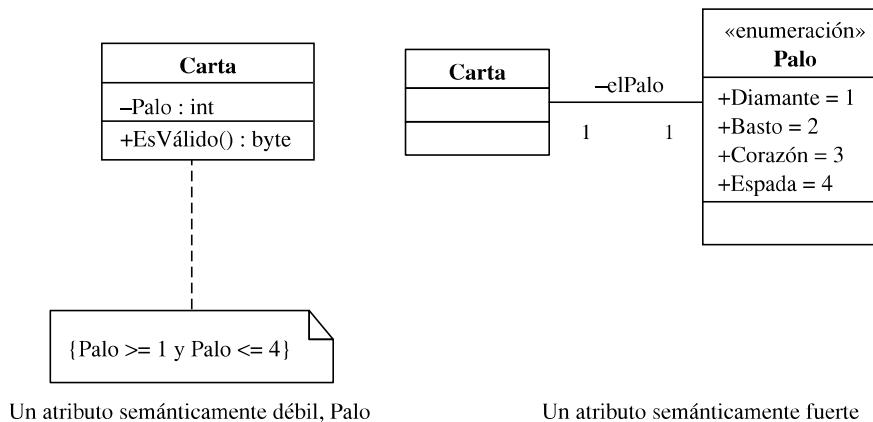


Figura 5-19 El entero “Palo” de la izquierda necesita explicación por el camino de una restricción con el fin de limitar y aclarar los valores posibles del entero, en tanto que la enumeración semánticamente más fuerte del “Palo” que se da a la derecha no necesita esa explicación.

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

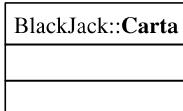


Figura 5-20 A menudo, los paquetes se codifican como espacios de nombres y se muestran en los diagramas UML en el lado izquierdo del operador de resolución de alcance, dos puntos dobles.

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 di 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, enfascarse 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».

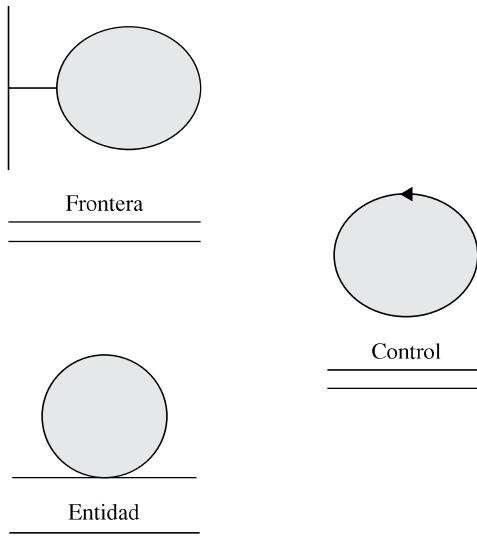


Figura 5-21 Los símbolos rectangulares para las clases y los estereotipos se pueden reemplazar con símbolos que representan específicamente las clases de frontera, control y entidad.

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 (“hackeo”)—. 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, introduce 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#. Esta relación se describe como una *clasificación* de Pablo el 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”.

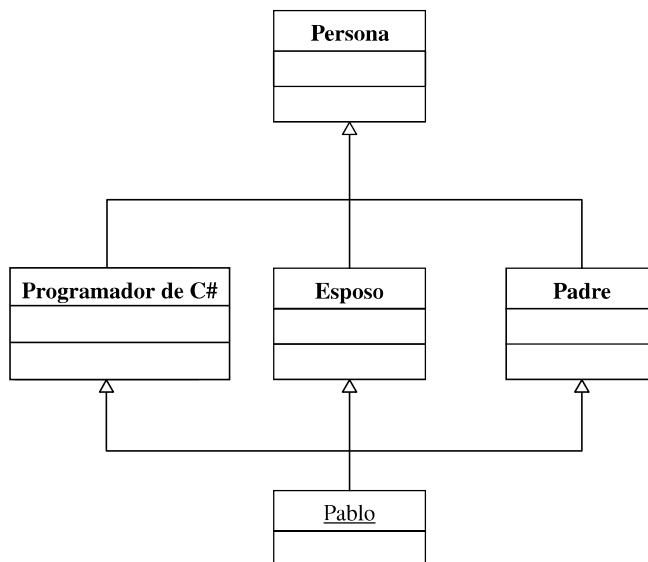


Figura 6-1 Diagrama UML de clases en el que se muestra una generalización rígida donde el objeto “Pablo” intenta reflejar de manera incorrecta “Padre”, “Esposo” y “Programador de C#”.

