UML

[redacted]

CONTROL #2
(CAPS. 5 A 8)
STEVENS

Capítulo 5

Fundamentos de los modelos de clases

Este capítulo presenta los diagramas de clases de UML, que se utilizan para documentar la estructura estática del sistema; esto es, qué clases hay y cómo están relacionadas, pero no cómo interactúan para alcanzar comportamientos particulares. Un diagrama de clases puede también mostrar otros aspectos de la estructura estática, tales como paquetes, que se tratarán en los Capítulos 6 y 14.

En UML, una clase aparece en una diagrama de clases como un rectángulo con su nombre. La Figura 5.1 es un icono de clase para la clase `Libro`.

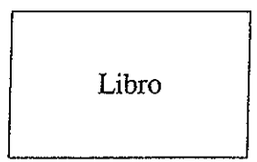


Figura 5.1 Un modelo de clases muy sencillo.

Más tarde, en este capítulo se verá cómo representar más información sobre los datos y el comportamiento encapsulado por una clase, pero por ahora nos centraremos en la identificación de clases y de las asociaciones entre ellas.

5.1 Identificar objetos y clases

La construcción de un modelo de clases incluye la identificación de las clases que deberían existir en nuestro sistema: esta es una parte fundamental del trabajo de diseñar un sistema orientado a objetos. Antes de tratar cómo identificar objetos y clases, discutamos los criterios para tener éxito.

5.1.1 ¿Qué hace que un modelo de clases sea bueno?

En última instancia, hay dos objetivos que se pretenden alcanzar:

- Construir, lo más rápido y barato posible, un sistema que satisfaga nuestros requisitos actuales.
- Construir un sistema que sea fácil de mantener y adaptar a futuros requisitos.

Estos objetivos muchas veces se encuentran enfrentados; una razón del éxito de las técnicas orientadas a objetos, y especialmente de las técnicas de diseño basadas en componentes, es que permiten dar un paso hacia su reconciliación.

Para cumplir el primer objetivo,

> Los objetos de las clases elegidas tienen que ser capaces de proporcionar cada parte del comportamiento que requiere el sistema.

Ya se ha visto, en el Capítulo 1, que para cumplir el segundo objetivo habría que construir un sistema compuesto por módulos encapsulados, con débil acoplamiento y fuerte cohesión. Además:

> Un buen modelo de clases está formado (dentro de lo posible) por clases que representan clases permanentes de los objetos del dominio, que no dependen de la particular funcionalidad hoy requerida.

Por ejemplo, cualquier sistema de biblioteca contendrá libros, por lo que es razonable tener una clase `Libro`. (Destacar que los nombres son importantes: no sería tan claro llamar a la clase `ArtefactodePapelLiterario` o `L`).

Pregunta de Discusión 30

¿Por qué hay un conflicto entre los dos objetivos anteriores? ¿Qué consideraciones podrían determinar que una organización considere una más significativa que la otra?

5.1.2 Cómo construir un buen modelo de clases

Destacar primero que puede utilizar cualquier técnica que le guste para obtener sus clases: cualquier cosa, incluyendo la inspiración divina, está bien, si nos dirige a un modelo de clases bueno por los criterios que se han dado. A la inversa, si se produce un modelo de clases malo (uno que no cumple los criterios) ¡a nadie le importará qué método maravilloso haya utilizado para obtenerlo!

En la práctica, es probable que lo haga correctamente la primera vez. La colección de clases en su modelo de diseño es una de las cosas que probablemente cambiará a lo largo y dentro de las iteraciones de desarrollo. Normalmente, identificará las clases más importantes de los

objetos de *dominio* —es decir, aquellas q[redacted] obvia al problema en vez de aquellas que se introducen para resolverlo— primero y más fácilmente; las otras clases, que se corresponden con menor claridad con los objetos del dominio, son más difíciles de identificar con seguridad.

¿Qué dirige?

Los expertos en OO tienden a dividirse en dos grupos, aquellos partidarios del *diseño dirigido a los datos* y aquellos que defienden el *diseño dirigido a la responsabilidad*. Tal y como se ha visto en los Capítulos 2 y 3, las clases tienen tanto datos como responsabilidades. Una caricatura del diseño dirigido a los datos (DDD) es que supone la identificación de todos los datos en el sistema y luego la división en clases, antes de considerar las responsabilidades de las clases; la técnica de identificación de nombres, que se utiliza en el Capítulo 3 y que se considerará aquí más adelante, es una parte fundamental en el DDD. Una caricatura del diseño dirigido a la responsabilidad (DDR) es que supone la identificación de todas las responsabilidades en el sistema y su división en clases, antes de considerar los datos de las clases.

Por supuesto, ambas caricaturas describen aproximaciones extremas que no funcionarían: nadie propone seriamente nada tan extremo. Se sabe que la distinción entre DDR y DDD no es útil en el contexto de este libro. Es mucho más fácil utilizar una aproximación mezclada en un proyecto que describirlo en un libro: por claridad, se presentarán las técnicas de forma separada, pero la naturaleza de un proyecto orientado a objetos con éxito es el que utilice varias técnicas, a menudo de manera simultánea. La identificación de nombres, que se describe a continuación, es una técnica asociada con DDD; las tarjetas CRC, que se tratarán al final del capítulo, es más una técnica DDR.

Una técnica: identificación de nombres

En el Capítulo 3 se vio un ejemplo de cómo identificar objetos y clases. Se procede en dos etapas:

1. Identifica las clases candidatas seleccionando todos los nombres y locuciones nominales de la especificación de requisitos del sistema. (Considérelos en forma singular, y no incluya frases que contengan "o" como candidatas).
2. Descarta las candidatas que son inapropiadas por cualquier razón, renombrando las clases restantes, si fuera necesario.

Las razones (algunas de las cuales se vieron en el Capítulo 3) por las que se podría decidir que una clase candidata es inapropiada incluye que es[1]:

- **Redundante**, donde a la misma clase se le ha dado más de un nombre. Es, sin embargo, importante recordar que los objetos parecidos no tienen que ser completamente iguales: una de las cosas que hay que decidir es si las clases son lo suficientemente diferentes para considerarlas clases distintas. Por ejemplo, se incluyen aquí pares como "préstamo" y "préstamo a corto plazo": son diferentes, pero probablemente sólo en los valores de los atributos. Elija un nombre para la clase que abarque todas las descripciones que quiera que incluya.

[1] Esta lista está inspirada, pero no es idéntica, a una descrita en Rumbaugh *et al.* en [41], y adaptada en el curso de la Open University M868 [47].

- **Impreciso,** donde no se puede indicar de forma no ambigua lo que significa un nombre. Obviamente, hay que eliminar la ambigüedad antes de poder decir que se trata de una clase.
- **Un evento u operación,** donde el nombre hace referencia a algo que se hace para, por o en el sistema. A veces, tales cosas *están* bien modeladas en una clase, pero no es lo normal. Retomando la discusión del Capítulo 2, pregúntese si la instancia del evento u operación tiene estado, comportamiento e identidad. Si no, descártelo.
- **Meta-lenguaje,** donde el nombre forma parte de la manera en que se definen las cosas. Se utilizan los nombres *requisitos* y *sistema*, por ejemplo, como parte del lenguaje de modelado, en vez de representar objetos en el dominio del problema.
- **Fuera del alcance del sistema,** donde el nombre es relevante para describir cómo funciona el sistema pero que no hace referencia a algo interno al sistema, por ejemplo, *biblioteca* en el ejemplo del Capítulo 3. Los nombres de los actores muchas veces se descartan por esta norma, cuando no es necesario modelarlos en el sistema. Se podría utilizar también esta norma para justificar el descarte de *semana* en el Capítulo 3, aunque este es un ejemplo de dónde es mucho más obvio decir que algo no es una clase que decir por qué.
- **Un atributo,** donde está claro que un nombre hace referencia a algo sencillo sin un comportamiento interesante, que es un atributo de otra clase. ("Nombre" de socio de biblioteca podría ser un ejemplo).

Pregunta de Discusión 31

En general, si se duda sobre si mantener una clase, ¿cree que es mejor mantenerla (posiblemente eliminándola más tarde) o eliminarla (posiblemente volviendo a incluirla)?

Esta es una pregunta en la que no se ponen de acuerdo las personas con mayor experiencia: probablemente hay un elemento de psicología individual. Algunas personas mantienen dos listas, una con los candidatos más firmes y otra con los más dudosos, y esta es una técnica útil para evitar perder información mientras se está distinguiendo todavía las cosas de las que se está seguro, de las cosas que tienen que ser fijados todavía. Por ejemplo, en el ejemplo de la biblioteca del Capítulo 3, puede que no se esté esté seguro de descartar préstamo o norma, por lo que podrían estar en la lista de "posibles". (Se discutirán las clases asociación, de las que resultará ser un ejemplo el préstamo, en el Capítulo 6. Norma no es probable que sea una clase en este sistema particular, pero puede ser útil codificar las tan nombradas *normas de negocio* como clases, especialmente en los casos en los que las normas son complejas y propensas a cambiar). Una vez que se empieza a identificar asociaciones entre clases, la pregunta de qué clases pertenecen a un modelo de clases a nivel conceptual se responde con mucha rapidez.

Esta técnica tan sencilla es una manera extraordinariamente útil de empezar. Esto es todo lo que es: por ejemplo, la lista de razones para descartar clases candidatas no es exhaustiva, y puede haber más de una razón para descartar la misma candidata. Cuando se tiene más experiencia, probablemente, se utiliza de cabeza como comprobación para encontrar cualquier abstracción del dominio que se puede haber olvidado. Probablemente, se identificarán clases y asociaciones en paralelo, aunque, por claridad, aquí se presentan como procesos separados. También es útil empezar a utilizar tarjetas CRC en esta etapa: se tratarán las tarjetas CRC al final de este capítulo.

Pregunta de Discusión 32

¿Es razonable la lista de razones para el descarte? ¿Puede pensar algún caso donde pueda ser muy selectiva o muy permisiva? ¿Quiere modificar la lista?

5.1.3 ¿Qué son las clases?

Una clase describe un conjunto de objetos con un rol o roles equivalentes en un sistema.

Los objetos y su división en clases se derivan, normalmente, de una de las siguientes fuentes (originalmente definidos por Shlaer y Mellor [42] y, más tarde, parafraseado por Booch [7] y adaptado aquí más ampliamente).

- **Cosas tangibles** o "del mundo real": libro, copia, curso.
- **Roles:** socio biblioteca, estudiante, director de estudios.
- **Eventos:** llegada, salida, petición.
- **Interacciones:** encuentro, intersección.

Estas categorías se solapan, y las dos primeras son fuentes de objetos y de clases mucho más comunes que las dos últimas. Por el contrario, si se han identificado objetos que entran dentro de las dos primeras categorías, las otras dos pueden ayudar a encontrar y nombrar asociaciones entre ellos.

5.1.4 Objetos del mundo real frente a su representación en el sistema

Es importante recordar que los objetos son realmente cosas dentro de un programa de ordenador —que cuando se habla sobre "libros" y "copias", por ejemplo, realmente nos referimos a la representación de estas cosas dentro de nuestro sistema—. Las consecuencias de esto son que hay que tener cuidado:

- de no almacenar información que sea definitivamente irrelevante para nuestro sistema.
- de no perder la visión del hecho de que ¡los objetos *son* el sistema!

El último punto es particularmente interesante. Un error clásico en la gente que todavía no está empapada de OO es inventarse una clase, a menudo llamada [Cualquier cosa]Sistema, que implementa todo el comportamiento interesante del sistema. Pero en OO todo el negocio es el sistema —¡este es el tema! Es fácil dejarse llevar hacia un diseño *monolítico* donde hay un único objeto que conoce y hace todo. Esto está mal porque tales diseños son muy difíciles de mantener: tienden a tener presunciones sobre cómo será utilizado el sistema. (Hay una manera muy diferente de tener una clase que encapsula el sistema: muchas veces es útil, y en algunos lenguajes obligatorio, tener una clase principal, que se instancia una sola vez en cada instancia de ejecución del sistema, y que proporciona el punto de entrada al programa. Al empezar el programa automáticamente crea este objeto principal, que, por turnos, crea el resto de objetos en el sistema. El objeto principal, sin embargo, no tiene ningún comportamiento complejo por sí

mismo. Todo lo más, podría enviar los mensajes que le llegan de fuera al objeto apropiado. Esto está relacionado con el *patrón Fachada (Façade)*, que consideraremos en el Capítulo 18).

En el Capítulo 7 se volverá a la pregunta sobre cómo los participantes de fuera de nuestro sistema informático (conocidos en UML como *actores*) se representan en nuestro diseño. Esto (se volverá a ello más tarde) es una parte fundamental de lo que se ha tratado aquí hasta el momento. Ya se vio en el Capítulo 3.

P: Revise la descripción de los requisitos para el sistema de la biblioteca en el Capítulo 3. Aparte de las clases identificadas para la primera iteración, ¿qué clases debería haber en el sistema final?

5.2 Asociaciones

Al igual que las clases se corresponden con nombres, las asociaciones se corresponden con verbos. Expresan las relaciones entre clases. En el ejemplo del Capítulo 3, se vieron asociaciones tales como `es una copia de` y `toma prestado/devuelve`.

Hay instancias de asociaciones, al igual que hay instancias de clases. (Las instancias de las clases se llaman objetos; las instancias de las asociaciones se llaman *enlaces*, en UML, aunque este término actualmente se utiliza muy poco). Una instancia de una asociación relaciona un par[2] de objetos.

Se pueden ver las asociaciones conceptualmente o desde el punto de vista de la implementación. Conceptualmente, se almacena una asociación si hay una asociación en el mundo real descrita mediante una sentencia corta como "un socio de la biblioteca toma prestado un libro" y la sentencia parece relevante para el sistema en cuestión.

La clase A y la clase B están asociadas si:

- un objeto de la clase A envía un mensaje a un objeto de la clase B.
- un objeto de la clase A crea un objeto de la clase B.
- un objeto de la clase A tiene un atributo cuyos valores son objetos de la clase B o colecciones de objetos de la clase B.
- un objeto de la clase A recibe un mensaje con un objeto de la clase B pasado como argumento.

En resumen, si algún objeto de la clase A tiene que saber de algún objeto de la clase B. Cada enlace, es decir, cada instancia de la asociación, relaciona un objeto de la clase A y un objeto de la clase B. Por ejemplo, la asociación llamada `toma prestado/devuelve` entre `SocioBiblioteca` y `Copia` podría tener los siguientes enlaces:

- Jo Bloggs `toma prestado/devuelve` copia 17 de *El Principio de Dilbert.*
- Marcus Smith `toma prestado/devuelve` copia 1 de *El Principio de Dilbert.*
- Jo Bloggs `toma prestado/devuelve` copia 4 de *Reutilización de Software.*

[2] En este libro se tratan sólo las asociaciones binarias, aunque en realidad UML tiene asociaciones de otras aridades.

Pregunta de Discusión 33

Como alternativa de llamar a esta asociación `toma prestado/devuelve`, (los autores) se podría haber decidido tener dos asociaciones separadas, una llamada `toma prestado` y la otra llamada `devuelve`. Está claro que, si en vez de considerar quién toma prestado y quién devuelve, se hubiese considerado quién es el autor de una copia y quién es el dueño, se habría decidido tener dos asociaciones separadas. ¿Qué es diferente en las dos situaciones? ¿Está de acuerdo con nuestra decisión?

Pregunta de Discusión 34

Piense cómo se solapan los casos de asociaciones listados arriba, y considere si alguno de ellos debería eliminarse. Es discutible, por ejemplo, que si un objeto de la clase A recibe un mensaje con un objeto de la clase B como argumento, pero no va a enviarle más tarde a ese objeto ningún mensaje ni lo va a almacenar en un atributo, entonces esto no debería considerarse asociación. En realidad esta situación muchas veces es, aunque no siempre, un mal diseño. Construya algunos ejemplos y considere si piensa que son sensatos. ¿Tiene alguna opinión sobre si se debe considerar como asociación?

El secreto de un buen diseño orientado a objetos está en terminar con un modelo de clases que no distorsione la realidad conceptual del dominio —de forma que alguien que comprenda el dominio no se lleve sorpresas desagradables— pero que también permita una implementación coherente de la funcionalidad requerida. Cuando se desarrolla el modelo de clases inicial, antes de identificar los mensajes que pasan entre los objetos, necesariamente uno se centra en el aspecto conceptual del modelo. Más tarde, cuando se utilizan los modelos de interacción para comprobar el modelo de clases, estaremos más implicados en ver si el modelo permite una implementación coherente de la funcionalidad requerida. Sin embargo, el proceso no es que primero se desarrolle el modelo conceptual, y entonces se olviden las relaciones conceptuales para desarrollar la implementación del modelo de clases. A lo largo del desarrollo, se tiene como objetivo construir un modelo que sea bueno tanto en el aspecto conceptual como en el de implementación. Prosperar en esto es una parte muy importante de lo que nos lleva a un sistema de fácil mantenimiento, debido a que un modelo así tiende a ser relativamente fácil de comprender, y por lo tanto relativamente fácil de modificar de manera sensata.

En la Figura 5.2 se puede ver cómo UML representa una asociación general entre dos clases dibujando una línea entre sus iconos. Esto, normalmente, se representa de varias maneras. Debería, al menos, tener una etiqueta con un nombre, por legibilidad. Se puede incluir una flecha en la etiqueta para indicar en qué sentido se aplica. En el Capítulo 6 se discutirá la utilización de las flechas en la línea de asociación para denotar la *navegabilidad:* ¿es el libro quien sabe de la

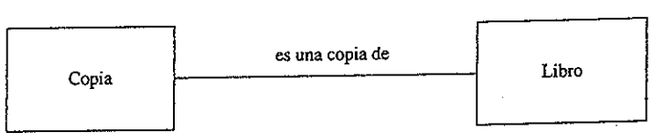


Figura 5.2 Asociación sencilla entre clases.

copia, o viceversa, o ambos? Puede que se quiera pensar sobre estas preguntas ahora: algunos expertos están convencidos de que es importante responder estas preguntas pronto, otros no están de acuerdo.

Pregunta de Discusión 35

¿Cuáles son las ventajas y las desventajas de decidir la navegabilidad en esta etapa?

En las primeras etapas del desarrollo de un modelo, muchas veces es suficiente dibujar una simple línea, indicando, pero sin fijar, la existencia de acoplamiento. Según se va madurando el diseño, la línea puede sustituirse por varias, indicando diferentes tipos de asociaciones. Algunos tipos de asociaciones son tan comunes que UML tiene una manera predefinida de mostrarlos; otros pueden ser definidos por el diseñador cuando lo necesite. Se verán diferentes formas en el Capítulo 6.

ANOTACIÓN TÉCNICA DE UML

La definición de asociación que se utiliza en este libro es una definición *dinámica*: si en tiempo de ejecución los objetos pueden intercambiar un mensaje, tiene que haber una asociación navegable entre sus clases. A veces es conveniente seguir una visión *estática* más restrictiva, donde la clase A sólo tiene una asociación navegable a la clase B si un atributo de A contiene un objeto (o colección de objetos) de la clase B. UML no especifica qué definición utilizar: en la práctica, es tarea del modelador decidir exactamente qué significa la existencia de una asociación. (Algunas veces, la elección podría estar determinada por la elección de la herramienta, especialmente si se usan características para la generación de código. UML2 también incluye la notación de un "Conector", siendo un camino más general para la conexión de clases (y otros clasificadores). Si se toma una visión restrictiva de las asociaciones, también se podría querer utilizar conectores en otras circunstancias. Miraremos, de forma concreta, una clase útil de conector, un "conector de ensamblado", en el próximo capítulo.