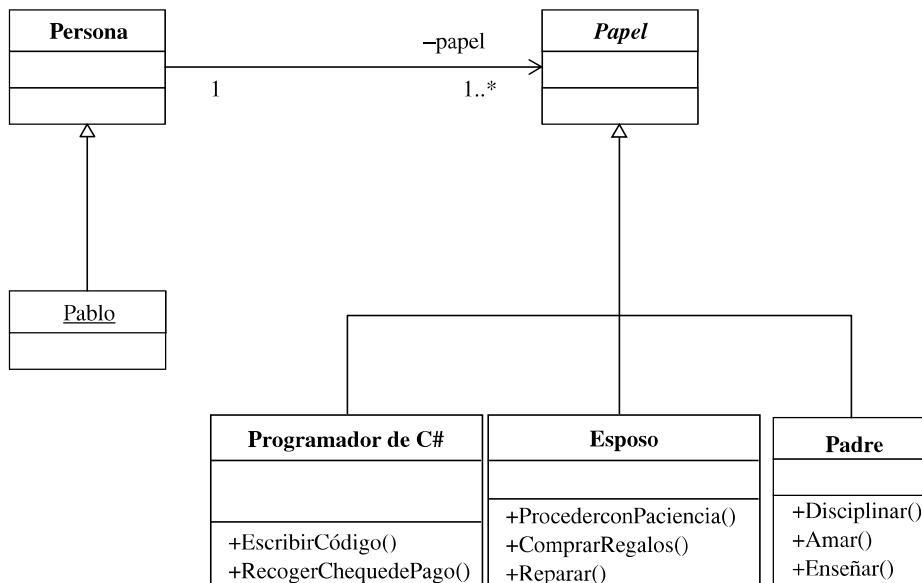


Figura 6-2 Un segundo diagrama UML de clases en el que se usa la asociación para un papel que refleja cómo se comportan las personas en ciertos papeles.

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 a veces 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.

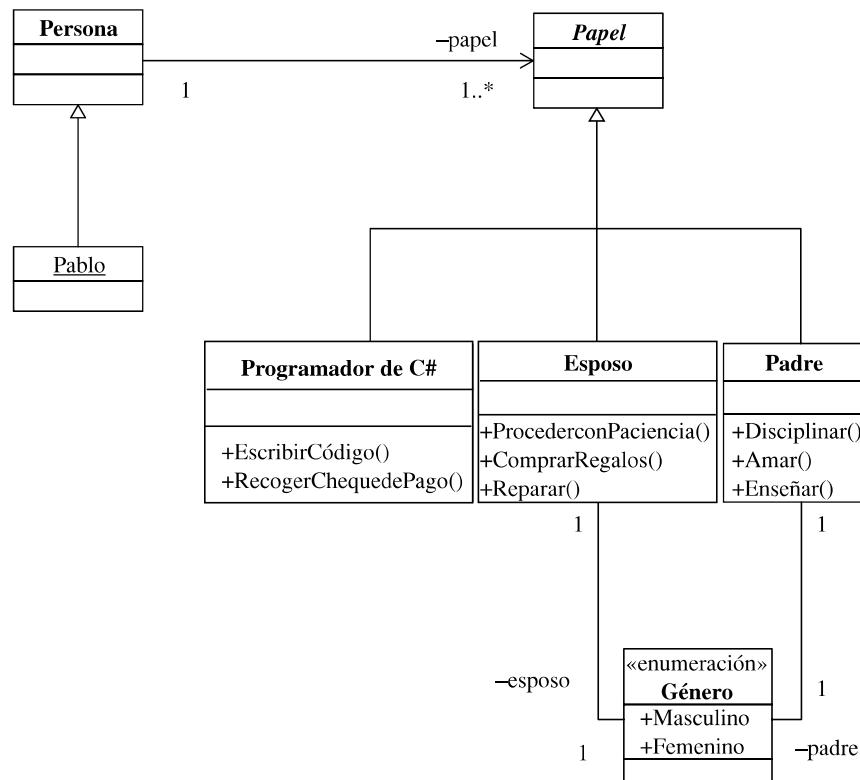


Figura 6-3 Abstracción del género a partir de los papeles de esposo y padre.

La herencia múltiple es incluso más difícil porque todavía tenemos los problemas de generalización y clasificación, y éstos se exacerbán 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

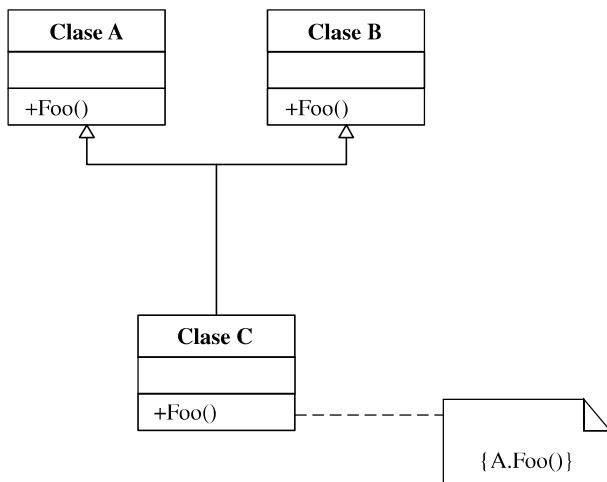


Figura 6-4 Resuelva de manera explícita los conflictos de nombres en las clases con herencia múltiple, lo que se muestra aquí con el uso de una restricción.

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 bote y avión, 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,

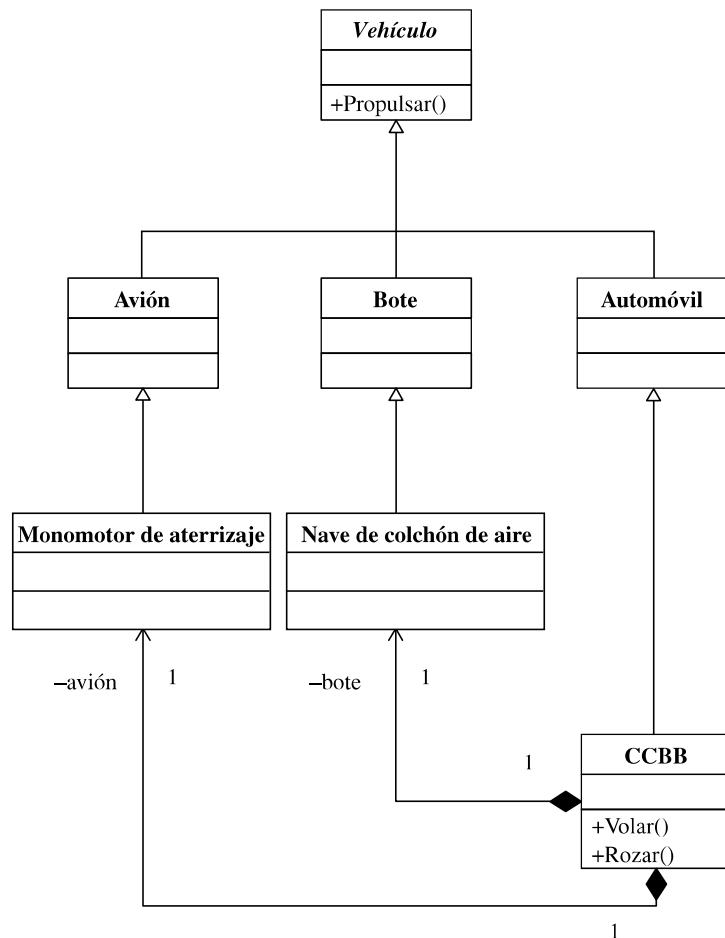


Figura 6-5 En esta figura, mostramos que “CCBB” hereda de “Automóvil”, pero usa la composición para mostrar sus capacidades de bote y avión.