Una anotación que se utiliza con frecuencia es la *multiplicidad* de una asociación. Aunque no siempre puede quedar claro al principio y puede cambiar con un refinamiento del diseño, esto es tan fundamental que llevará algún tiempo pensar sobre ello.

5.2.1 Multiplicidades

En el ejemplo del Capítulo 3, se ponía un 1 en la parte de la asociación `es una copia de`, correspondiente a `Libro` porque toda copia (es decir, cada objeto de la clase `Copia`) está asociada mediante `es una copia de` con un solo libro (objeto de la clase `Libro`). Por el contrario, en nuestro sistema puede haber cualquier número de copias de un libro determinado. Por lo que la multiplicidad en la parte de `Copia` es `1..*`.

Como se puede ver, es posible especificar:

- **Un número exacto,** simplemente escribiéndolo.
- **Un rango de números,** utilizando dos puntos entre un par de números.
- **Un número arbitrario, no especificado,** utilizando * (asterisco).

Libremente, se puede pensar que * en UML es como un símbolo de infinito, por lo que la multiplicidad 1..* expresa que el número de copias puede ser cualquier cosa entre 1 e infinito. Por supuesto, cada vez habrá, en la realidad, un número finito de objetos en nuestro sistema completo, por lo que esto realmente indica que puede haber cualquier número de copias de un libro, a condición de que haya, al menos, uno.

Atributos, que se discutirán en la siguiente sección, también pueden tener multiplicidades.

P: Exprese en UML que un `Estudiante` empieza seis `Módulos`, donde como máximo pueden matricularse 25 `Estudiantes` en cada `Módulo`.

P: Considere las diferentes maneras en las que se puede implementar una asociación en su lenguaje de programación.

Pregunta de Discusión 36

Exprese en UML la relación entre una persona y sus camisetas. ¿Qué pasaría con los zapatos de la persona? ¿Cree que ha introducido alguna debilidad en UML? ¿Por qué, o por qué no?

Pregunta de Discusión 37

El número cero nunca puede ser una multiplicidad significativa, ¿o sí?

Pregunta de Discusión 38

La existencia de una multiplicidad mayor que uno a veces se supone que significa que los objetos de esa clase tienen que existir como una colección de cualquier tipo. ¿Es una suposición segura?

5.3 Atributos y operaciones

El sistema que se construye consistirá en una colección de objetos, que interactúan para completar los requisitos del sistema. Se ha empezado a identificar las clases y sus relaciones, pero esto no puede ir más lejos sin considerar el estado y el comportamiento de los objetos de estas clases. Es necesario identificar las operaciones y los atributos que cada clase debería tener. Algunos serán obvios; otros aparecerán cuando se consideren las responsabilidades de los objetos y las interacciones entre ellos.

5.3.1 Operaciones

Lo más importante son las operaciones de una clase, que definen las maneras en que los objetos pueden interactuar. Tal y como se dijo en el Capítulo 2, cuando un objeto envía un mensaje a otro, le está solicitando al receptor que ejecute una operación. El receptor invocará a un método

para ejecutar la operación; el emisor no sabe qué método será invocado, ya que puede haber muchos métodos implementando la misma operación a diferentes niveles de la jerarquía de herencia. La *signatura* de una operación da el selector, los nombres y tipos de cualquier parámetro formal (argumentos) de la operación, y el tipo del valor de retorno. Aquí, como siempre, un tipo puede ser tanto un tipo básico como una clase. Las operaciones se listan en el compartimento de abajo, el tercero, del icono de clase.

P: Revise el ejemplo del Capítulo 3 y obtenga las operaciones para todas las clases.

5.3.2 Atributos

Los atributos de una clase —los cuales, tal y como se trató en el Capítulo 2, describen los datos contenidos en un objeto de la clase— se listan en el segundo compartimento, el de enmedio, de un icono de clase. La Figura 5.3 es una versión del icono de clase `Libro` que muestra que cada objeto de la clase `Libro` tiene un `título`, que es una cadena, y entiende un mensaje con el selector `copiasEnEstantería`, que no tiene argumentos y devuelve un entero, al igual que `tomarPrestado`, que tiene como argumento un objeto de la clase `Copia` y no devuelve ningún resultado.

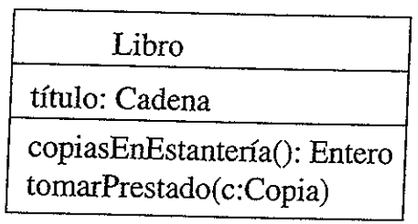


Figura 5.3 Un modelo de clases sencillo, con atributo y operación.

Destacar que no se incluyen los atributos que sólo implementan las asociaciones mostradas en el diagrama de clases. Por ejemplo, *no* se incluye que `Libro` tiene un atributo `copias` en el que almacenar la colección de referencias a los objetos `Copia` asociadas al `Libro`. La implementación final probablemente tendrá dicho atributo, pero incluirlo añadiría notación extra al diagrama sin añadir información útil. (Esto implicaría incluso una decisión sobre la navegabilidad de la asociación, cuya discusión sería prematura. La navegabilidad se tratará más ampliamente en el Capítulo 6).

Una norma es que los tipos de los atributos deberían ser cualquiera de los tipos primitivos (entero, cadena, etc.) o clases que no aparezcan en el diagrama de clases, como librerías de clases. Si el tipo de un atributo en una clase no aparece en el diagrama de clases, normalmente es mejor registrar una asociación entre las dos clases.

P: Busque cualquier atributo obvio que podría existir para el resto de clases del ejemplo del Capítulo 3.

Vistas de operaciones y atributos

Sólo con asociaciones se tiene una aproximación conceptual y pragmática que se intenta hacer consistente.

La aproximación conceptual incluye identificar qué datos están asociados conceptualmente con un objeto de esta clase, y qué mensajes parece razonable esperar que entienda un objeto. La última vista puede llevarle a una visión antropomórfica de los objetos como si tuvieran inteligencia por sí mismos —"Si un libro pudiese hablar, ¿qué preguntas cree que sería capaz de responder?"— cosa que algunas personas ven desconcertante, ¡pero que pueden, sin embargo, merecer la pena!

De forma pragmática, hay que comprobar que se han incluido los datos y el comportamiento suficientes para los requisitos en cuestión. Para hacer esto, hay que empezar a considerar cómo trabajan juntos los objetos de nuestras clases para satisfacer los requisitos. Una técnica muy útil para hacer esto es la utilización de las tarjetas CRC, que se describen más tarde en este capítulo.

5.4 Generalización

Otra relación importante que puede existir entre las clases es la *generalización*. Por ejemplo, `SocioBiblioteca` es una generalización de `SocioPlantilla` porque, conceptualmente, todo `SocioPlantilla` es `SocioBiblioteca`. Todo lo que pueden hacer todos los `SocioBiblioteca`, naturalmente lo puede hacer un `SocioPlantilla`. Por lo que si alguna parte de nuestro sistema (por ejemplo, la facilidad de reservar un libro) funciona para un `SocioBiblioteca` arbitrario, debería funcionar también para un `SocioPlantilla` arbitrario, ya que todo `SocioPlantilla` es un `SocioBiblioteca`. En el lado contrario, puede haber cosas que no tengan sentido para todos los `SocioBiblioteca`, pero sí para `SocioPlantilla` (por ejemplo, tomar prestada una revista). `SocioPlantilla` está más especializado que `SocioBiblioteca`; o `SocioBiblioteca` es una generalización de `SocioPlantilla`.

En otras palabras, un objeto de la clase `SocioPlantilla` debería *ajustarse* a la interfaz dada por `SocioBiblioteca`. Esto es, si algún mensaje lo acepta cualquier `SocioBiblioteca`, también tiene que ser aceptado por cualquier `SocioPlantilla`. `SocioPlantilla`, por el otro lado, puede entender otros mensajes especializados que un `SocioBiblioteca` podría no ser capaz de aceptar —esto es, la interfaz de un `SocioPlantilla` puede ser estrictamente más amplio que la de un `SocioBiblioteca`—.

De esto se puede ver que decir que existe una relación de generalización entre clases es realizar una declaración importante, aunque informal, sobre la manera en que se comportan los objetos de ambas clases.

> Un objeto de una clase especializada puede sustituirse por un objeto de una clase más general en cualquier contexto que espere un miembro de la clase más general, pero no al revés.

Esto nos lleva a una regla sobre el diseño de clases donde una es una especialización de la otra.

> No tiene que haber un abismo conceptual entre lo que hacen los objetos de las dos clases al recibir el mismo mensaje.

Establecer esto de una manera mucho más precisa es sorprendentemente difícil: *véase* el Panel 5.1.

PANEL 5.1 Diseño por contrato 2: posibilidad de sustitución

En este panel se continúa la discusión del contrato que tiene que cumplir un objeto, considerando cómo se relaciona este concepto con la herencia. Se supone que un objeto de una subclase es reutilizable en cualquier parte en la que es utilizado un objeto de la superclase. En otras palabras, se supone que la subclase cumple el contrato firmado por la superclase. Examinemos qué significa esto en la práctica.

El aspecto más sencillo es lo que significa para los atributos de la clase y el invariante de clase. (Recuerde del Capítulo 3 que un invariante de clase es una declaración de los valores de los atributos que tiene que tener para todos los objetos de la clase).

P: En el Capítulo 2 se dijo que una subclase podía tener atributos y operaciones extra además de los de su superclase, pero no podía eliminar ningún atributo u operación de la superclase. ¿Por qué? ¿Qué pasa con las operaciones que no están en la interfaz pública?

P: Si una superclase tiene un invariante de clase, ¿necesita la subclase un invariante de clase? ¿Cómo debería ser la relación entre los dos invariantes? De ejemplos de herencia correcta (sustitutiva) e incorrecta en este caso.

La posibilidad de sustitución necesita comprobar si la subclase anula cualquiera de los métodos de la superclase: esto es, define sus propios métodos para implementar las operaciones. Se dijo que los métodos nuevos tienen que hacer, conceptualmente, lo mismo; ahora seremos más precisos. El eslogan general para la subclase, visto como un subcontratista, es:

> No exija más: no prometa menos

"No exija más" describe las circunstancias bajo las que la subclase debe aceptar el mensaje (sin quejarse). Tiene que estar preparado para aceptar cualquier argumento que la superclase hubiese aceptado. Por lo tanto, su precondición *no* debe ser *más fuerte* que la precondición de la implementación de la superclase. Puede ser que la implementación de la subclase sea una mejora de la implementación de la superclase en el sentido de que funciona en más situaciones: esto está bien.

"No prometa menos" describe lo que el cliente puede asumir sobre el estado del mundo después de que se haya ejecutado la operación. Cualquier suposición que fuese válida cuando se estaba utilizando la implementación de la superclase debería seguir siendo válida cuando se utilice la implementación de la subclase. Por lo tanto, la post-condición de la subclase tiene que ser *al menos tan fuerte como* la versión de la superclase.

La subdivisión por tipos de comportamiento, también conocida como posibilidad de sustitución de Liskov, después de que Barbara Liskov lo popularizara, es una versión fuerte y precisa de este slogan. Suponga que algunos programas esperan interactuar con un objeto de la clase `C`, y que en vez de él, recibe un objeto `s` de la clase `S`, subclase de `C`. Si la posibilidad de sustitución de Liskov es válida, hay algún objeto de la clase `C` que podría utilizarse en vez de `s` sin alterar nada en el comportamiento del programa.

Pregunta de Discusión 39

Construya unos cuantos ejemplos para ver lo que esto significa en la práctica. ¿Hay circunstancias bajo las cuales este requisito podría ser demasiado fuerte? ¿Demasiado débil?

P: En su lenguaje de programación, si una subclase anula un atributo u operación de la superclase, ¿puede cambiar sus tipos? ¿De qué manera?

Considere la lista de casos en que las dos clases estaban asociadas (Sección 5.2). Quizá se debería añadir la palabra "intencionadamente" a algunos de los casos: la clase A y la clase B están asociadas si un objeto de la clase A "intencionadamente" le envía un mensaje a un objeto de la clase B, y así sucesivamente. Suponga que el código de la clase A hace mención a un objeto que el escritor de la clase A espera que pertenezca a la clase B. El tema es que el objeto podría realmente pertenecer a cualquier subclase de la clase B, y —si la posibilidad de sustitución es válida— la clase A funcionará como se esperaba. No es útil añadir una asociación entre la clase A y cada subclase de la clase B: esto confundiría el diagrama sin aportar ninguna información nueva.

En otras palabras, si hay una operación como `tomarPrestado(c:Copia)` definida por cualquier objeto de la clase `SocioBiblioteca`, la manera en que se ejecuta la operación debería ser claramente comparable a la manera en que se ejecuta para `SocioPlantilla`. Su comportamiento no tiene que ser idéntico, pero tienen que ser suficientemente parecidos para que otras operaciones que confían en lo que hace la operación `tomarPrestado(c:Copia)` cuando la solicita un objeto de una clase más general (SocioBiblioteca) funcionen también con un objeto de una clase especializada (SocioPlantilla).

La Figura 5.4 muestra cómo UML representa la generalización.

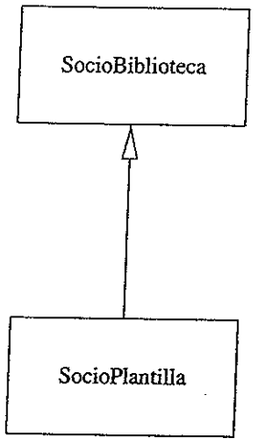


Figura 5.4 Una generalización sencilla.

5.4.1 Utilización del lenguaje para comprobar si existe una generalización

La clase `Bar` es probablemente una generalización de la clase `Foo` si es verdad que *cada* `Foo` *es un* `Bar`. A menudo hablamos de relaciones "es un" —*un* `Foo` *es un* `Bar`.

Sin embargo, si se confía con demasiada ingenuidad en esta comprobación, es muy posible que se fracase, debido a la ambigüedad del lenguaje. Por ejemplo (del libro de Fowler y Scott, *UML Distilled*, página 75 [19]), se podría decir, por casualidad, que "un Border Collie es una Raza" y determinar que Raza es una generalización de Border Collie, cosa que es mentira. "todo Border Collie es una Raza" prueba que es más obvia su falsedad.

El problema está en que realmente está mal empezar con la declaración en español de "un Border Collie es una Raza", ya que relaciona una única instancia de Border Collie con un nombre colectivo, Raza. Debería haber sido "Border Collie es una Raza". Si se pone "todos", en el lugar de "un", ayuda a ver que la primera afirmación es absurda.

5.4.2 Implementación de la generalización: herencia

Una manera (pero no la única) de implementar una generalización es mediante la *herencia*. Mientras la generalización es una relación conceptual entre clases, la herencia es una relación de implementación[3].

En vez de duplicar la definición de `SocioBiblioteca`, y añadirla para definir `SocioPlantilla` —cosa que sería posible, y que todavía sería una generalización entre `SocioBiblioteca` y `SocioPlantilla`— a menudo se decide utilizar herencia, proporcionada por los lenguajes de programación orientados a objetos para definir `SocioPlantilla` en función de `SocioBiblioteca`. Se trata la herencia como concepto técnico en el Capítulo 2; aquí hay que tratar los temas pragmáticos de cómo se utiliza la herencia.

La herencia no es la gran solución que puede parecer. El principal problema es que una subclase es dependiente de sus superclases, por lo que el uso de la herencia tiende a aumentar el acoplamiento en un sistema. Si una superclase cambia más tarde, incluso de una manera que no afecte a su comportamiento, esto puede forzar la recompilación de las subclases, o incluso puede obligar a realizar cambios en su código. Esto se suele llamar *problema de clase base frágil*. Otro efecto del fuerte acoplamiento entre la subclase y su superclase es que, incluso aunque se esté seguro de que una clase es correcta, no siempre es posible utilizar esta confianza para reducir la cantidad de comprobaciones que hay que hacer en la subclase. Se volverá a esto cuando se traten las pruebas en el Capítulo 19.

A causa de estos problemas se recomienda utilizar la herencia entre clases, sólo cuando se modela una relación de generalización conceptual. Se tiende a utilizarla, por conveniencia, en otros casos, pero no siempre merece la pena; una implementación que utiliza la composición a menudo es mucho más robusta. Supongamos, por ejemplo, que se tiene una clase `Lista` disponible y que se quiere implementar una clase `Agenda` donde las direcciones se almacenan en una `Lista`. Un clásico error es hacer que `Agenda` herede de `Lista`. Una solución mejor es hacer que `Agenda` tenga una `Lista`, por ejemplo, por medio de un atributo `direcciones:Lista`.

[3] Cuando las dos nociones se distinguen completamente —tal y como se indica en el Capítulo 2—, muchas veces se utilizan como sinónimos.

Pregunta de Discusión 40

¿Por qué es mejor la solución de la composición que la solución de la herencia? Compare el esfuerzo implicado en el desarrollo inicial de `Agenda`. Compare entonces los cambios requeridos si la implementación de la `Lista` o la interfaz cambian, y aquellos requeridos si se decide utilizar una clase diferente en vez de `Lista` (digamos, `Diccionario`).

P: ¿Libro y Revista se relacionan por generalización?

5.5 El modelo de clases durante el desarrollo

Tal y como se ha mencionado, el modelo de clases se desarrollará gradualmente a lo largo de varias iteraciones del diseño del sistema. Se empieza almacenando las relaciones conceptuales y las funciones, independientemente de cualquier implementación lo más lejana posible.

Más tarde se añaden más detalles, introduciendo nuevas operaciones y asociaciones más específicas. Los atributos son añadidos en estos procesos de refinamiento, que son repetidos hasta que se esté satisfecho de que nuestro sistema es completo y consistente.

Modelos frente a diagramas

Recuerde que en el Capítulo 4 se explicó la diferencia en UML entre un *modelo* de un sistema, que es una colección de elementos del modelo que representan una vista del diseño en un determinado nivel de abstracción, y un *diagrama* de ese modelo, que es una manera de representar el modelo gráficamente. Se avisó también de que, aunque la distinción a veces es importante, en la práctica los términos modelo y diagrama muchas veces se utilizan indistintamente. Aquí hay un caso en el que es importante distinguirlos.

Hay un único modelo de clases —la guía de la notación de UML lo llama, de forma más exacta, el modelo estructural estático, ya que, como se verá, puede representar más que clases— para un determinado sistema, y cada clase aparece en él sólo una vez. El modelo de clases describe la estructura estática general del sistema. Sin embargo, puede describirse con varios *diagramas* de clases diferentes por legibilidad. De forma similar, aunque cada clase aparece una sola vez en el modelo de clases, es posible representar una clase más de una vez en un diagrama de clases. Se podría querer hacer esto si la distribución del diagrama es más legible de esa manera. Sin embargo, ya que en el modelo subyacente hay sólo un elemento de modelo que representa la clase, es vital que los dos iconos de la clase en un diagrama no proporcionen información contradictoria sobre la clase. Una herramienta CASE puede ayudar a asegurar esta consistencia. Sin ella, se recomienda que no representar la misma clase más de una vez en un diagrama de clase.

5.6 Tarjetas CRC

Una forma común de comprobar el refinamiento de un buen diseño y una buena guía es utilizar las tarjetas CRC. CRC quiere decir Clase, Responsabilidades, Colaboraciones. Las tarjetas

CRC fueron introducidas (y descritas en un escrito [2] en la mayor conferencia de OO OOPSLA'89) Kent Beck y Ward Cunningham, como técnica para ayudar a los programadores con experiencia en los lenguajes no OO a "pensar en objetos".

Aunque las tarjetas CRC no son parte de UML, añaden algunas ideas útiles en todo el desarrollo, incluyendo las etapas más tempranas en las que se está identificando las clases y sus asociaciones. Por lo tanto, se va a dar una vuelta breve para ver cómo crearlas y utilizarlas.

5.6.1 Creación de tarjetas CRC

En una pequeña tarjeta, se almacenan:

- **el nombre** de la clase, en la parte de arriba.
- **las responsabilidades** de la clase, en la parte izquierda.
- **los colaboradores** de la clase, que ayudan a ejecutar cada responsabilidad, en la parte derecha de la tarjeta.

Las responsabilidades de la clase describen, a alto nivel, el propósito de la existencia de la clase: están relacionadas con las operaciones que proporciona la clase, pero son más generales de lo que se podría suponer. Por ejemplo, "mantener los datos del libro" se acepta como descripción de una responsabilidad, aunque podría haber muchas operaciones necesarias para obtener o restablecer los diversos bits de información afectados.

Una clase normalmente no debería tener más de tres o cuatro responsabilidades, y muchas clases tendrán sólo una o dos. Si una clase resulta tener más responsabilidades, piense si puede describirlas de manera más concisa, y si no, piense si sería mejor dividir las responsabilidades de la clase inmediatamente —si una clase no tiene responsabilidades, no debería habérsela inventado— pero las responsabilidades se revisarán cuando utilice las tarjetas CRC.

Demasiadas responsabilidades se corresponden con una débil cohesión en tu modelo; muchos colaboradores se corresponden con un fuerte acoplamiento. La utilización de las tarjetas CRC puede ayudar a identificar y fijar estos dos fallos en un modelo.

Al principio, puede que no sepa con qué otras clases tiene que colaborar una clase concreta para cumplir una responsabilidad; esto se aclara cuando se utilizan las tarjetas. Si un objeto de una clase colabora con un objeto de otra, lo hace enviándole un mensaje; así que una clase tiene una asociación con cada uno de sus colaboradores. La utilización de tarjetas CRC es la mejor manera de resolver en qué dirección debería ser navegable una asociación, y puede también ayudar a identificar las asociaciones, especialmente entre clases que no representan objetos del mundo real.

5.6.2 Utilización de tarjetas CRC en el desarrollo del diseño

Se pueden utilizar las tarjetas CRC para recorrer los casos de uso, resolviendo cómo el modelo de clases proporciona la funcionalidad requerida por los casos de uso, y dónde están los bits perdidos. (Se puede ver esto como el descubrimiento de colaboraciones e interacciones que llevan a cabo sus casos de uso. Se tratará la forma de almacenar esta información de forma permanente en la notación de UML en el Capítulo 9).

Una técnica que puede ser útil es la realización de un rol. Si se está trabajando en un equipo, cada persona puede desempeñar una o más responsabilidades de las tarjetas CRC, donde el conjunto completo de tarjetas representa las clases encargadas del aspecto más importante de las responsabilidades del sistema. Se puede comprobar la completitud del diseño trabajando a través de varios escenarios de los casos de uso relevantes.

Primero seleccione un escenario típico de un caso de uso. Por ejemplo, en el caso de uso `tomarPrestada copia de libro`, se empieza con el escenario de un prestatario que consigue tomar prestado un libro. Según se va volviendo más sólido el diseño, cualquiera que tenga la sospecha de que pueda haber algún problema con el diseño puede sugerir un escenario que ilustre dicho problema (por ejemplo, qué pasa si no hay ninguna copia disponible). La petición inicial se le da a una persona cuyas tarjetas CRC representan una clase cuyas responsabilidades incluyen la realización de un escenario; esto representa un objeto de esa clase recibiendo un mensaje del iniciador del escenario. Si el objeto necesita asistencia de uno de sus colaboradores, le enviará, en la implementación eventual, un mensaje solicitando que ejecute una operación. Cada operación que un objeto puede ejecutar debería formar parte de una de las responsabilidades de la clase del objeto; las responsabilidades de una clase pueden verse como un resumen de la operación que pueden ejecutar. Cuando se utilizan por primera vez las tarjetas CRC, probablemente sólo se considerará si es suficientemente razonable esperar que un colaborador en particular sea capaz de ayudar en la ejecución de una parte determinada de una responsabilidad; más tarde, se pueden examinar las interacciones con mayor detalle para diseñar la verdadera colección de operaciones para cada clase.

Si falta un enlace —hay que hacer algo, pero ninguna clase tiene la responsabilidad de hacerlo— esto significa que el diseño es defectuoso o incompleto. Puede que se necesite crear una nueva clase, cambiar las colaboraciones y responsabilidades de una clase existente, o ambas. Es importante recordar que los cambios que se realizan tienen que preservar la coherencia general de los modelos: evitar lanzarse a la solución obvia de un caso de un problema particular que se haya presentado, sin pensar en las otras implicaciones del cambio. Muchas veces es útil escribir una tarjeta nueva para una clase si se ha alterado sustancialmente la original.

Destacar que el sistema realmente trabaja por medio de la comunicación entre *objetos*, en vez de clases: hay que tener presente si una tarjeta CRC para una clase representa, siempre, al mismo objeto de la clase, o si están implicados varios objetos diferentes.

Un efecto lateral derivado de trabajar con ejemplos de este tipo es que se crea un espíritu de equipo y todo el mundo se siente partícipe en el diseño. ¡Naturalmente algunos equipos encontrarán esta técnica más útil que otros!

Como alternativa, especialmente cuando se trabaja solo, se pueden utilizar las tarjetas para que representen relaciones entre ellos —se puede considerar esto como hacer un borrador de un modelo de clases. Por ejemplo, a algunas personas les gusta agrupar las tarjetas que están relacionadas entre sí mediante generalización, con la más abstracta arriba, de forma que sólo se utiliza la clase especializada cuando la especialización particular es relevante.

5.6.3 Ejemplo de tarjeta CRC

La Figura 5.5 es un ejemplo obtenido del estudio del caso del Capítulo 3. Se empieza dibujando algunas tarjetas CRC para las clases `SocioBiblioteca`, `Libro` y `Copia`. Las decisiones tomadas se basan en la intuición sobre cómo colaborarán las instancias de estas clases. Entonces, se puede comprobar cómo se podría ejecutar un caso de uso particular.

SocioBiblioteca	
Responsabilidades	**Colaboradores**
Mantener los datos sobre las copias prestadas actualmente Atender peticiones para tomar prestados y devolver copias	Copia

Copia	
Responsabilidades	**Colaboradores**
Mantener los datos sobre una copia determinada de un libro Informar del correspondiente Libro cuando es prestado y devuelto	Libro

Libro	
Responsabilidades	**Colaboradores**
Mantener los datos sobre un libro Saber si hay copias disponibles para tomar prestadas	

Figura 5.5 Ejemplo de tarjetas CRC para la biblioteca.

Aunque el texto escrito en cada tarjeta es muy escaso, no es algo a obviar: el peso que tiene es el adecuado. Las tarjetas CRC llenas significan, bien responsabilidades expresadas de forma incorrecta, o bien clases no cohesivas.

Si se sigue a través del procesamiento de una petición realizada por un `PrestatarioLibro` para `Tomar prestada una copia de un libro`, se puede comprobar si sería necesario que estuviesen implicadas otras clases. Se pueden comprobar también los mensajes que se podrían enviar. Si un objeto no puede cumplir una responsabilidad, necesita bien añadirla a su propia definición o bien definir una colaboración con otra clase que la cumpla.

Se puede también examinar la relación de generalización entre `SocioBiblioteca` y `SocioPlantilla` viendo hasta qué punto comparten las responsabilidades y los colaboradores. Se encuentra que todas las responsabilidades de un `SocioBiblioteca` las comparte un `SocioPlantilla`, pero no al revés. Esto confirma la idea de que un `SocioPlantilla` es una especialización de un `SocioBiblioteca`. De manera más general, se pueden buscar oportunidades para *refactorizar* el modelo de clases en uno mejor.

5.6.4 Refactorización

La refactorización es el proceso de alterar el modelo de clases de un diseño orientado a objetos sin alterar su comportamiento visible. Por ejemplo, si en cualquier etapa uno se da cuenta de que una operación no encaja de forma apropiada en la clase en la que se encuentra —las responsabilidades no están situadas en las clases de la mejor manera posible— un paso de refactorización sería obligar a poner las operaciones donde deberían estar, actualizando cualquier documentación de código y diseño que se necesite modificar.

La norma ESCRIBA UNA SOLA VEZ es una buena fuente para obtener los pasos de refactorización. Por ejemplo, si uno se encuentra con que tiene dos clases con responsabilidades y comportamiento solapados, muchas veces se pueden tomar los comportamientos comunes y crear una nueva superclase de la que ambas hereden. En realidad, este proceso puede ayudar en uno de los aspectos más difíciles del análisis y el diseño orientados a objetos, a saber, en encontrar las abstracciones correctas que hagan el diseño limpio y robusto. Tales clases, a menudo, no destacan nada en el documento de requisitos —porque no son clases de dominio en sí, aunque expresan lo que varias clases de dominio tienen en común— por lo que la técnica de identificación de nombres no las encontraría. *Véase* el libro de Kent Beck [3] para ver ideas de refactorización.

P: Analice el ejemplo del préstamo de la biblioteca, utilizando las tarjetas CRC vistas y modifíquelas si fuera necesario.

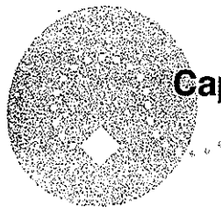
P: Intente realizar un análisis similar para devolver un libro y devolver una revista.

RESUMEN

Este capítulo ha presentado los diagramas de clases, que representan la estructura estática del sistema que se va a construir. Se ha tratado cómo identificar las clases y sus asociaciones, y el concepto de multiplicidad de una asociación. Se ha mostrado cómo se representan los atributos y las operaciones de una clase. A continuación, se ha cubierto la generalización, que puede ser implementada por ejemplo, por medio de la herencia. Finalmente, se ha tratado el rol del modelo de clases a través del desarrollo y se ha ilustrado el uso de las tarjetas CRC para validar un modelo de clases.

PREGUNTAS DE DISCUSIÓN

1. ¿Por qué las clases deben tener los nombres en singular? ¿Cree que hay alguna excepción?
2. ¿Cuáles son las ventajas y desventajas de decidir formalmente desde qué perspectiva se está dibujando un modelo de clases en particular?
3. ¿Bajo qué circunstancias se podrían tener clases en un modelo que no se correspondiesen con objetos del dominio? ¿Puede pensar en algún caso donde esto sea necesario en el ejemplo de la biblioteca?
4. ¿Cómo se establecen las asociaciones en la implementación? ¿Cómo describiría esto su modelo de clases, si fuese necesario? ¿Depende de la etapa de desarrollo? ¿Cómo?
5. Piense en las nociones estáticas y dinámicas de las asociaciones mencionadas en la Anotación Técnica de UML, de la sección 5.2. Construya un pequeño ejemplo de clases que se asocian dinámicamente, pero no estáticamente, y piense en las dos versiones del diagrama de clases de UML que obtenga adoptando cualquier noción. ¿Cuáles cree que son las ventajas y desventajas de cada convención? ¿Qué significa la *ausencia* de una asociación entre clases, en cada caso?



Capítulo 6

Más sobre los modelos de clases

En este capítulo se consideran otras funciones de los diagramas de clases de UML. Son menos importantes que las descritas en el Capítulo 5, y puede decidir saltarse este capítulo en una primera lectura. Sin embargo, una de las fortalezas de UML es su expresividad, y aquí se da una muestra de ello, cubriendo la mayoría (pero no todas) de las funciones de los diagramas de clases.

A lo largo del camino se tratarán algunos aspectos de UML que no son específicos de los modelos de clases, pero que aquí se utilizarán por primera vez. Estos son las *restricciones*, los mecanismos de principal extensibilidad de UML (*estereotipos, propiedades* y *valores etiquetados*), *interfaces, dependencias* y *paquetes.*

Empezaremos considerando información extra que puede almacenarse junto con la asociación entre dos clases.

6.1 Más sobre asociaciones

6.1.1 Agregación y composición

La agregación y la composición son tipos de asociación: en vez de simplemente mostrar que dos clases están asociadas se puede decidir mostrar más información sobre qué tipo de asociación tienen. La agregación y la composición son dos formas de almacenar que un objeto de una clase *es parte de* un objeto de otra clase.

Por ejemplo, la Figura 6.1, obtenida del estudio del caso del Capítulo 15, muestra que un `Módulo` es parte de un `CursoDeDoctorado`. Esta notación, con el diamante vacío, denota agregación, que es una manera general de denotar una relación todo-parte en UML. Destacar que el diamante se coloca en el extremo del todo, no de la parte. También se pueden utilizar todas las otras notaciones que van con la asociación. Por ejemplo, se pueden mostrar las multiplicidades

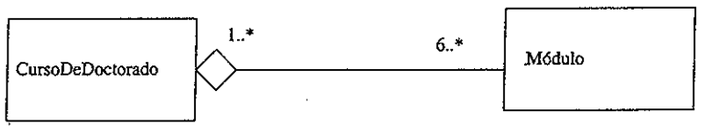


Figura 6.1 Una agregación.

como con una asociación normal. Destacar que a un objeto se le permite ser parte, a la vez, de varios objetos: en nuestro caso, un único `Módulo` podría ser parte de varios `CursosDeDoctorado` diferentes. (Por ejemplo, la Ingeniería del Software con Objetos y Componentes 2 es parte tanto del curso de doctorado de Ingeniería del Software como del de Informática).

Normalmente, uno no se toma la molestia de nombrar una asociación de agregación, ya que el nombre de la asociación sería "es una parte de" y esto está ya reflejado en la notación de agregación, por lo que no es necesario indicarlo también con palabras. Sin embargo, si ayuda a nombrar la asociación (por ejemplo, si se quiere describir la función de la parte en el todo) esto es perfectamente legal.

Desde la carencia de cualquier otra restricción nueva se ve que la agregación es fundamentalmente una idea conceptual: ver una agregación en un modelo de clases debería ayudar a comprender las relaciones entre las clases a nivel informal, pero no añade ninguna información formal sobre cómo deben implementarse o qué se puede hacer con ellas. (Si bien, *véase* la Pregunta de Discusión 47 más tarde en este capítulo: es discutible que alguna agregación implique algunas cosas. Aunque actualmente en UML no se hace).

La composición es un tipo especial de agregación que impone algunas restricciones más. En una asociación de composición, el todo *posee fuertemente* a sus partes: si se copia o borra el objeto todo, sus partes se copian o borran con él. (Para que esto se pueda implementar, la asociación debe ser navegable desde el todo hacia la parte, aunque UML no especifica esto de forma explícita). La multiplicidad en el extremo del todo en una asociación de composición tiene que ser 1 ó 0..1 —una parte no puede pertenecer a más de un todo por composición. El ejemplo con `Módulo` y `CursoDeDoctorado` no sigue estas restricciones, por lo que una composición no sería apropiada en este caso. Por otro lado, piense en una aplicación del juego tres en raya en el marco de trabajo considerado en el Capítulo 16, que de forma sensata debería implementarse en función de las clases `Cuadrado` y `Tablero`. Cada `Cuadrado` es parte de un `Tablero`, y no tendría sentido copiar o borrar un objeto `Tablero` sin copiar o borrar los objetos `Cuadrado` que forman el `Tablero`. Así que, en este caso, la composición es apropiada, y se muestra en la Figura 6.2. La composición se representa como la agregación, salvo que el diamante está relleno.

ADVERTENCIA

Desde nuestra experiencia, la gente nueva en el modelado orientado a objetos utiliza la agregación y la composición demasiado a menudo. Recuerde que ambos son tipos de asociación, por tanto, siempre que una asociación o composición sean correctas, lo es una simple asociación. En caso de duda, utilice una simple asociación.

Figura 6.2 Una composición.

Pregunta de Discusión 41

Piense en algunos casos donde la agregación o composición sean apropiadas. Por ejemplo, ¿Qué pasa con la relación entre un `Empleado` y un `Equipo`? ¿Entre `Rueda` y `Coche`? ¿`Cuenta` y `Cliente`? ¿Puede describir diferentes contextos en los que habría clases con estos nombres, donde, debido a las diferencias en el contexto, serían apropiadas diferentes relaciones?

Pregunta de Discusión 42

Si sabe C++ u otro lenguaje orientado a objetos que pueda hacer referencia a los objetos bien *por valor* o bien *por referencia*, sabe que a menudo la gente distingue entre agregación y composición diciendo que se tiene una agregación si el todo tiene una referencia o puntero a la parte, y composición si el todo contiene la parte por valor. Piense por qué es esto, y si es apropiado.

Consejo mnemotécnico

El símbolo más fuerte, el diamante sólido, representa la relación más fuerte, la composición. Cuando se borra un diamante sólido, hay que borrar el contenido del símbolo del diamante, así como el borde, tal y como cuando se borra un objeto compuesto hay que borrar las partes al igual que el todo[1].

6.1.2 Roles

Se ha visto cómo nombrar una asociación. Muchas veces se puede leer un nombre de asociación en ambas direcciones ("está tomando", "es tomado por"). A veces, sin embargo, es más legible tener nombres separados para los roles que desempeñan los objetos en la asociación, o bien como nombre de la asociación o bien en lugar del mismo. Por ejemplo, la Figura 6.3 deja claro que el rol del `Estudiante` en esta asociación es `dirigido`, que podría ser útil, por ejemplo, si `dirigido` fuese un término comúnmente utilizado en un documento adjunto. Se puede poner tanto el nombre de la asociación como los nombres de rol, aunque lo normal será omitirlos en la mayoría de los casos.

Pregunta de Discusión 43

¿Sería útil dar nombres de rol a la asociación entre `Estudiante` y `Módulo`? ¿Por qué?

[1] Gracias a Ben Kleiman por sugerir esto.

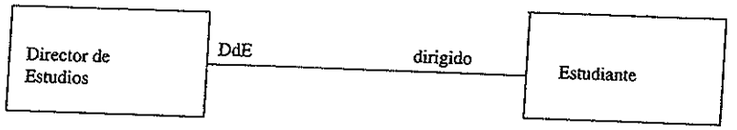


Figura 6.3 Una asociación con nombres de rol.

6.1.3 Navegabilidad

Considere una situación en la que se tiene una asociación entre dos clases, por ejemplo, entre `Módulo` y `Estudiante`, tal y como se muestra en la Figura 6.4. El diagrama almacena que:

- Para cada objeto de la clase `Estudiante` hay seis objetos de la clase `Módulo` que están asociados con el `Estudiante`;
- Para cada objeto de la clase `Módulo` hay algunos objetos `Estudiante` (el número de estudiantes no está especificado) asociados con el `Módulo`.

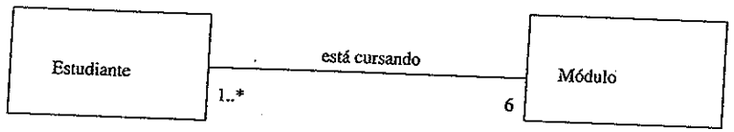


Figura 6.4 Asociación sin navegación.

Sin embargo, todavía *no* almacenan si sería posible obtener estos objetos en ambas direcciones o sólo en una. ¿Debería un objeto `Estudiante` ser capaz de enviar mensajes a los objetos `Módulo` asociados con él? ¿Debería un objeto `Módulo` ser capaz de enviar mensajes a los objetos `Estudiante` que representan a los estudiantes del curso? ¿Y ambos?