se muestra una manera en que podríamos trazar un diagrama de clases para ilustrar las características aerodinámicas y anfibias 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

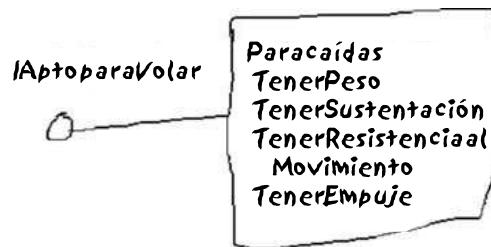


Figura 6-6 Se muestra realizada la interfaz “IAptoparaVolar” por medio de “Paracaídas”, como podríamos dibujarla sobre un pizarrón.

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 caramelos 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

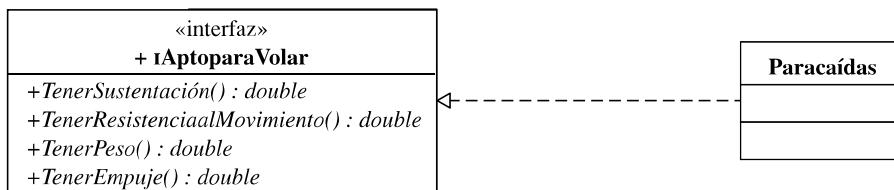


Figura 6-7 El mismo diagrama que se muestra en la figura 6-6, presentado en Visio (double = doble).

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 caramelos 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 caramelos 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.

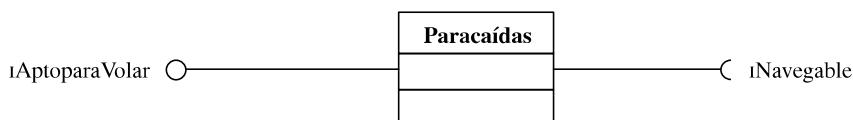


Figura 6-8 “IAptoparaVolar” es una interfaz proporcionada por “Paracaídas”, e “INavegable” muestra una interfaz requerida por “Paracaídas”.

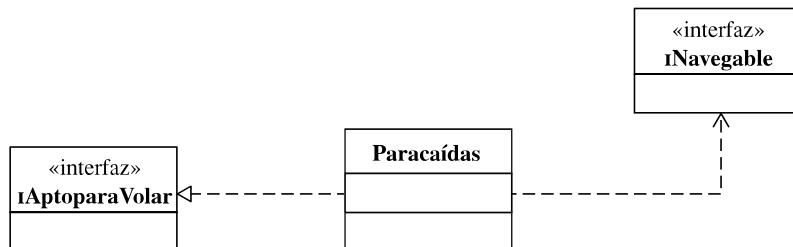


Figura 6-9 En esta figura, se ilustran las mismas relaciones que las descritas en la 6-8; específicamente, que “Paracaídas” realiza “iAptoparaVolar” y depende de “iNavegable”.

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.

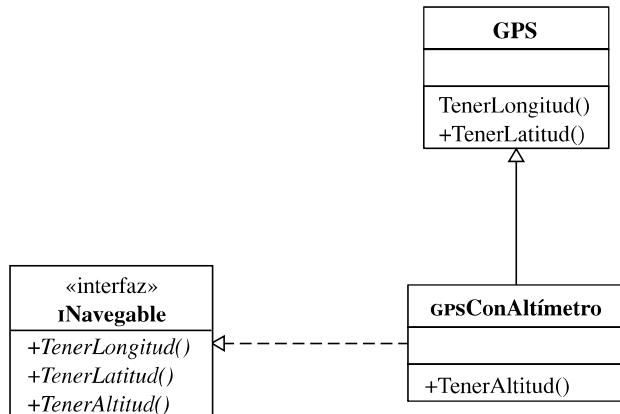


Figura 6-10 Implementación de una interfaz a través de herencia.