Se puede poner una flecha en uno o en ambos de los extremos de la línea de la asociación para representar que es posible enviar mensajes en la dirección de la flecha. Por ejemplo, la Figura 6.5 indica que un objeto `Módulo` puede enviar mensajes a los objetos que representan a los estudiantes del `Módulo`, pero no al revés.

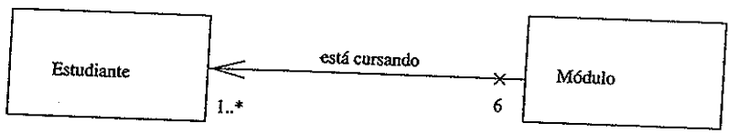


Figura 6.5 Asociación con navegación en un sólo sentido.

Se dice que `Módulo` *conoce a* `Estudiante`, pero no al revés. Una asociación como esta debería ser implementada, por ejemplo, permitiendo a `Módulo` tener un atributo (`estudiantes: ColecciónEstudiante`) que sea una colección de objetos correspondiente a los estudiantes que reciben el curso. Esto tiene consecuencias tanto buenas como malas. La aplicación puede necesitar esta información para estar disponible fácilmente: en nuestro caso, por ejemplo, se tienen que crear listas de estudiantes para cada curso, ya que los profesores adjuntos las necesitan, y la manera más fácil de hacer esto es permitir al objeto Módulo recuperar un conjunto de nombres de estudiantes a través del envío de un mensaje a cada objeto de tal colección de estudiantes. Sin embargo, si la clase A conoce a la clase B, entonces es imposible reutilizar la clase A sin la clase B; así no se introduciría la navegabilidad hasta que lo requiera la aplicación actual o haya una buena razón para pensar que se necesitará en el futuro. A veces es fundamental permitir la navegabilidad en ambos sentidos a lo largo de la asociación, pero tal decisión debe ser justificada en cada caso.

Pregunta de Discusión 44

(Después de considerar el Capítulo 15). ¿Debería esta asociación ser navegable también en la otra dirección —esto es, debería el `Estudiante` conocer al `Módulo`? ¿Por qué?

La notación cruzada para mostrar la no-navegabilidad es nueva en UML2. Utilizando tanto flechas como cruces es posible ser totalmente explícito sobre cómo las asociaciones pueden ser recorridas. Sin embargo, no se deseará o podrá siempre incluir toda esta información. Cuando se construye el modelo de clases inicial, se debería (de forma adecuada) no tener todavía decidido como debería ser la navegabilidad. Incluso cuando se han tomado todas las decisiones, no se debería querer almacenarlas explícitamente: haciéndolo dificultoso, y muchas herramientas de UML no permiten todavía el uso de la notación cruzada. UML sugiere diferentes convenios que podrían ser útiles para ahorrar esfuerzo y reducir el desorden en los diagramas. Por ejemplo, la ausencia de una flecha podría significar no-navegabilidad, o podría significar que la navegabilidad está sin especificar. En este libro no utilizaremos habitualmente la notación cruzada. Si un diagrama de clases muestra alguna flecha de navegabilidad se puede suponer que todas están incluidas; sino, no está especificada la navegabilidad.

Pregunta de Discusión 45

¿Cuándo, si es que hay alguna vez, podría una asociación no ser navegable en ninguna dirección?

Pregunta de Discusión 46

¿Cuándo debería decidirse la navegabilidad?

Pregunta de Discusión 47

Según UML, la navegabilidad de una asociación es independiente de si la asociación es una agregación, una composición o ninguna de ellas. ¿Qué navegabilidad cree que tiene una asociación de agregación? ¿Y una composición? ¿Puede afirmar qué tipo de navegabilidad debe tener *siempre* cualquier tipo de asociación para tener sentido?

6.1.4 Asociaciones calificadas

De vez en cuando es útil dar más detalles de los que se tienen sobre una asociación. Considere de nuevo la aplicación del tres en raya en el marco de trabajo del juego descrito en el Capítulo 16, que está implementado utilizando las clases `Cuadrado` y `Tablero`, y suponga que se identifica que la relación entre `Cuadrado` y `Tablero` es a través de los atributos `fila` y `columna`, cada uno de los cuales toma un valor entre 1 y 3. Si nos olvidamos por ahora del hecho de que la asociación es una composición (recuerde que una asociación es siempre correcta cuando una agregación o composición lo son), la asociación puede aparecer como en la Figura 6.6.

Figura 6.6 Asociación sencilla entre `Cuadrado` y `Tablero`.

Sin embargo, no capta la idea de localizar los nueve `Cuadrados` dando los nueve pares de valores de los atributos `fila` y `columna`. Para hacer esto se utiliza una *asociación calificada* como la mostrada en la Figura 6.7.

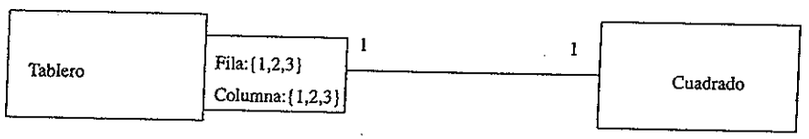


Figura 6.7 Asociación calificada.

El 1 en el extremo derecho, junto a `Cuadrado`, especifica que si se tiene un objeto `Cuadrado`, llamado *b*, y se especifican los valores tanto para el atributo `fila` como para el atributo `columna`, entonces hay exactamente un objeto `Cuadrado` asociado al objeto Tablero *b*. El 1 en el extremo del `Tablero` significa lo de siempre: cada objeto de la clase `Cuadrado` está exactamente en un `Tablero`.

Fundamentalmente, para cada `Tablero` se tiene una tabla de búsqueda en la que se localiza un `Cuadrado` por su `fila` y `columna`. Las condiciones descritas aseguran fácilmente que cada `Cuadrado` aparezca en una sola tabla de búsqueda una sola vez. Cabe destacar que, en general, puede haber más de un objeto para un valor concreto de la clave de búsqueda: los elementos en la tabla de búsqueda pueden ser conjuntos, no sólo elementos únicos.

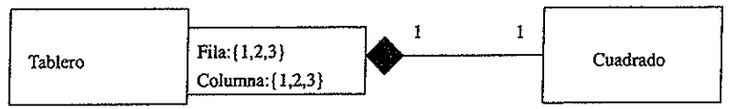


Figura 6.8 Composición calificada.

En realidad se puede combinar la notación de asociación calificada con otros adornos de las asociaciones; por ejemplo se puede añadir información de que esta asociación particular es una composición, tal y como aparece en la Figura 6.8.

¡Puede que ya se haya dado cuenta de que astutamente no se ha dicho a qué clase pertenecen los atributos `fila` y `columna`! Podrían ser atributos de `Cuadrado`; pero formalmente son atributos de la asociación. Cada enlace entre un `Tablero` y un `Cuadrado` (recuerde que un enlace es una instancia de una asociación) tiene valores para `fila` y `columna`, que identifican dónde se encuentra el `Cuadrado` dentro del `Tablero`.

P: (Obtenido de un ejemplo de la guía de notación de UML). Dibuje una asociación entre `Persona` y `Banco` para almacenar el hecho de que una `Persona` puede estar asociada con muchos Bancos, pero que dados un `Banco` y un número de cuenta, hay como mucho una `Persona` con ese número de cuenta en ese banco.

6.1.5 Asociaciones derivadas

Mientras se desarrollan diagramas de clases, a menudo surge la pregunta sobre si se necesita mostrar una asociación o si es suficiente deducir su existencia a partir de algún elemento del diagrama. Por ejemplo, si un `Estudiante` está asociado con `Módulo` por medio de *está cursando* y `Módulo` está asociado con `ProfesorAdjunto` a través de *enseña curso*, ¿Sería necesario también mostrar una asociación *enseña estudiante* entre `ProfesorAdjunto` y `Estudiante`, que relacionara un profesor adjunto con todos los estudiantes que reciben los cursos que enseña ese profesor adjunto?

Pregunta de Discusión 48

¿Cuáles son las ventajas y desventajas de hacer esto?

Se puede hacer esto o no, o se puede utilizar la tercera opción que proporciona UML, que consiste en mostrar esa asociación como una *asociación derivada*. En otras palabras, existe automáticamente una vez que se han implementado las asociaciones principales: el diseñador no necesita tener en cuenta de forma separada cómo implementar esta asociación. Una asociación derivada se muestra utilizando un signo "/" delante de su nombre, como en la Figura 6.9. (Los triángulos negros, a propósito, pueden utilizarse sobre cualquier nombre de asociación e indican, simplemente, la dirección que describe el nombre de la asociación).

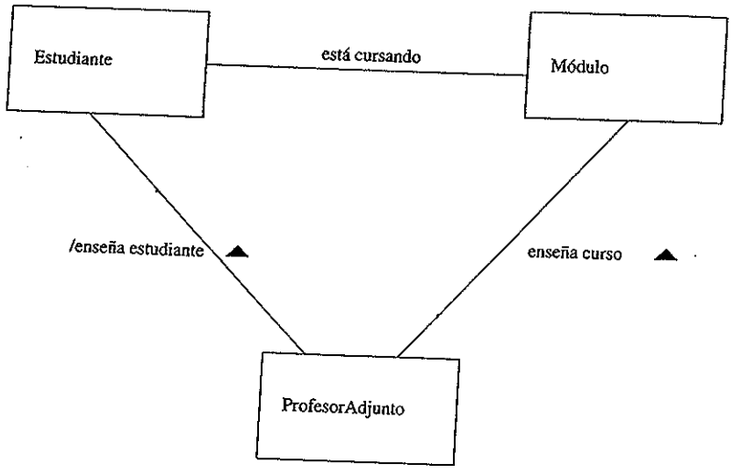


Figura 6.9 Una asociación derivada.

ANOTACIÓN TÉCNICA DE UML

En UML puede haber *elementos derivados* de diferente manera que las asociaciones derivadas. En general, un elemento derivado se distingue del elemento normal añadiendo un símbolo "/" delante de su nombre. Por ejemplo, se puede indicar que un atributo de una clase es derivado —esto es, que se puede calcular su valor en un determinado objeto si se conoce el valor de todos los atributos normales del objeto, y se tiene acceso a otros objetos que conoce— poniendo un símbolo "/" delante del nombre del atributo en el icono de clase. (Puede que quede más claro, sin embargo, definir una operación de la clase, sin argumentos y cuyo valor de retorno es un valor derivado. Es cuestión de gustos).

En el ejemplo que se ha dado hay una sola posibilidad sensata de lo que realmente tiene que ser una asociación derivada. Un `ProfesorAdjunto` *p* está asociado mediante *enseña estudiante* con un `Estudiante` *e* si, y sólo si, hay algún `Módulo` *m* tal que, tanto *e* está asociado con *m* mediante *está cursando* como *p* está asociado con *m* mediante *enseña curso*. Si se quiere almacenar esto de forma explícita en el diagrama, se puede bien escribirlo en lenguaje natural en una anotación, o bien escribirlo de manera formal.

6.1.6 Restricciones

Una restricción es una condición que tiene que ser satisfecha por cualquier implementación correcta de un diseño. Por ejemplo, un sistema con las asociaciones *enseña curso, está cursando* y *enseña estudiante* tiene un error si la condición que se añade en la sección anterior no se satisface siempre. Sin embargo, las restricciones pueden ser más generales que esto. Pueden restringir elementos del modelo de forma individual, o colecciones de elementos del modelo. Uno de los usos más comunes —y seguros— es expresar un *invariante de clase*. Por ejemplo:

```
{self.númeroEstudiantes >50 implica (no(self.aula = 3317))}
```

como invariante de la clase `Módulo` indica que para cada objeto de la clase `Módulo` se cumple siempre que si el número de estudiantes matriculados en el curso es mayor que 50 entonces el curso no tendrá lugar en el aula 3317 (probablemente, porque el aula 3317 tiene sitio sólo para 50 personas).

Esta restricción formal está escrita en OCL (*Object Constraint Language,* en Lenguaje de Restricción de Objetos, LRO), que es el Lenguaje de Restricción de Objetos que UML adopta para sus propósitos. *Véase* el Panel 6.1 para más información sobre OCL.

Otra situación común en la que las restricciones pueden ser útiles es aquella en la que hay un "o exclusivo" entre dos relaciones de asociación: un objeto forma parte de (un enlace que es una instancia de) exactamente una de las asociaciones. Por ejemplo, en el ejemplo de la biblioteca del Capítulo 3, se asume que, aunque puede haber varias copias de un libro, había una sola copia de una revista. Ahora supongamos que se quiere modelar un sistema en el que cada objeto `Copia` representa, bien una copia de un `Libro`, o bien una copia de una `Revista`. Se podría comenzar con el diagrama mostrado en la Figura 6.10; pero esto no excluye la posibilidad disparatada de que una `Copia` podría estar asociada tanto con un `Libro` como con una `Revista`, o con ninguno. Para hacer esto se puede utilizar una restricción `xor`, tal y como aparece en la Figura 6.11.

La restricción `xor` no está escrita en OCL; es una restricción especial predefinida que es parte de UML.

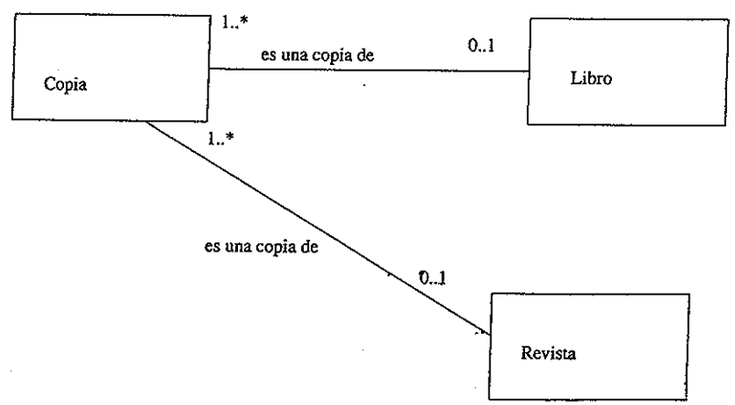


Figura 6.10 Un diagrama bajo restricción.

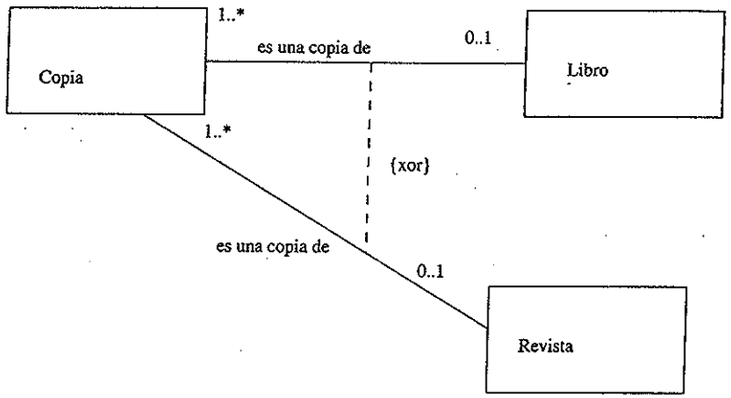


Figura 6.11 Utilizando una restricción xor.

Pregunta de Discusión 49

¿Se le ocurre alguna forma de solucionar este problema sin introducir una restricción? Consejo: piense en la posibilidad de introducir una clase extra.

El hecho de que UML permita añadir restricciones de una manera bastante general aumenta su poder de forma importante. Sin embargo, ¡como siempre, es importante no dejarse llevar por potentes funciones! Cualquier persona tiene que poder leer cualquier diagrama que se escriba, y eso más difícil cuanto más complejo sea el diagrama —se deberían utilizar poco las restricciones—, considerando en cada caso si es sensato introducir más información. Hay también razones técnicas para evitar los diseños que necesitan utilizar restricciones, las cuales limitan varios elementos del modelo que no están contenidos dentro de una clase; tal y como apunta Ian Graham, tales restricciones indican dependencias entre los elementos del modelo restringido, que pueden impedir tanto el mantenimiento como la reutilización.

Pregunta de Discusión 50

En realidad, las restricciones pueden expresar parte de la información que de forma general se expresa en UML utilizando una notación especializada más conveniente. ¿Cómo podría especificar la multiplicidad de una asociación utilizando restricciones en vez de la notación normal de multiplicidad?

PANEL 6.1 OCL (*Object Constraint Language,* en español, Lenguaje de Restricción de Objetos, LRO)

El Lenguaje de Restricción de Objetos se pretende que sea:

- formal, de manera que las restricciones que describa no sean ambiguas.
- fácil de utilizar, de forma que todo desarrollador lo utilice para escribir restricciones.

OCL surgió a partir del método Syntropy desarrollado por Steve Cook y John Daniels, y ha sido desarrollado por IBM como lenguaje de modelado de negocio. Se utiliza en los documentos de semántica de UML para especificar las restricciones para que los modelos de UML estén correctamente formados, al igual que está disponible para los usuarios de UML para indicar restricciones en sus propios modelos.

La especificación de UML (*véase* la página web de este libro) incluye detalles completos de OCL, y hay disponible también un libro [52].

Combinar los dos objetivos de OCL es muy difícil, y no se tiene la convicción de que OCL alcance, todavía, cualquier objetivo. No tiene una semántica formal, por lo que no puede decirse que sea un lenguaje formal. Por el contrario, no está claro que un lenguaje formal con el tipo de potencia requerido pueda llegar a ser realmente fácil de aprender. Mientras le animamos a que aprenda más sobre OCL, nos gustaría darle también un aviso:

ADVERTENCIA

Mientras la utilización de notaciones formales puede ser útil, ya que las personas implicadas realmente saben cómo leer y escribir en él, una restricción en español es mucho más útil que una restricción en un lenguaje formal con errores, o cuya intención los lectores no entienden. Por lo que si usted y sus compañeros no están seguros de saber escribir algo en OCL, utilicen el español o su lengua habitual.

6.1.7 Clases asociación

Algunas veces es tan importante la manera en que están asociados dos objetos como los objetos en sí. Piense, por ejemplo, en la asociación entre `Estudiante` y `Módulo`. ¿Dónde debería almacenar el sistema las notas del estudiante en ese curso? Las notas están relacionadas realmente con el par formado por un estudiante y un módulo. Se podría pensar en implementar un objeto por cada par: el objeto almacenaría las notas del estudiante en ese curso, evitando confundirlos conceptualmente con las notas de otro estudiante de ese curso, o con las notas de ese estudiante en otro curso. Esto significa tratar la asociación entre las clases `Estudiante` y `Módulo` como una clase; por supuesto una instancia de la asociación relaciona un estudiante con un módulo, y entonces se dice que son datos adjuntos a ese enlace. Probablemente se quiera también realizar operaciones, aunque sólo sea para obtener y establecer las notas. El resultado es algo que es tanto una clase como una asociación, que se denomina *clase asociación*. La notación aparece en la Figura 6.12.

El icono de clase y la línea de asociación tienen que tener el mismo nombre, ¡porque son lo mismo! Esto presenta un pequeño problema, ya que las asociaciones normalmente tienen como nombre frases verbales y las clases, frases nominales. (También, si se utiliza la convención de mayúsculas y minúsculas que dice que los nombres de las asociaciones se escriben en minúscula y que los nombres de las clases van en mayúsculas, hay que buscar un caso especial para las clases asociación, ¡ya que no se pueden cumplir ambas convenciones a la vez!) Se puede pensar un nombre mejor que *está cursando* para cubrir ambos.

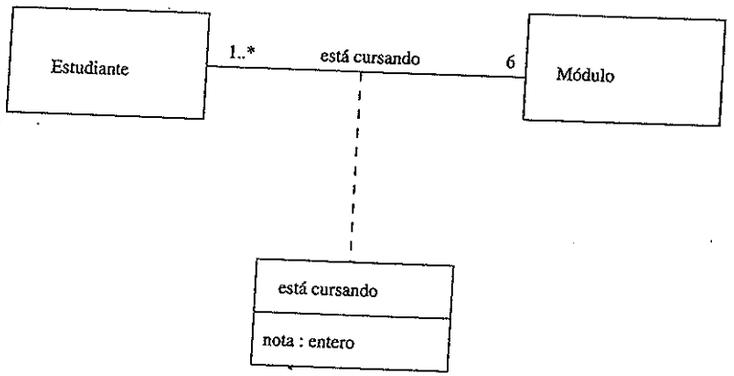


Figura 6.12 Una clase asociación.

Por supuesto, hay otras formas de almacenar la misma relación entre `Estudiante` y `Módulo`: se podría pensar en una nueva clase, por ejemplo `Nota`, y asociarla con ambas clases como siempre, tal y como se muestra en la Figura 6.13.

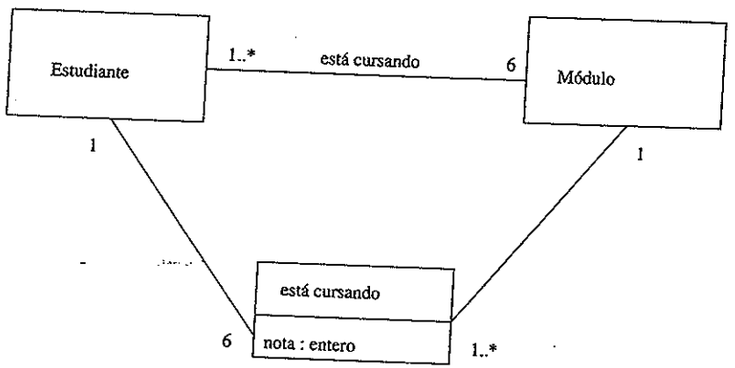


Figura 6.13 Evitando una clase asociación.

Pregunta de Discusión 51

¿Cuáles son las ventajas y los inconvenientes de ambas aproximaciones? Piense: qué pasaría si el mismo estudiante cursa el mismo módulo dos veces por alguna razón; si se necesita una restricción en cualquiera de los casos; si habría, en realidad, alguna diferencia en las implementaciones que permitiría cualquier notación.

6.2 Más sobre clases

En el Capítulo 2 se destacó que una clase realmente cumple dos propósitos: define la interfaz que los objetos presentan al resto del sistema, y define una implementación de dicha interfaz. Algunas veces es importante para el diseño separar los dos conceptos, particularmente para distinguir entre niveles diferentes de *dependencia* entre los elementos del modelo.

En realidad, hay tres variantes de la idea de clase —tres maneras distintas de clasificar los objetos— que se considerarán juntas. Varían en la cantidad de información que contienen sobre los atributos, operaciones, y sus implementaciones. En resumen, son:

1. **Interfaz**. Una interfaz especifica una lista de operaciones que tiene que proporcionar todo aquello que cumpla la interfaz. No tiene asociadas las implementaciones con ninguna de las operaciones. Una interfaz no especifica nada sobre el estado de un objeto que lo incluye; por lo que no tiene atributos y ninguna asociación puede ser navegada desde una interfaz. Se tratarán las interfaces con mayor detalle en la siguiente subsección.
2. `<<implementation class>>`. Una clase que tiene el estereotipo `<<implementation class>>` define la implementación física de sus operaciones y atributos. Puede implementar un tipo.

Cualquier objeto tiene exactamente una clase implementación, aunque puede tener varios tipos e incluir varias interfaces. Normalmente se trabaja con clases sin estereotipos, pero algunas veces la precisión extra que proporcionan los estereotipos es útil.

ANOTACIÓN TÉCNICA DE UML

Anteriores ediciones de este libro también incluían clases con el estereotipo `<<tipo>>` en este apartado. La diferencia entre tal clase y una interfaz, es que las interfaces pueden no tener atributos en UML 1.x, y las clases `<<tipo>>` sí podrían. Ambas ideas todavía existen en UML2, pero ahora las interfaces también tienen permiso para tener atributos. Nosotros pensamos que esto hace a las clases `<<tipo>>` prácticamente redundantes.

PANEL 6.2 Estereotipos

Un *estereotipo* es la manera que tiene UML de adjuntar clasificaciones extra a los elementos de un modelo. Es una de las formas que UML ha hecho ampliable. Describe un elemento del modelo y se sitúa en el diagrama cerca del elemento. Por ejemplo, la Figura 6.14 muestra el estereotipo `<<interface>>` sobre la clase símbolo y el estereotipo `<<use>>` sobre una flecha[2], de dependencia. Esto proporciona información extra sobre la clase y sobre la dependencia.

Algunos estereotipos están predefinidos en UML; están disponibles automáticamente y se pueden redefinir. Un ejemplo es `<<implementation class>>`. Más interesante, es

[2] Técnicamente, en la especificación de UML2, ni unos ni otros son realmente un estereotipo —son aplicaciones de la notación de estereotipos para indicar metaclases. Sin embargo, esto hace que no haya diferencia entre la notación o el uso de UML, por eso se obviará la distinción.

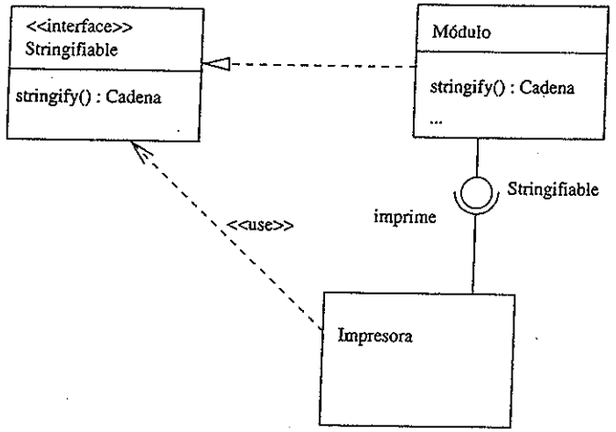


Figura 6.14 Una interfaz y su utilización.

que se pueden definir estereotipos propios para expresar cualquier otra clasificación útil. Por ejemplo, si se estuviese implementando una aplicación que tuviese clases persistentes podría decidir, correctamente, definir un estereotipo `<<persistente>>` para indicar qué clases son persistentes. UML también permite definir un nuevo icono gráfico para representar una clase `<<persistente>>`. El equipo del proyecto necesita, por supuesto, ponerse de acuerdo sobre dónde documentar los estereotipos.

En este libro se intenta comentar la realidad, siempre que se utiliza un estereotipo que no es una parte predefinida de UML.

6.2.1 Interfaces

Una *interfaz* especifica algunas operaciones de algunos elementos del modelo, tales como una clase, que son visibles fuera del mismo. No necesita especificar *todas* las operaciones que soporta el elemento, por lo que el mismo elemento podría incluir varias interfaces diferentes.

En UML2, una interfaz podría también especificar algunos atributos y asociaciones. Esto puede ser útil esporádicamente, pero es necesario utilizarlo con cuidado. La clave es *especificar* —si hay distintos modos en los que una clase podría aparecer para aportar un atributo, por ejemplo, la interfaz no restringe esta elección.

Todos los elementos de una interfaz son públicos. Una interfaz se define en un diagrama de clases utilizando un rectángulo como el del icono de clase, con las operaciones listadas en un compartimento del rectángulo como si fuera una clase. El icono se marca con `<<interface>>`, y no tiene un compartimento de atributo, porque una interfaz no puede tener atributos. Por ejemplo, la Figura 6.14 define una interfaz que se satisface cuando se entiende el mensaje "stringify" y devuelve una cadena. El diagrama también muestra cómo se utilizan las interfaces en UML.

La clase `Módulo` *concuerda* (o *realiza*, o *soporta*) la interfaz; esto es, `Módulo` tiene un método `stringify` del tipo correcto. Esto aparece de dos maneras, éstas son:

1. El círculo pequeño etiquetado `Stringifiable` junto al icono `Módulo`;
2. La flecha desde `Módulo` a `Stringifiable`. Hay que destacar la utilización de la flecha de *realización*: es igual que una flecha de generación excepto por la línea de guiones. Tal y como sugiere la notación, incluir una interfaz puede verse como una forma de herencia débil. El `Módulo` proporciona al menos las operaciones especificadas en `Stringifiable`, y puede proporcionar más, así como con la herencia entre clases. Sin embargo, `Módulo` *tiene que* proporcionar sus propias implementaciones, ya que la interfaz `Stringifiable` no tiene ninguna implementación: sólo se heredan las especificaciones de las operaciones.

Por supuesto, no es necesario mostrar la misma información dos veces. Puede ser conveniente omitir la flecha de realización, especialmente cuando un diagrama contiene muchas clases que realizan la misma interfaz. El icono de clase de la interfaz tiene que estar ahí, para definir lo que significa dicha interfaz.

La clase `Impresora` *depende* sólo *de la* interfaz `Stringifiable`; esto es, `Impresora` no se preocupa de ninguna otra función de una clase `Modulo`; siempre que proporcione el método `stringify` la `Impresora` puede utilizarla. Esto se muestra a través de medio circulo unido a la clase `Impresora`. Por razones obvias, el círculo y el medio círculo son llamados *notación de conexión y bola*. La parte de "conexión" es nueva en UML2; ésta es muy conveniente sobre todo si el diseño tiene el uso de muchas interfaces.

Hablando estrictamente, debería haber una asociación entre las clases `Impresora` y `Módulo`, especialmente si tomamos una vista estática de las asociaciones (como vimos en el Capítulo 5). Vimos una en anteriores ediciones de este libro. Sin embargo, es un poco chapucero tener la clases unidas por una asociación así como por la bola y notación de conexión y bola. Para la mayoría de los casos, sólo la notación de conexión y bola es bastante informativa, de tal forma que hemos decido omitir la asociación.

El diagrama también muestra la flecha de dependencia con el estereotipo `<<use>>` indicando que `Impresora` depende de la interfaz `Stringifiable`. Como la flecha de realización, esto puede ser omitido del diagrama desde que los *sockets* traen la misma información. La Figura 6.14 ha sido diseñada para mostrar toda la notación utilizada con interfaces. La Figura 6.15 ilustra con notación más parsimoniosa. Cuando usamos muchas interfaces, es común situar todas las cajas de interfaz juntas, siempre en filas ordenadas desde la parte principal del diagrama de clases. Esto facilita el buscar qué operaciones aporta cada interfaz, conservando el diagrama de clases principal ordenado. El diagrama de clases principal puede entonces utilizar sólo los nombres de las interfaces, con la notación de conexión y bola.

P: ¿Cómo se implementa una interfaz en su lenguaje? Escriba el esqueleto del código que se corresponde con la Figura 6.14.

Pregunta de Discusión 52

¿Por qué son todos los elementos de una interfaz públicos?

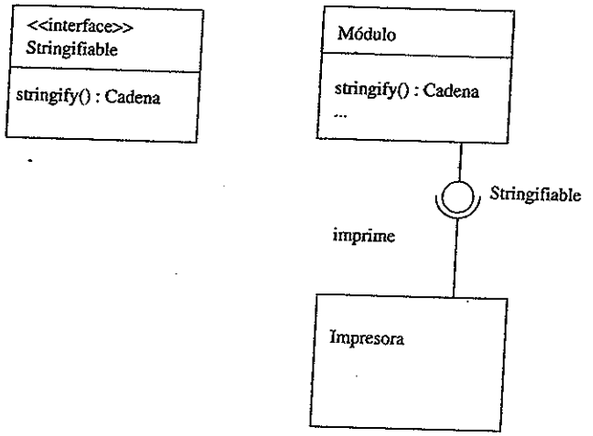


Figura 6.15 Notación más escasa para la dependencia de interfaz.

Tal y como se verá en el Capítulos 13, los componentes pueden incluir interfaces también.

ANOTACIÓN TÉCNICA DE UML

En realidad, cualquier *clasificador* puede incluir una interfaz. Los clasificadores incluyen, por ejemplo, actores y casos de uso, así como las clases, subsistemas y componentes que se han mencionado. La notación es la misma que se ha descrito; por ejemplo, se puede adjuntar un círculo con un palito al símbolo del caso de uso. La definición en sí de la interfaz aparece siempre como un rectángulo. Nos hemos referido al rectángulo como el símbolo de la clase, pero de hecho en UML2 es más general el símbolo para cualquier clasificador. Por defecto, éste es leído como una clase.

6.2.2 Clases abstractas

Una idea relacionada es la *clase abstracta*. Una clase abstracta, que se enseña utilizando la *propiedad* `{abstract}` en el icono de clases, puede tener implementaciones definidas para algunas de sus operaciones. Sin embargo, decir que es abstracta significa que, al menos, una de sus operaciones no tiene definida implementación. Por lo tanto, no se puede instanciar una clase abstracta. Una clase abstracta en la que *ninguna* de las operaciones tiene una implementación, y en la que no hay atributos, es, efectivamente, lo mismo que una interfaz. Los programadores de C++ a menudo utilizan las clases abstractas para hacer lo mismo que un programador de Java hace con las interfaces de Java. Una clase abstracta puede ser utilizada correctamente para implementar una interfaz de UML en una aplicación de C++. En este caso, una clase que implementa una interfaz hereda de la clase abstracta.

Pregunta de Discusión 53

En el Capítulo 2 se dijo que las clases a menudo tienen un rol de fábrica. ¿En qué sentido una clase abstracta es una fábrica de objetos?*

* Gracias a Bend Kahlbrandt por presentar esta cuestión.

PANEL 6.3 Propiedades y valores etiquetados

Acabamos de ver un ejemplo en el que se añade a un elemento, una *propiedad* para dar más información sobre él. Este es un potente mecanismo en UML, y es la otra manera fundamental, aparte del estereotipado, en el que UML se ha extendido.

Al igual que los objetos tienen valores para sus atributos, los elementos del modelo, como son las clases, tienen valores para sus propiedades. Las propiedades fundamentalmente tienen que ver con el *modelo* en vez de con el *sistema* implementado. Por ejemplo, en un modelo de UML todas las clases tienen una propiedad lógica `esAbstracta`, que se supone que tendrá el valor `verdadero` si la clase es abstracta y `falso` en otro caso. No hay ninguna sugerencia de que deba haber un atributo real `esAbstracto` en cualquier parte del sistema: la propiedad sólo proporciona al diseñador una manera sistemática de almacenar la decisión de diseño de que ésta es una clase abstracta.

Los diferentes tipos de elementos del modelo de UML tienen disponibles distintas propiedades para almacenar las decisiones apropiadas. Otra, particularmente útil, es la propiedad de operaciones `esPregunta`. Si un desarrollador especifica que una operación tiene su propiedad `esPregunta` a `verdadero`, almacena la decisión de diseño en la que la llamada a la operación no debería afectar al estado del sistema de ninguna manera. Tales operaciones pueden utilizarse en el modelo siempre que sea conveniente y todas las veces que se quiera, sin miedo a provocar efectos laterales no deseados. Por ejemplo, pueden utilizarse en condiciones o restricciones sin causar confusión.

Cualquier propiedad puede escribirse en un diagrama añadiendo una etiqueta `{nombrePropiedad = valor}` cerca del nombre del elemento sobre el que se aplica la propiedad. Debido a que muchas propiedades tienen valor lógico con nombres `esAlgo`, UML proporciona una forma abreviada para ellas. Se podría escribir `{esAbstracto = verdadero}`, pero en su lugar se puede escribir solamente `{abstract}`. Destacar la similitud con la manera en que se escriben las restricciones: en ambos casos, la expresión se escribe entre llaves cerca del nombre del elemento. Se podría pensar en una propiedad como un tipo de restricción.

Las propiedades predefinidas de cada tipo de elemento del modelo están descritas en el documento de Semántica de UML [48]. Uno mismo también puede definir sus propios *valores etiquetados*. Esto es, para cualquier elemento de un modelo se puede definir un nombre (una etiqueta) que debería contener un valor. Por ejemplo, si se quisiera almacenar quién escribió el código de una clase y quién lo revisó, se podrían definir dos etiquetas autor y revisor para aplicarlas a cada clase. El diagrama de clases puede almacenar la información; por ejemplo, insertando `{autor = "Perdita Stevens", revisor = "Stuart Anderson"}` cerca del nombre de la clase. Por supuesto, esto sólo será realmente útil si se tiene tanto un conjunto de etiquetas acordadas por el equipo de desarrollo, como algún soporte de herramientas para gestionar y mostrar las etiquetas.

¿Cuál es la diferencia entre definir un nuevo valor etiquetado, y definir un nuevo estereotipo? Estereotipar es una opción más potente, con peso, que es especialmente útil cuando se quieren hacer varias especializaciones de un tipo de elemento del modelo. Definir un valor etiquetado es un mecanismo con menos peso, para cuando simplemente se quiere asociar algo más de información a un elemento.

6.3 Clases parametrizadas

¡Una clase parametrizada realmente no es un tipo de clase! Puede considerarse como un tipo de función: en vez de tomar algunos valores y retornar un valor, una clase parametrizada toma algunas clases y devuelve una clase. A veces se le llama *plantilla*: la idea es que tiene algunas ranuras en las que se ponen clases, para obtener una nueva clase. El ejemplo clásico es `Lista<T>`, el cual, dada una clase `C` para sustituir el parámetro formal `T`, describe la clase de listas de objetos `C`. Por ejemplo, un objeto de clase `Lista<Estudiante>` representa una lista de estudiantes, de igual modo un objeto de clase `Lista<Juego>` representa una lista de juegos. Por supuesto, se podrían implementar nuevas clases `ListaEstudiante` y `ListaJuego` sin necesitar ninguna facilidad especial. Sin embargo, las listas de estudiantes y las listas de juegos tendrán mucho en común: por ejemplo, ambas proporcionarán operaciones para añadir y eliminar elementos de la lista, y el código para ambas es prácticamente el mismo, sin pensar si los elementos de las listas son `Estudiantes` o `Juegos`. Una clase parametrizada permite aprovecharse de este hecho reutilizando la misma clase parametrizada en ambos casos. Definiendo una clase parametrizada una sola vez, y utilizándola dos, se puede ahorrar esfuerzo tanto en desarrollo como en mantenimiento. En UML se muestra una clase parametrizada utilizando una variante del símbolo de clase, el cual tiene un pequeño rectángulo punteado en la esquina superior derecha, donde se listan los parámetros formales de la clase parametrizada. Los tipos de los miembros de la clase pueden (y casi siempre lo hacen) mencionar el parámetro formal. Hay dos formas de indicar que una clase es el resultado de aplicar un argumento en una clase parametrizada. Se muestran ambas en la Figura 6.16.

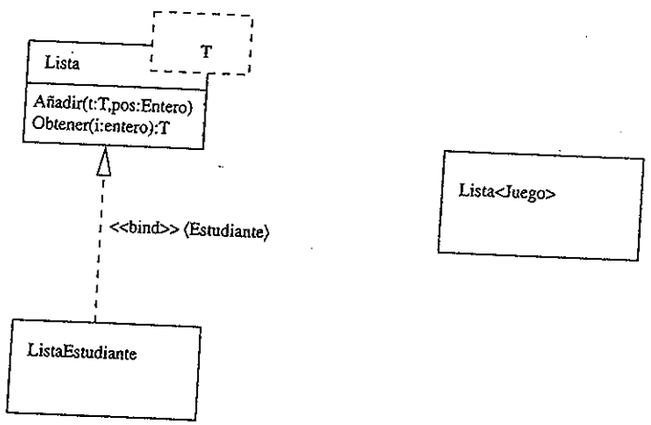


Figura 6.16 Una clase parametrizada y su utilización.