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#.

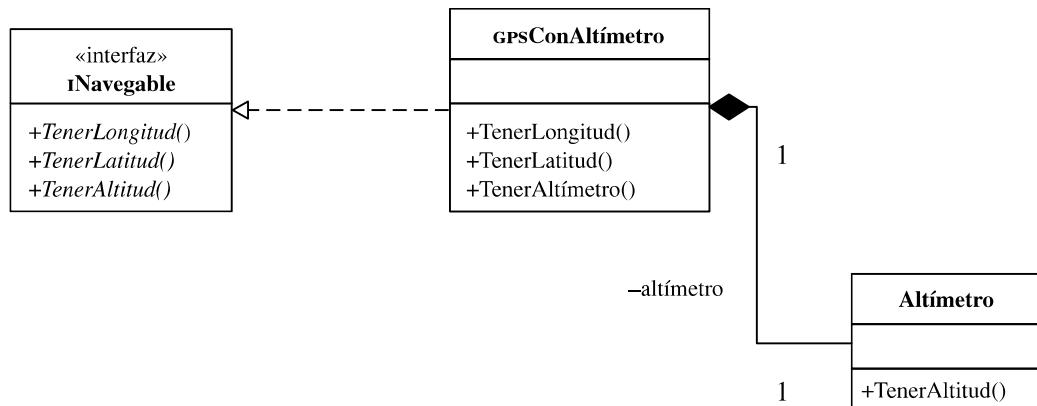


Figura 6-11 Figura 6-10 revisada para usar la composición con el fin de agregar el comportamiento del altímetro.

```

public interface 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, ésa 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 quisieramos 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.

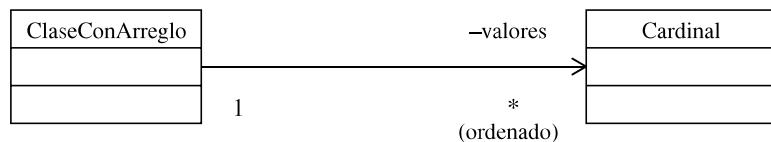


Figura 6-12 Podemos agregar modificadores y detalles a las asociaciones precisamente como las agregaríamos a los campos.

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 quisieramos 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.

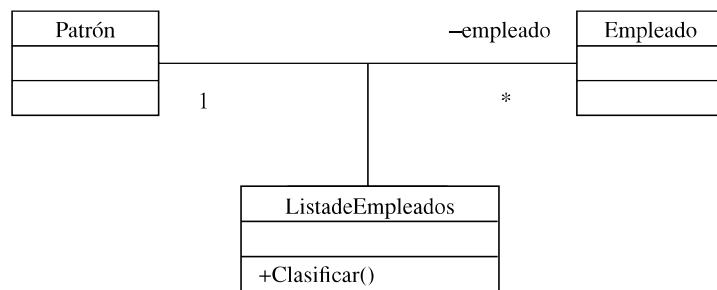


Figura 6-13 Una clase asociación que muestra que la clase “ListadeEmpleados” vincula en forma indirecta “Patrón” con “Empleados”.

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.