La especificación de UML sugiere una notación más prolija como `Lista<T → Juego>` en vez de `Lista<Juego>`. Donde sólo hay un parámetro, como en nuestro ejemplo, esto es una exageración. Sin embargo, puede haber cualquier número de parámetros; si se necesita utilizar más de uno puede ser que encuentre de mérito utilizar la notación más explícita.

Destacar que en la Figura 6.16 se muestra una dependencia, pero no otras. Ambas instanciaciones dependen de la clase parametrizada, así como de sus respectivos parámetros. Se omite, por ejemplo, la dependencia de `Lista<Juego>` sobre `Lista<T>` porque esta dependencia queda clara con el nombre de la clase.

P: ¿Por qué no se podría construir la clase `ListaEstudiante` a partir de la clase `Lista` utilizando herencia?

No todos los lenguajes soportan esta manera de construir clases directamente (a menudo conocido como *genericidad*); C++ lo soporta, pero Java, por ejemplo, no. Aunque el lenguaje que se vaya a utilizar en un proyecto no soporte clases parametrizadas, el diseño puede ser más legible utilizando la notación.

Pregunta de Discusión 54

Otra manera de obtener beneficios de la reutilización en el ejemplo de la Lista podría ser definir una clase `Lista` única, no parametrizada en función de una clase muy general como es `Object`, de la cual toda clase es subclase, y después confiar en la posibilidad de sustitución para permitir introducir cualquier objeto de cualquier clase en una `Lista`. Esto se hace a menudo en lenguajes que no soportan directamente las clases. ¿Qué ventajas e inconvenientes tiene esta aproximación, comparada con una clase parametrizada `Lista`? ¿De qué otra manera se podría alcanzar alguno de los beneficios de la reutilización?

6.4 Dependencia

Se han visto algunos ejemplos de una dependencia entre dos clases (y, en un caso, entre una clase y una clase parametrizada —¡recuerde que una clase parametrizada no es realmente una clase!—. Recordemos del Capítulo 1 que `A` depende de `B` si un cambio en `B` puede forzar un cambio en `A`. Se puede mostrar una dependencia entre dos elementos del modelo UML cualesquiera (y prácticamente cualquier cosa en un diagrama de UML es un elemento del modelo). En cada caso, la dependencia de `A` con `B` se indica con una *flecha de dependencia* punteada desde `A` hacia `B`, como en la Figura 6.16.

Hay que destacar la diferencia entre una dependencia entre dos clases y una asociación entre las clases. Una asociación entre dos clases representa el hecho en que los *objetos* de esas clases están asociadas. Una dependencia existe entre las clases en sí, no entre los objetos de esas clases.

En realidad, cuando hay una dependencia entre clases normalmente es posible, y preferible, ser más específicos sobre la naturaleza de la dependencia. Por ejemplo, una clase siempre depende de una clase de la que hereda, por lo que no es necesario mostrar explícitamente una dependencia en tal caso.

6.5 Componentes y paquetes

Las dependencias se utilizan de forma más habitual entre *paquetes*. Un paquete es una colección de elementos del modelo, y define un espacio de nombres para los elementos. Por ejemplo, un paquete podría ser una colección de clases relacionadas y las relaciones entre ellas, o una colección de objetos y las relaciones entre ellos. A menudo es conveniente poder empaquetar cosas como ésta, bien porque forman un componente o simplemente para dividir el trabajo en partes. Un paquete puede aparecer en un diagrama de clases (o, claro está, en cualquier otro tipo de diagrama), como un rectángulo con una "etiqueta" en la esquina superior izquierda. Si un diagrama muestra una dependencia o asociación enlazando un paquete con algo, esto significa que hay algún elemento en el paquete que tiene la dependencia o asociación. No es necesario especificar cuál. Se hablará de componentes, paquetes y subsistemas en más detalle en los Capítulos 13 y 14.

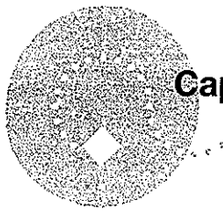
6.6 Visibilidad, protección

Las clases, a menudo, tienen atributos u operaciones que no están disponibles para todos los clientes de la clase. UML permite el uso de los símbolos +, # y − para distinguir entre los miembros de una clase públicos, protegidos y privados[3]. Los miembros públicos pueden ser accedidos por cualquier cliente de la clase. El significado exacto de protegido y privado depende del lenguaje. Los miembros privados normalmente sólo pueden ser accedidos por miembros de la misma clase. Los miembros protegidos, habitualmente, tienen un acceso más amplio que los miembros privados, pero no tanto como los miembros públicos.

RESUMEN

Este capítulo ha considerado algunas funciones avanzadas de los diagramas de clases de UML, y los mecanismos principales de UML de extensibilidad: estereotipos, restricciones y valores etiquetados. Se han tratado algunas formas de dar información extraordinaria sobre las asociaciones entre clases, considerando la agregación y composición, roles, navegabilidad, asociaciones calificadas, asociaciones derivadas y clases asociación. También se han cubierto las restricciones, que son una característica fundamental de UML, que pueden utilizarse para llevar a cabo una gran variedad de propósitos; aquí se ha tratado su utilización para especificar invariantes de clase y para almacenar las relaciones entre varias asociaciones. Después, se han considerado las interfaces que, otra vez, pueden aplicarse de forma más general. Se han mencionado las clases abstractas, y las conocidas clases parametrizadas, que no son clases, sino funciones que toman una o más clases como argumentos y devuelven clases como resultado. Finalmente, se ha considerado la dependencia y la visibilidad

[3] También está permitido ~ para la visibilidad del paquete, lo cual no se trata aquí.



Capítulo 7
Fundamento de los modelos de casos de uso

Los casos de uso documentan el comportamiento del sistema desde *el punto de vista del usuario*. En este caso, por "usuario" se entiende cualquier cosa que se desarrolla, ajena al sistema, que interactúa con el mismo. Un usuario podría ser una persona, otro sistema de información, un dispositivo hardware, etc. El modelado de los casos de uso ayuda con tres de los aspectos más difíciles del desarrollo:

- La captura de requisitos.
- La planificación de las iteraciones del desarrollo.
- La validación de los sistemas.

Los casos de uso fueron presentados, por primera vez por Ivar Jacobson a principios de los noventa, como un desarrollo a partir de la idea de *escenarios*. Los escenarios todavía existen en UML, y se tratarán más tarde en este capítulo.

Un *diagrama de casos de uso* es relativamente fácil de comprender de forma intuitiva, incluso sin conocer la notación. Esto es una ventaja importante, ya que el modelo de casos de uso se puede tratar de forma coherente con un cliente que no necesita estar familiarizado con UML. Para entender esto, *véase* la Figura 7.1, que muestra el diagrama de casos de uso del estudio del caso introductorio del Capítulo 3. Antes, examinemos en detalle los elementos de un modelo de casos de uso.

El diagrama muestra, no un único caso de uso, sino todos los casos de uso de un sistema dado. Un *caso de uso* individual, que aparece como un óvalo con un nombre, representa un tipo de tarea que tiene que soportar el sistema en el desarrollo. (El estándar de UML llama a esto "unidad coherente de funcionalidad": en este libro se prefiere el término "tarea"). Por supuesto, el diagrama de casos de uso muestra sólo una parte de la información que se necesita. Cada caso de uso también se describe en detalle, normalmente en texto. El diagrama de casos de uso puede verse como un resumen conciso de la información contenida en todas las descripciones de los casos de uso.

Un *actor*, que normalmente aparece con el símbolo de un muñeco, representa un tipo de usuario del sistema (donde, recuerde, por *usuario* se entiende cualquier cosa externa al sistema

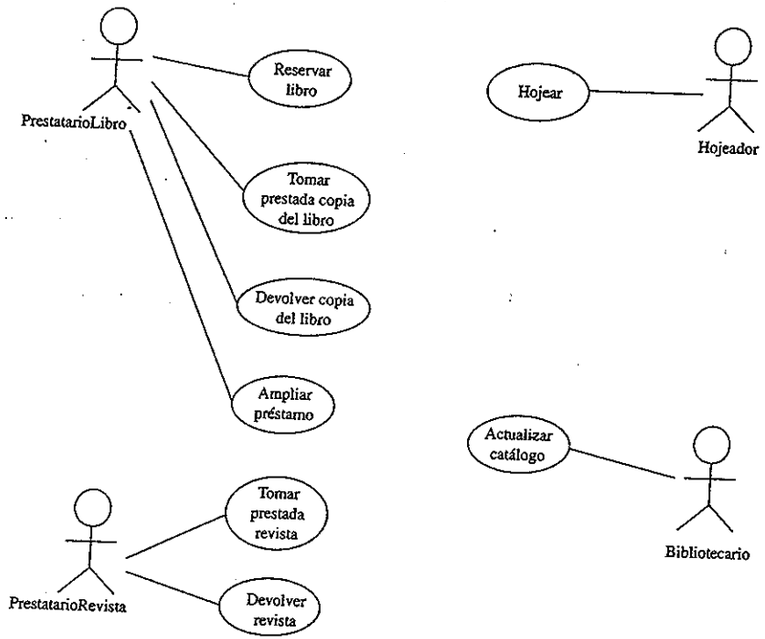


Figura 7.1 Diagrama de casos de uso para la biblioteca (diagrama 1).

que interactúa con él —no piense que el actor tiene que ser humano, confundido por la apariencia del icono).

Hay una línea que conecta un actor con un caso de uso si el actor (mejor dicho, alguien o algo que realice el rol representado por el actor) puede interactuar con el sistema para realizar parte de la tarea.

Un diagrama de casos de uso es bastante parecido a un diagrama de clases, en el sentido de que los iconos representan *conjuntos* de cosas y *posibles* interacciones, en vez de cosas individuales e interacciones definidas. En el Capítulo 5 ya se vio que la asociación simple es una `copia` de entre las clases `Copia` y `Libro`, mostrada en la Figura 7.2, que representa una

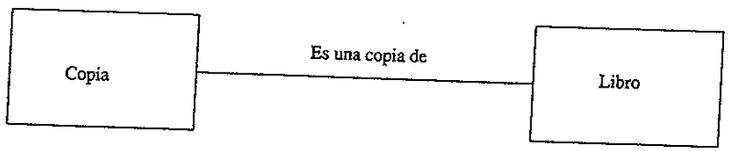


Figura 7.2 Asociación simple entre clases.

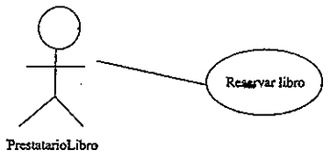


Figura 7.3 Comunicación sencilla entre un actor y un caso de uso.

relación entre el conjunto de objetos de la clase `Copia` y el conjunto de objetos de la clase `Libro`. Esto es, un objeto individual `miCopia` de la clase `Copia` y un objeto individual `eseLibro` de la clase `Libro` podrían o no estar relacionados. De la misma manera, la relación de comunicación muy sencilla mostrada en la Figura 7.3 indica que hay una relación entre el conjunto de `PrestatariosLibro`, y el conjunto de *escenarios* en el cual un `PrestatarioLibro` reserva un libro de la biblioteca. Un `PrestatarioLibro` en particular podría o no estar implicado en un escenario concreto de reserva de un libro. Un escenario es una instancia de un caso de uso, al igual que un objeto es una instancia de una clase. Se volverá a este punto más tarde. Por ahora, destacar dos cosas:

- Un actor, en un diagrama de casos de uso, representa un *rol* que alguien tiene que cumplir, en vez de representar a un individuo en particular. Por ejemplo, un bibliotecario puede ser también un prestatario de un libro: alguien que a veces realiza el papel de un bibliotecario, pero otras veces puede realizar el rol de prestatario de un libro.
- Una relación de comunicación entre un actor y un caso de uso no significa que *necesariamente* alguien de ese rol tenga que estar implicado en la ejecución de una tarea; simplemente significa que puede ser, dependiendo de las circunstancias.

Ahora, consideremos los actores y los casos de uso con mayor detalle.

7.1 Actores en detalle

Beneficiarios

Cada caso de uso tiene que representar una tarea, o unidad coherente de funcionalidad, que se le requiere al sistema que soporte. Normalmente, esto significa que el caso de uso tiene valor para al menos uno de los actores. Al actor para el que un caso de uso tiene valor se le llama *beneficiario* de ese caso de uso. Es importante identificar los beneficiarios de cada caso de uso, ya que si un caso de uso tiene valor para un determinado actor, ese actor permanecerá conectado al caso de uso a lo largo del desarrollo. (Quizá el nombre del actor puede cambiar, y quizá, incluso se pueda terminar con varios actores porque se reclasifican los roles, pero las personas o los sistemas externos representados por el actor siempre estarán representados de alguna manera). Sin embargo, si un actor no es beneficiario de un caso de uso, entonces la conexión entre el actor y el caso de uso es menos cierta. Puede haber otras maneras de proporcionar el mismo valor, es decir, satisfacer los mismos requisitos. Por esta razón, los desarrolladores tienen que conocer quién *necesita* un caso de uso y quién está implicado en él sin obtener ningún beneficio del mismo.

P: Piense en el caso de uso `Tomar prestada copia de libro` en el sistema de la biblioteca. No se ha mostrado al bibliotecario como actor conectado con ese caso de uso. ¿Cuáles serían las ventajas y desventajas de hacerlo?

Pregunta de Discusión 55

A veces se discute que sólo los actores que son beneficiarios deberían aparecer en un diagrama de casos de uso, porque la decisión de los actores que están implicados en la realización de un caso de uso es de diseño, no es parte del análisis de requisitos. ¿Qué piensa y por qué?

Hay circunstancias en las que ninguna de las personas que interactúan con un sistema para ejecutar una tarea, realmente, se benefician de él: la interacción del beneficiario con el sistema en esa tarea es indirecta. *Véase* la sección 7.4.1.

Identificar actores

Los usuarios potenciales de un sistema tienden a ser relativamente fáciles de identificar[1]. Para desarrollar un modelo de casos de uso se necesitan identificar los diferentes roles que estos humanos pueden desempeñar, recordando que una persona puede representar distintos roles en distintos momentos. Identificar los roles es algo parecido a considerar usuarios desde el punto de vista del sistema. Si Mary Smith y Joe Bloggs pudiesen ambos estar implicados en alguna parte del comportamiento del sistema (por ejemplo, reservar un libro), y "no provocan ninguna diferencia importante al comportamiento del sistema" si está interactuando con Mary Smith o Joe Bloggs, esto puede significar que tanto Mary Smith como Joe Bloggs son capaces de desempeñar el rol representado por un actor en particular conectado con el caso de uso que representa esa tarea.

Pregunta de Discusión 56

¿Qué diferencias en el comportamiento de un sistema piensa que podrían ser *importantes* en este contexto? ¿Podría precisarlo?

Hay algunas sutilezas a las que se volverá más tarde en este capítulo y en el siguiente, pero normalmente cualquier persona que interactúa con el sistema estará representado al menos por un actor en el modelo de casos de uso. Por supuesto, un usuario que desempeña varios roles diferentes es representado por varios actores, uno por cada rol.

Actores no humanos

La situación con los actores no humanos tiende a ser menos clara, principalmente debido a que es menos claro qué se debería considerar como un sistema externo o dispositivo. Por ejemplo, un teclado no se considera como un dispositivo que interactúa con el sistema, porque hay una persona manejando el teclado. Mostrar el teclado no sería útil; en su lugar, de forma natural uno se abstrae del hecho de que un operador (un humano) golpea un teclado cuya presión sobre las

[1] Si no lo son, es sospechoso: ¡posiblemente esté siendo embarcado en la tarea sorprendentemente común de construir un sistema que alguien quiere tener construido, pero que nadie realmente quiere utilizar!

teclas se envía al sistema, y muestra un actor que representa al humano interactuando directamente con el sistema. ¿Qué pasaría si se considerase un sistema que toma la entrada desde un lector de código de barras? ¿Y desde un reloj? ¿Y desde Internet? ¿Y desde un sistema informático distinto dentro de la misma compañía? ¿Qué pasaría si el sistema enviase la salida a un sistema externo dispositivo como éstos? ¿Dónde están los límites entre sistemas? Por ejemplo, supongamos que el sistema de la biblioteca permite a los usuarios solicitar préstamos entre bibliotecas, y que cuando se lleva a cabo tal petición, el sistema se pone en contacto con la otra biblioteca vía Internet. ¿Qué actores deberían aparecer en nuestro diagrama de casos de uso? ¿Internet? ¿El otro sistema de biblioteca? ¿Ninguno?

La solución es pragmática: se hace lo que parece que va a ser más útil, y distintas personas tienen visiones diferentes. Incluso si está claro lo que es un sistema externo o dispositivo, hay una pregunta sobre qué cosas deberían aparecer en un diagrama de casos de uso. Fowler y Scott [19] tratan las posibles vistas, que se pueden resumir diciendo que se pueden mostrar las interacciones con sistemas externos:

1. Siempre.
2. Cuando es el otro sistema o dispositivo el que inicia el contacto.
3. Cuando es el otro sistema o dispositivo el que toma valor del contacto.

De nuevo, hay personas que piensan que los actores deberían representar siempre humanos: por ejemplo, que se podría considerar que el bibliotecario de otra biblioteca apareciese como un actor, pero no el otro sistema de la biblioteca. El peligro con esta forma de ver las cosas, es que significa que puede que haya que conocer aspectos irrelevantes del funcionamiento de un sistema externo, para saber qué roles de las personas están implicados.

> ANOTACIÓN TÉCNICA DE UML
>
> En realidad, en UML se tiende a utilizar el rol para denotar lo que un objeto o actor hace en una colaboración específica: de este modo, un actor, técnicamente, desempeña un rol distinto en cada caso de uso y es un *conjunto coherente de roles*. Puede que se prefiera pensar en un actor como en una persona "que lleva un sombrero particular".

7.2 Casos de uso en detalle

Se dijo que un escenario es una instancia de un caso de uso, al igual que un objeto es una instancia de una clase. Como con los objetos y las clases, es más fácil describir qué es un escenario a describir qué es un caso de uso, ya que un caso de uso describe un conjunto de escenarios relacionados.

Un escenario es una posible interacción entre el sistema y algunas personas o sistemas/dispositivos (en sus diversos roles). La interacción puede describirse como una secuencia de mensajes. Por ejemplo, aquí hay dos escenarios:

- La prestataria de libros Mary Smith toma prestada la tercera copia de *Guerra y paz* de la biblioteca, cuando no tiene ningún otro libro en préstamo. El sistema se actualiza de acuerdo con esto.

- El prestatario de libros Joe Smiths intenta tomar prestada la primera copia de *Anna Karenina* de la biblioteca, pero es rechazado porque tiene ya seis libros en préstamo, que es el máximo permitido.

Pregunta de Discusión 57

¿Cuáles son los mensajes en cada caso? ¿"Mensaje" significa lo mismo en este contexto que en el Capítulo 2?

Ambos escenarios son posibles instancias del caso de uso `Tomar prestada copia de libro`. Hay que destacar que no sólo los interactores, sino también el resultado, son diferentes en los dos casos. Esto es común. Al igual que no todos los objetos de una misma clase envían los mismos mensajes durante su ciclo de vida, los escenarios en un mismo caso de uso pueden suponer un comportamiento diferente. Los escenarios en un caso de uso deberían tener en común que todos ellos intentan ejecutar, fundamentalmente, la misma tarea, incluso aunque el caso de uso incluya flujos alternativos o inusuales.

De este modo, un caso de uso engloba un conjunto de requisitos del sistema, posiblemente complejo, que surge durante la captura de requisitos iniciales y se refinará durante el desarrollo del sistema. Se necesita alguna manera de almacenar la información detallada que se tiene sobre lo que implica un caso de uso: ¿cuáles son los posibles escenarios, y qué determina cuál de ellos hay que aplicar en un conjunto de circunstancias dadas? Normalmente, esto se hace asociando una descripción textual con el caso de uso.

Una herramienta puede permitir al usuario hacer *click* sobre el icono ovalado que representa un caso de uso para ver el texto que proporciona la descripción detallada de lo que es ese caso de uso. Se puede utilizar también un *diagrama de actividad* de UML, o una descripción en algún lenguaje formal, y se puede asociar de forma similar con el caso de uso que describe.

Más tarde, se necesitará poder mostrar cómo la colección de clases y componentes diseñada, activa el sistema para realizar el caso de uso. Un caso de uso puede estar asociado con diagramas de *interacción*, que muestran cómo se realiza en un diseño de sistema en particular en dicho caso de uso, o algún subconjunto de escenarios de los que incluye.

7.3 Límite del sistema

Opcionalmente, puede haber una caja en un diagrama de casos de uso, alrededor de los casos de uso, etiquetada con el nombre del sistema. La caja representa el límite del sistema. Un ejemplo es el que aparece en la Figura 7.4.

Esto puede ser útil cuando se modela un sistema complejo, el cual se divide en diferentes subsistemas: se podría tener un diagrama de casos de uso por cada subsistema, en cuyo caso el límite del sistema puede ayudar a dejar claro qué subsistema se está modelando.

Sin embargo, cuando se dibuja un diagrama de casos de uso para representar un sistema sencillo, es común omitir la caja, y es lo que se hará en el resto de este libro.

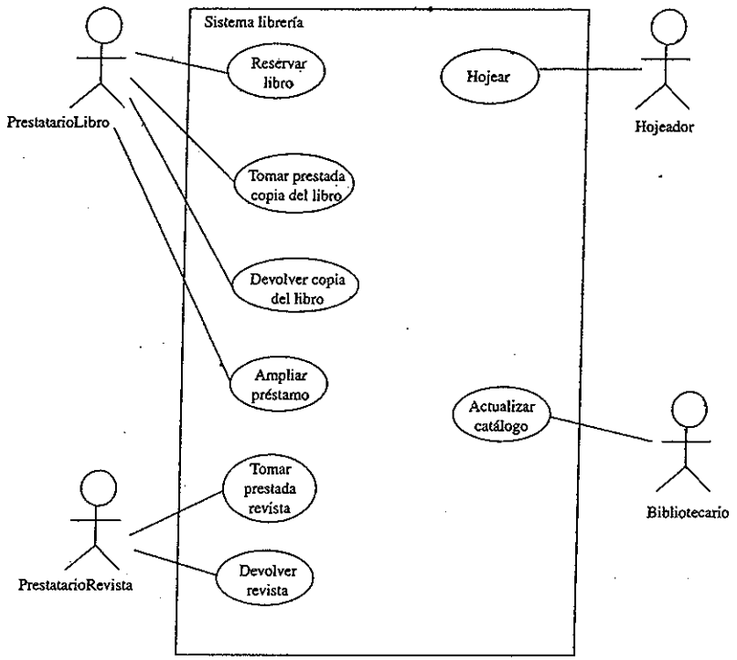


Figura 7.4 Diagrama de casos de uso para la biblioteca (diagrama 2).

7.4 Utilización de los casos de uso

7.4.1 Casos de uso para la captura de requisitos

Los casos de uso pueden ayudar con la captura de requisitos, proporcionando una forma estructurada de abordarlos:

1. Identificar los actores.
2. Para cada actor, averiguar:
   - lo que necesitan del sistema: es decir, qué casos de uso hay que tienen valor para ellos.
   - cualquier otra interacción que esperan tener con el sistema, esto es, en qué casos de uso podrían formar parte para el beneficio de otro.

Para propósitos de priorización de trabajo y planificación de iteraciones del desarrollo, también hay que saber *hasta qué punto* necesita una persona que se realice un caso de uso dado.

Puede ser útil hacer una lista de qué personas o sistemas reales pueden ejecutar el rol de cada actor, pero normalmente esto no se almacena en el diagrama de UML. En la discusión del sistema con los usuarios potenciales es necesario averiguar los actores que podrían estar en el diagrama de casos de uso, y si hay alguna función en la que parece que deberían participar y que no están incluidos actualmente.

Tal y como se destacó cuando se trataron los beneficiarios del sistema como actores, puede haber aspectos del comportamiento del sistema que no se presenten fácilmente como casos de uso para actores. Se conecta un actor con un caso de uso sólo cuando el actor participa en él, no, de manera más general, cuando un actor está involucrado de alguna forma con la existencia de un caso de uso. De modo que pueden existir casos en que una tarea sea esencial, pero no especialmente valiosa para cualquiera de los actores que participan en ella. El clásico ejemplo (mencionado en [19], por ejemplo) es cuando una compañía de servicios envía facturas trimestrales a cada uno de sus clientes. La compañía en conjunto se beneficia de esto; pero el único actor obvio en el caso de uso es el cliente, y posiblemente la compañía de correo, la cual no se beneficia del envío de las facturas. Debería presentar tales tareas cuando las descubra, aun cuando no se beneficie directamente alguno de los actores implicados. (Tenga cuidado, sin embargo, en asegurar que el caso de uso representa un requisito real).

7.4.2 Casos de uso a través del desarrollo

Planificación

Antes de realizar una estimación coherente y planificar el proceso para todo el proyecto, se necesita tener una lista con todos los casos de uso del sistema junto con:

- una buena idea de lo que significa cada uno.
- un entendimiento de quién quiere cada uno y cuánto.
- el conocimiento de qué casos de uso tienen más riesgo.
- un plan de cuánto tiempo llevaría implementar cada caso de uso.

Un punto que parece elemental, es que ¡no se debería planificar la entrega de un sistema en menos tiempo (esfuerzo total, de todos los desarrolladores implicados) que la suma de los tiempos de entrega que se ha planificado para los casos de uso![2] Llevar a cabo esta aritmética puede ser un antídoto útil para el optimismo de más. Cuando (y es cuando, en vez de si) se encuentra con que el sistema tiene que ser entregado completamente antes de lo previsto, tiene que encargarse de la negociación con el cliente sobre qué casos de uso no deberían proporcionarse en la primera versión. (Kent Beck lo llama *juego de planificación*, que engloba el hecho de que hay normas, por ejemplo, sobre cómo transcurre el tiempo, que no pueden romperse, ni siquiera para obligar al cliente). Este es uno de los puntos donde se diferencia el nivel de granularidad en que se describen los casos de uso. También es importante la identificación de la funcionalidad común a varios casos de uso y que puede, por lo tanto, reutilizarse: esto se tratará en el siguiente capítulo.

Una vez que se sabe qué casos de uso se ha contratado proporcionar, hay que decidir en qué orden se van a implementar, y qué casos de uso pertenecen a qué iteración del sistema.

[2] Teniendo en cuenta cualquier utilización que pueda predecir.

(Recuerde que una iteración puede ser de alto nivel, interna al proyecto, o puede ser de alto nivel o externa, es decir, puede producir la entrega de un sistema al cliente. El cliente está implicado en la decisión sobre qué casos de uso deben proporcionarse en las iteraciones externas, pero no en las decisiones sobre las iteraciones internas).

Aspectos políticos

¿Recuerda el 25% de los sistemas que nunca se entregan? No es porque los desarrolladores decidan tomarse vacaciones: es porque el proyecto se cancela. Es decir, alguien en alguna parte decide que no merece la pena continuar con él. ¿Cómo se puede impedir que pase esto en nuestro proyecto?

Si se han capturado los requisitos en función de los actores y los casos de uso, es probable que se tenga una buena idea de qué casos de uso —esto es, qué aspectos del comportamiento del sistema— son más importantes para qué personas. Por lo que, al igual que otras cosas, se quiere poder demostrar que el sistema *primero* hace algo de valor para las personas con mayor influencia. Tan pronto como sea posible, se quiere asegurar que todo aquél que tiene poder para hundir el sistema (¡y recuerde que puede incluir gente que no tiene poder formal en la organización!) no tiene una buena razón para hacerlo. Esto significa que han de ver que *si* el sistema se completa y entrega, obtendrán algo que quieren, y que perderán si cancelan el proyecto.

Comparando los casos de uso que son importantes para una persona dada, por supuesto, al igual que otras cosas, habría que implementar primero los de mayor prioridad.

Por supuesto, puede decidirse no *empezar* un proyecto si el análisis de los casos de uso no demuestra que proporcionará beneficio suficiente. ¡Esta es otra manera de reducir la probabilidad de que un proyecto determinado sea cancelado!

Aspectos técnicos

Otro criterio, que puede estar en conflicto con los anteriores, es que hay que entregar primero los casos de uso con mayor riesgo, para abordar los mayores riesgos cuando todavía se tiene la contingencia de abordarlos, y así uno no se queda atado en un diseño que no permita tratar los casos de uso más duros (los de mayor riesgo).

Pregunta de Discusión 58

En otras circunstancias, se pueden abordar primero las partes más *fáciles* de un problema. ¿Cuándo es ésta la aproximación correcta?

Hay que destacar que la manera en que los casos de uso son descritos, varía a lo largo del proceso de desarrollo. Para empezar, es importante identificar *qué* se debería alcanzar en cada caso de uso, no *cómo* debería alcanzarse. Más tarde se elegirá una implementación. Esto puede provocar el cambio de los actores; probablemente no los que quiere el caso de uso, sino los que intervienen en capacidad "de ayuda".

Validación del sistema

Cada caso de uso describe un requisito del sistema, por lo que un diseño correcto permite que se ejecute cada caso de uso; es decir, *realiza* cada caso de uso. Una técnica obvia, y muy útil, para

validar el diseño de un sistema es tomar de uno en uno los casos de uso y comprobar que el sistema permite ejecutar dicho caso de uso. A veces esto se llama recorrido (revisión) del caso de uso.

La misma técnica puede utilizarse para derivar pruebas del sistema: puede haber pruebas para cada caso de uso, y donde un caso de uso incluye familias de escenarios significativamente diferentes, debería incluirse un ejemplo de cada familia. Por ejemplo, el sistema de biblioteca tiene que probarse para ver tanto si permite a un usuario tomar prestado un libro, como si no permite a ningún usuario tomar prestados demasiados libros de una vez. La necesidad de estas pruebas se deriva directamente del modelo de casos de uso.

7.5 Posibles problemas con los casos de uso

Tal y como se ha visto, los casos de uso ayudan con algunos de los aspectos más difíciles del desarrollo de un sistema, a saber la captura de requisitos, planificación, gestión de iteraciones y planificación de pruebas. Sin embargo, algunos expertos, de entre los que destaca Meyer [35], están en contra de esto. Nosotros también tenemos nuestras reservas, pero más sobre las características del modelo de casos de uso descritas en el siguiente capítulo, que de la forma sencilla aquí tratada. Sin embargo, el modelo de casos de uso debería utilizarse con cuidado, ya que:

1. Hay peligro de construir un sistema que no sea orientado a objetos. El centrarse en los casos de uso puede ayudar a los desarrolladores a perder visión de la arquitectura del sistema y de la estructura de objetos estática, en las prisas para entregar de alguna manera los casos de uso que son necesarios en la iteración actual. Más tarde, si la funcionalidad de uno de los casos de uso ya ha sido desarrollada, puede ser difícil justificar el tiempo para modificar el diseño manteniendo su integridad con respecto a los casos de uso siguientes. Se puede terminar de nuevo donde se empieza, desarrollando un sistema de arriba a abajo, orientado a la función, imposible de mantener, inflexible. Este peligro puede reducirse con una gestión cuidadosa al principio de cada iteración. Si la iteración anterior deja el sistema en un estado que no es satisfactorio, debería ser *refactorizado* antes de añadir ninguna funcionalidad nueva, y el plan de la iteración debe permitirlo.
2. Hay peligro de tener un diseño de requisitos erróneo. Ya se ha visto un ejemplo de cómo sucede esto: la presunción de que un actor está implicado en un caso de uso del cual no obtiene ningún valor es, normalmente, una decisión de diseño, no una restricción. De forma más general, los requisitos por medio de los casos de uso pueden animar a los desarrolladores a pensar de forma operacional: los usuarios tienden a describir los casos de uso como una secuencia muy concreta de interacciones con el sistema que es una manera, no la única, de alcanzar su meta real. Por ejemplo, los usuarios piensan de forma natural en las cosas que tienen que hacerse en un determinado orden, quizá el orden en que se hacen en el presente, aunque otro orden podría ser igualmente apropiado. Es importante que los desarrolladores distingan entre requisitos y diseños candidatos.
3. Hay peligro de perder requisitos si se pone demasiada confianza en el proceso sugerido para encontrar los actores y después encontrar los casos de uso que necesita cada actor. Como se mencionó, no todos los requisitos surgen naturalmente de esta manera.

Este peligro puede reducirse haciendo el análisis de los casos de uso y el modelo de clases conceptual en paralelo.

Pregunta de Discusión 59

Una táctica podría ser desarrollar un modelo de casos de uso en el que sólo los actores que necesitan un caso de uso dado se comunican con él; los actores "auxiliares" no deberían aparecer. ¿Cuáles serían las ventajas y desventajas de este acercamiento?

PANEL 7.1 ¿Desarrollo dirigido por casos de uso?

"Dirigido por casos de uso" es una frase complicada, asociada a menudo con UML, presentada en [28] y utilizada por gran parte de la comunidad de UML. ¿Qué significa, y somos partidarios de ella?

En esencia, la idea es que los casos de uso son el aspecto más importante del proceso de diseño. No son desarrollados para hacer la captura de requisitos y después abandonarlos una vez que empieza el diseño: deberían utilizarse a lo largo de todo el proyecto, para seguir los cambios y definir las iteraciones, por ejemplo. Esta aproximación ayuda a mantener el centro donde debería estar, en los requisitos del usuario. Aquí hay un solapamiento con el *diseño centrado en el usuario* que se tratará brevemente en el Capítulo 19.

De manera más controvertida, [28] aboga por el examen de los casos de uso como método principal para encontrar objetos y clases, por ejemplo, así como método principal para encontrar componentes y formas de utilizarlos. Sin embargo, se han descrito algunos de los peligros de la sobre-confianza en los casos de uso. En particular, no creemos que el examen de los casos de uso sea, *por sí misma,* una buena forma de encontrar objetos y clases. En su lugar, pensamos que el desarrollo de un modelo de clases conceptual debería darse en paralelo con el desarrollo del modelo de casos de uso, y que cada uno se alimentara del otro. Algunas clases se descubrirán examinando los casos de uso. Algunos casos de uso se descubrirán examinando las clases. No consideramos útil clasificarnos como abogados del desarrollo "dirigido por casos de uso" o "dirigido por datos" o "dirigido por responsabilidad". Un buen desarrollo OO incluirá siempre aspectos de cada una de las tres aproximaciones.

RESUMEN

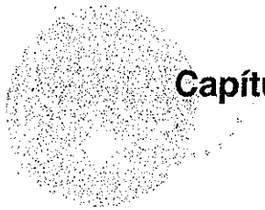
Este capítulo ha introducido los modelos de caso de uso simples y ha mostrado cómo se utilizan para especificar el comportamiento de un sistema de una forma independiente del diseño. Se ha tratado cómo identificar actores, casos de uso y relaciones de comunicación entre ellos, y cómo utilizar el modelo de casos de uso en el contexto de un proyecto de desarrollo.

En el siguiente capítulo se tratarán más características y usos de los diagramas de casos de uso. En los Capítulos 9 y 10 se mostrará cómo los diagramas de interacción se utilizan para demostrar cómo el diseño de un sistema realiza un caso de uso.

PREGUNTAS DE DISCUSIÓN

1. Piense en los actores del ejemplo de la biblioteca, y considere los conjuntos de personas que pueden ser representados por cada actor. Piense las intersecciones entre los conjuntos: por ejemplo, el conjunto de personas (si hay alguno) que a veces desempeña el rol de prestatario de libros y a veces el rol de bibliotecario. ¿Hay algún conjunto contenido en otro? ¿Cree que sería útil que el diagrama presentase las relaciones entre estos conjuntos de personas así como las relaciones entre los actores? ¿Por qué, o por qué no? ¿Si es así, cómo?
2. ¿Hay algunas relaciones interesantes entre algún caso de uso? Si es así, ¿cuáles son? Otra vez, ¿sería útil representarlos en el diagrama, y si es así, por qué y cómo?
3. Se ha dicho que un caso de uso, normalmente, debería tener valor para al menos uno de los actores. De algunos ejemplos, de sistemas que conozca, de casos de uso que tienen valor para más de uno de los actores implicados.

El Capítulo 8 muestra cómo representar algunas relaciones entre actores y entre casos de uso en UML: puede ser interesante comparar lo que permite UML con lo que piense que podría querer.



Capítulo 8
Más sobre modelos de casos de uso

En este capítulo se consideran más aspectos de los modelos de casos de uso y su uso en el desarrollo. Se tratará:

- Cómo y por qué se pueden mostrar las relaciones entre casos de uso.
- Cómo y por qué se pueden mostrar las relaciones entre actores.

El lector debe ser advertido de que cada una de estas funciones hace a un modelo de casos de uso más complejo (aunque posiblemente más pequeño). Al principio del capítulo anterior se declaró que una fuerza importante de los diagramas de caso de uso es su simplicidad. Aquí hay un conflicto obvio. Además, hay un gran desacuerdo en cómo deberían utilizarse exactamente las funciones que aquí se describen. Diferentes partes de la comunidad de UML tienen diferentes ideas y la especificación de UML2 en sí misma deja abiertas, deliberadamente, muchas decisiones. Se recomienda una aproximación a KISS: si duda, no utilice estas características.

Finalmente, se tratarán las circunstancias en las que un actor en el modelo de casos de uso debería ser modelado en el sistema mediante una clase, ya que, a menudo, esto causa confusión.

8.1 Relaciones entre casos de uso

Hay dos tipos de situaciones principales en las que se puede querer documentar una relación entre dos casos de uso. En el diagrama de casos de uso esto aparece como una flecha punteada de cabeza abierta entre dos elipses de casos de uso (es la misma flecha que se utiliza para mostrar otros tipos de dependencia). Los dos casos se distinguen dándoles diferentes *estereotipos*: al primero se le da el estereotipo `<<include>>`, al segundo se le da el estereotipo `<<extend>>`.

8.1.1 Casos de uso para la reutilización: <<include>>

El caso más vital es cuando se puede sacar factor común del comportamiento de dos o más casos de uso originales, o (mejor todavía) cuando se descubre que se puede implementar parte de uno de los casos de uso utilizando un componente.