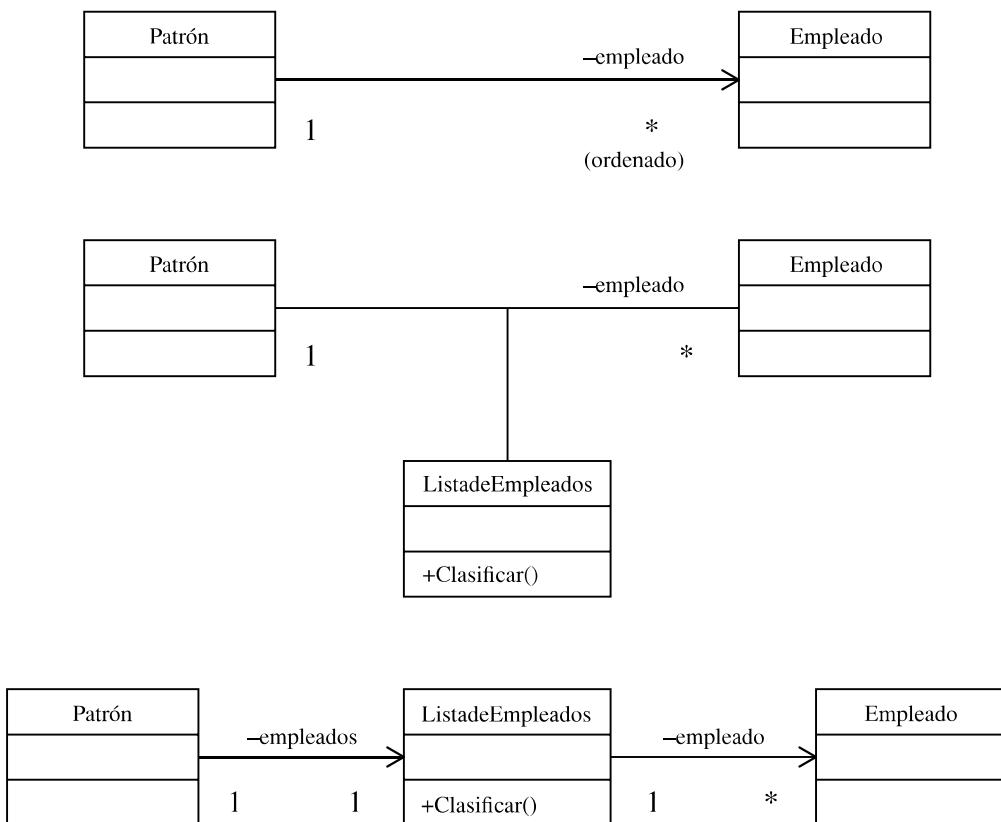


Figura 6-14 Tres variaciones que reflejan una relación uno a muchos entre un “Patrón” y “Empleados”.

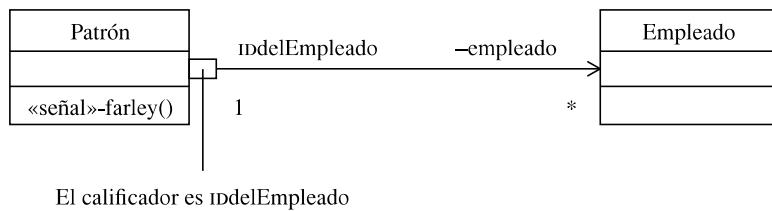


Figura 6-15 El calificador que da como resultado un empleado único es el “IDdelEmpleado”.

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.AreEqual(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

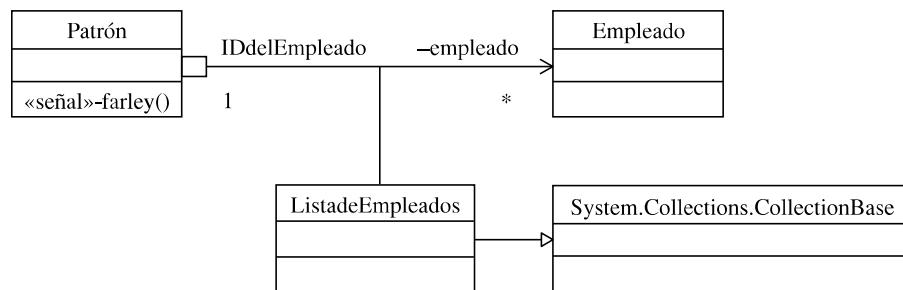


Figura 6-16 En el diagrama revisado se usa una clase asociación para introducir la generalización que muestra que “ListadeEmpleados” hereda de “System.Collections.CollectionBase”.

“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 *fuente* y *objetivo*. 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 *fuente* y *objetivo*, 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 metaclass (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 subclases 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