Por ejemplo, destacar que las descripciones de los casos de uso `Tomar prestada copia de libro` y `Ampliar préstamo` mencionan la necesidad de comprobar si existe una reserva sobre el libro. Si la hay, entonces ni el préstamo puede ser ampliado ni el libro puede ser prestado. Se podría decidir mostrar esta característica común de los dos casos de uso en el diagrama de casos de uso, como aparece en la Figura 8.1. Hay que destacar que la flecha va desde el caso de uso "usuario" hacia el caso de uso "usado", y se etiqueta (con el estereotipo) `<<include>>`, para representar que *el caso de uso origen incluye el caso de uso destino*. De forma más precisa, los escenarios que son instancias del caso de uso origen, contienen subescenarios que son instancias del caso de uso destino. Si el caso de uso destino cambia para contener escenarios diferentes, el caso de uso origen se verá afectado, porque todos sus escenarios también cambiarán; pero el caso de uso destino no depende del caso de uso fuente.

Por supuesto, hay una descomposición que se corresponde con las descripciones detalladas del caso de uso. Las nuevas descripciones podrían leerse:

- **Tomar prestada copia de libro** Un `PrestatarioLibro` presenta un libro. El sistema comprueba que el prestatario potencial es socio de la biblioteca, y que él o ella todavía no tiene el máximo número de libros en préstamo permitidos. Este máximo es seis, a menos que el socio sea de la plantilla, en cuyo caso es 12. Si ambas comprobaciones tienen éxito, el sistema comprueba si hay una reserva sobre el libro (caso de uso `Comprobar para reserva`); en otro caso, el sistema rechaza el prestar el libro. Si el libro está reservado, el sistema rechaza prestarlo. En otro caso, almacena que ese socio de la biblioteca tiene esta copia del libro en préstamo y apunta la fecha de devolución pegándola en el libro.
- **Ampliar préstamo** Un `PrestatarioLibro` solicita (bien en persona, bien por teléfono) ampliar el préstamo de un libro. El sistema comprueba si hay una reserva sobre el libro (caso de uso `Comprobar para reserva`). Si es así, el sistema rechaza ampliar el préstamo. En otro caso, almacena que el préstamo del libro por este socio de la biblioteca

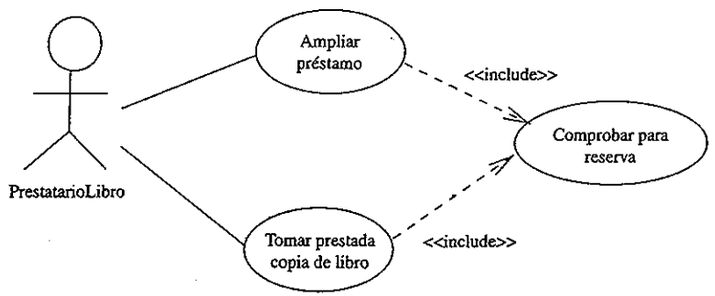


Figura 8.1. Reutilización de casos de uso: `<<include>>`.

ha sido ampliado, actualizando los registros de ambos. Se apunta la nueva fecha de devolución pegándola en el libro. Si el prestatario está en persona, esto se hace pegándola de nuevo; de forma alternativa, si la ampliación se hace por teléfono, se le solicita al prestatario que altere la fecha de devolución.

- **Comprobar para reserva** Dada la copia de un libro, el sistema busca en la lista de reservas pendientes alguna reserva sobre ese libro. Si encuentra alguna, compara el número de reservas, n, con el número de copias del libro que se sabe que hay en la estantería de libros reservados, m. Si $n > m$ entonces el sistema indica que esa copia está reservada, en otro caso indica que no.

Pregunta de Discusión 60

¿Es esta la mejor factorización de funcionalidad? ¿Cómo compararía esta factorización con otras? ¿Puede hacerlo basado sólo en un modelo de casos de uso?

La documentación compartida o la funcionalidad reutilizada, como ésta, en un diagrama de casos de uso tiene varias ventajas:

- Es una manera conveniente de almacenar la decisión de que se va a utilizar un componente, o de evitar almacenar la misma información en más de una descripción detallada de un caso de uso.
- Sacar factores de partes de la descripción del caso de uso puede hacer que las descripciones de los casos de uso sean más cortas y fáciles de comprender, con tal que los casos de uso incluidos sean unidades de funcionalidad coherentes por sí mismas.
- Identificar la funcionalidad común entre casos de uso en una etapa temprana puede ser una manera de descubrir una posible reutilización de un componente que puede implementar la funcionalidad compartida. Tal y como se trató en el Capítulo 7, el diagrama de casos de uso es una entrada importante en el proceso de planificación. Por lo tanto, es útil saber dónde se comparte la funcionalidad entre casos de uso; esto evita presupuestar el tiempo para la funcionalidad dos veces, que es lo que se haría si no se descubriera la funcionalidad compartida hasta más tarde. Sin embargo, destacar que no se desarrollaría un plan detallado basándonos sólo en el modelo de casos de uso, por lo que `<<include>>` no es la única manera de alcanzar planes precisos.

Sin embargo, hay también ciertos peligros asociados con la identificación y documentación de la reutilización de esta manera, especialmente si los casos de uso incluidos representan pequeñas partes de funcionalidad.

- Hay un serio peligro de que, en la búsqueda de la reutilización en el modelo de casos de uso orientados a la funcionalidad, se vuelva atrás a un estilo de diseño de descomposición funcional de arriba a abajo: exactamente igual al estilo inflexible que se supone que la orientación a objetos ayuda a evitar.
- La incorporación en un modelo de casos de uso de `<<include>>` es difícil de leer por alguien que no está acostumbrado a UML, por lo que empieza a perder su atractivo como documento visible al cliente. Además, cuanto más complejo es el diagrama de casos de uso, es más difícil mantenerlo actualizado, especialmente si permite incorporar información de diseño así como sobre los requisitos.

Para abordar el primer punto, es aconsejable desarrollar el modelo de clases del nivel conceptual en paralelo con el modelo de los casos de uso, y utilizar técnicas como las tarjetas CRC, para asegurar que cualquier reutilización que aparece en el diagrama de casos de uso le da sentido conceptual también al nivel de objetos. Por supuesto, a la inversa, el uso de tarjetas CRC y de técnicas similares puede ayudar a identificar funcionalidad compartida, que debe aparecer entonces en el modelo de casos de uso.

¿Cuánta funcionalidad se necesita compartir, antes de que merezca la pena documentar la sección compartida como un caso de uso separado? Como guía aproximada, probablemente, se debería separar la funcionalidad compartida sólo si el tiempo que lleva desarrollarlo es significativo en términos de planificación, lo que normalmente significa que el tiempo que se tiene que reservar para hacerlo es mayor que la unidad mínima de tiempo en que está formulado el plan, quizá, un "día de ingeniería ideal". En el caso que se ilustra, es improbable que sea verdad, por lo que *no* se querría, en la práctica, separar este caso de uso.

8.1.2 Componentes y casos de uso

Los componentes y los casos de uso interactúan (al menos) de dos maneras. Primero, consideremos el impacto de *utilizar* un componente en el modelo de casos de uso y, después, cómo un modelo de casos de uso puede ayudar a especificar un componente.

Si se es muy serio a la hora de hacer el diseño basado en componentes, es fundamental pensar en utilizar componentes lo más pronto posible, en realidad, desde el principio del proyecto. Hay varias razones para esto.

Primero, se necesita adaptar la manera en que se describen los casos de uso para encajar los componentes disponibles; se puede necesitar también negociar los cambios en los requisitos para hacer un buen uso de los componentes disponibles. Esto puede parecer radical: pero piense en el caso paralelo en, por ejemplo, arquitectura. No es probable que se le pregunte exactamente qué medida y forma quiere que tengan sus puertas. Es más probable, que se le pida elegir de entre una gama de tamaños y formas, cada uno con su propio coste; si es completamente posible especificar que quiere una puerta redonda, verde con un tirador de latón, puede esperar pagar un precio mucho más alto por ella, y esperar más tiempo, que si quiere algo estándar. Parece razonable decir que si la industria del software quiere hacer el cambio a la ingeniería basada en componentes, la aproximación a la flexibilidad de requisitos tendrá que cambiar de forma similar.

Pregunta de Discusión 61

¿Está de acuerdo? ¿O piensa, por ejemplo, que la flexibilidad inherente del software será suficiente para mitigar esto?

En segundo lugar, se quiere saber, antes de gastar tiempo y esfuerzo calculando cómo construirlo nosotros mismos, si se puede implementar algún requisito utilizando un componente.

Aquí uno se encuentra por primera vez con una pregunta que se repetirá: cuando se utiliza un componente que no se necesita desarrollar, ¿aparece en los diagramas de diseño? ¿o simplemente se trata como si fuese parte del lenguaje de programación, y se utiliza? Cualquiera es posible; cuál es mejor depende del componente que sea (y algunos lo amplían al gusto). Por ejemplo, supongamos que se utiliza una clase `ColecciónOrdenada` de una librería de clases de conjuntos, para

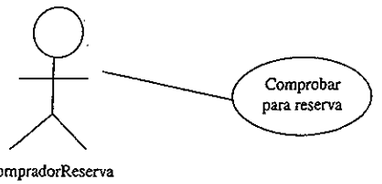


Figura 8.2 Un diagrama de casos de uso que describe un componente.

ordenar los artículos en una lista de libros retrasados. Probablemente no tiene valor mostrar `Ordenar cosas` como un caso de uso `<<include>>`; esta parte de funcionalidad es demasiado pequeña. Sin embargo, si se utiliza un componente más complejo probablemente se quiera mostrarlo.

Naturalmente, si se planifica desarrollar un componente reutilizable que incluya parte, pero no toda la funcionalidad de un caso de uso dado, también tendrá sentido describir el componente propuesto claramente con su(s) propio(s) caso(s) de uso. La principal diferencia entre un caso de uso para un componente y un caso de uso para todo el sistema, es que los actores que interactúan con un componente podrían ser objetos externos al mismo, en vez de humanos o sistemas externos o dispositivos. Un objeto externo al componente, se parece a un sistema externo o dispositivo desde el punto de vista del componente; sólo se ha cambiado la perspectiva. Por ejemplo, si `Comprobar para reserva` es un componente que debería documentarse de forma separada, su documentación podría incluir la descripción detallada dada antes y el diagrama de casos de uso, muy sencillo, que aparece en la Figura 8.2.

Tal y como se ha mencionado anteriormente, este pequeño ejemplo probablemente es demasiado sencillo para tratarlo de esta manera.

Resumen: utilización de `<<include>>`

Considere la utilización de una relación `<<include>>` entre casos de uso:

- Para mostrar cómo el sistema puede utilizar un componente que ya existe.
- Para mostrar la funcionalidad común entre casos de uso.
- Para documentar el hecho de que el proyecto ha desarrollado un nuevo componente reutilizable.

Un proyecto probablemente empezará desarrollando un diagrama de casos de uso sencillo que no utiliza las funciones descritas en este capítulo; un diagrama que utiliza `<<include>>` probablemente está mejor visto como refinamiento de tal diagrama, sobre el que se han tomado algunas decisiones de diseño.

8.1.3 Separación del comportamiento variable: `<<extend>>`

Si un caso de uso incorpora dos o más escenarios con diferencias significativas —esto es, pueden ocurrir varias cosas distintas dependiendo de las circunstancias— se puede decidir que sería más claro mostrarlos como un caso principal y uno o más casos secundarios. Hacer esto es un

problema de juicio, ya que siempre se pueden mostrar casos variables en un caso de uso. Por ejemplo, se podrían separar `Tomar prestada copia de libro` en el caso normal en el que al usuario se le permite tomar prestado el libro, y en el caso inusual en el que al usuario no se le permite tomar prestado el libro porque él o ella ya tiene prestados el máximo número de artículos.

Se utiliza la flecha `<<extend>>` desde el caso menos central al caso central, tal y como se muestra en la Figura 8.3. Tenga cuidado: la flecha va desde el caso excepcional al caso normal, ¡la mayoría de la gente piensa en ella como "la contraria" a la flecha `<<include>>`!

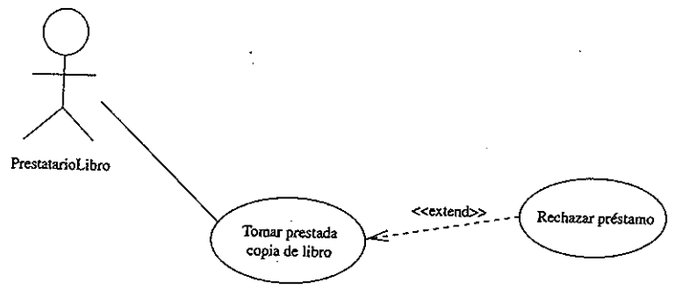


Figura 8.3 `<<extend>>`.

Otra vez hay una descomposición correspondiente de la descripción del caso de uso. En la nueva versión de la descripción del caso normal, se debe mostrar:

- la condición bajo la cual se aplica el caso excepcional.
- el punto en el que la condición se prueba y el comportamiento puede divergir: este es el *punto de extensión.*

UML permite (pero no requiere) mostrar la condición con la flecha de extensión, y almacenar el punto de extensión en la elipse del caso de uso central, tal y como aparece en la Figura 8.4. Probablemente es más útil si se está utilizando un lenguaje de descripción formal o semiformal para describir los casos de uso.

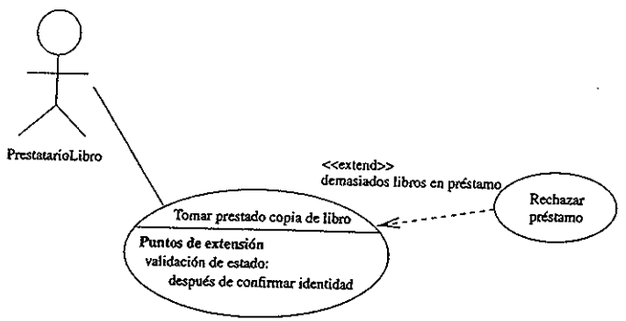


Figura 8.4 `<<extend>>` con punto de extensión.

8.2 Generalizaciones

Dos actores, o dos casos de uso, pueden estar relacionados por medio de la generalización, al igual que dos clases[1]. Por ejemplo, en el ejemplo de la biblioteca, toda persona `PrestatarioRevista` *es* `PrestatarioLibro`, porque las personas con derecho a tomar prestada una revista también pueden tomar prestados libros. Se puede decidir almacenar una relación de generalización entre los actores correspondientes utilizando la misma notación que se utiliza para las clases, tal y como se muestra en la Figura 8.5.

Cuando los casos de uso están relacionados a través de una generalización, la idea es mostrar una tarea y una versión especializada de la misma. Otra vez, se utiliza la flecha de generalización estándar, que va desde el caso de uso especializado al caso de uso más general. Por ejemplo, si se tiene un caso de uso `Reservar libro`, se podría tener una especialización de la misma llamada `Reserva de libro por teléfono`. Esto podría ser útil si el sistema de la biblioteca necesita comportarse de manera diferente para una reserva por teléfono; por ejemplo, si implica que haya que introducir de forma manual el número de la tarjeta de la biblioteca del usuario debido a que la tarjeta no puede ser escaneada. Esto es muy parecido a `<<extend>>` y es discutible que UML deba tener ambos. Una regla básica es que si se quiere describir un comportamiento extra que muchas veces hay que añadir dependiendo de las condiciones "de tiempo de ejecución", probablemente se quiera `<<extend>>`; mientras que si lo que se quiere es una etiqueta para una versión especializada de una tarea completa, probablemente se quiera generalización.

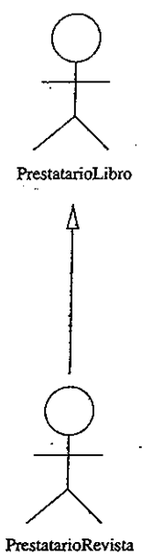


Figura 8.5 Generalización entre actores.

[1] Clases, actores y casos de uso son todos clasificadores en UML y cualquier clasificador puede ser generalizado.

8.3 Actores y clases

Es muy común que un sistema interactúe con un (instancia de) actor y que tenga también un objeto interno representando la instancia de actor. Hay dos tipos de situaciones fundamentales en las que puede pasar lo siguiente:

1. El sistema puede necesitar almacenar datos sobre un actor, normalmente una persona en un determinado rol. Por ejemplo, el sistema de biblioteca necesita saber qué personas tienen derecho a tomar prestados libros, y cuántos libros tiene cada uno de ellos en préstamo, para ejecutar el caso de uso `Tomar prestada copia de libro`. En sistemas orientados a objetos, suele significar que hay un conjunto de personas reales que pueden tomar el rol descrito a través de un actor del sistema dado, y también un conjunto de objetos del sistema, uno por persona, que almacena información sobre las personas en ese rol.
2. Una situación sutilmente diferente es cuando el sistema *envuelve* un sistema externo para proporcionar una forma manejable para que partes del sistema puedan acceder al sistema externo y viceversa. Por ejemplo, un monitor de procesamiento de transacciones externo podría ser accedido por medio de mensajes enviados a un objeto TPMonitor interno que, por turnos, invoca la funcionalidad real del sistema externo. O un programa de interfaz de usuario separado podría representarse dentro del sistema utilizando un objeto UI que en realidad media entre el programa UI externo y el sistema principal, pasando mensajes en ambos sentidos.

Las funciones de estos dos casos pueden combinarse: por ejemplo, una de las maneras más fáciles de proporcionar una interfaz de usuario sencilla a un sistema, que puede mantener la necesidad del sistema de responder a diferentes usuarios de distintas maneras, es hacer que los casos de uso, que pueden ser iniciados por Jane Bloggs, estén disponibles como métodos de un objeto del sistema que representa a Jane Bloggs. Un ejemplo de esto es la manera en que se implementa el caso de uso `Tomar prestada copia de un libro` en el Capítulo 3.

Sin embargo, es fácil confundirse e importante recordar dónde está el límite del sistema. La diferencia principal es que se pueden programar los objetos del sistema para que hagan lo que uno quiera, ¡pero no los actores del sistema!

Pregunta de Discusión 62

¿Cuáles son las ventajas y las desventajas de representar las instancias de actor como objetos del sistema?

8.3.1 Notación: actores como clases

Sólo para hacer las cosas aún más confusas, se puede escuchar a gente decir que los actores son clases, con el estereotipo `<<actor>>`. Esto es cierto a nivel de notación: un actor puede estar representado por un icono de clase con el estereotipo `<<actor>>` en vez de un muñeco, tal y como se muestra en la Figura 8.6. Sin embargo, como se adelantó anteriormente, en realidad los actores y las clases son ambos clasificadores, en vez de ser uno cualquiera, un tipo del otro.

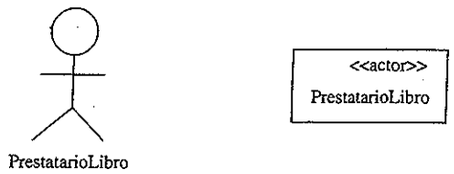


Figura 8.6. Estos dos símbolos significan lo mismo.

RESUMEN

En este capítulo se ha mostrado cómo la relación `<<include>>` puede almacenar la funcionalidad común a varios casos de uso, y cómo la relación `<<extend>>` puede almacenar lo que ocurre en casos inusuales. Se ha tratado la generalización entre actores y entre casos de uso, y la relación entre actores y clases. En los siguientes dos capítulos se demostrará cómo los diagramas de interacción pueden almacenar la interacción de los objetos para realizar los casos de uso.

PREGUNTA DE DISCUSIÓN

1. Se ha sugerido que la versión de un diagrama de casos de uso, utilizando las características aquí descritas, debería utilizarse en conjunción con una forma más sencilla de la descrita en el capítulo anterior. ¿Cómo cree que una herramienta CASE podría soportar esto de forma sensata?