

Two sets of letters are shown, each consisting of two rows. The first row contains the letters 'T', 'W', 'I', 'L', 'S', 'O', 'T', 'A' and the second row contains 'G', 'O', 'T', 'A'. The letters are rendered in a bold, black, sans-serif font.

T W I L S O T A

G O T A

Diagramas de clase: fundamentos

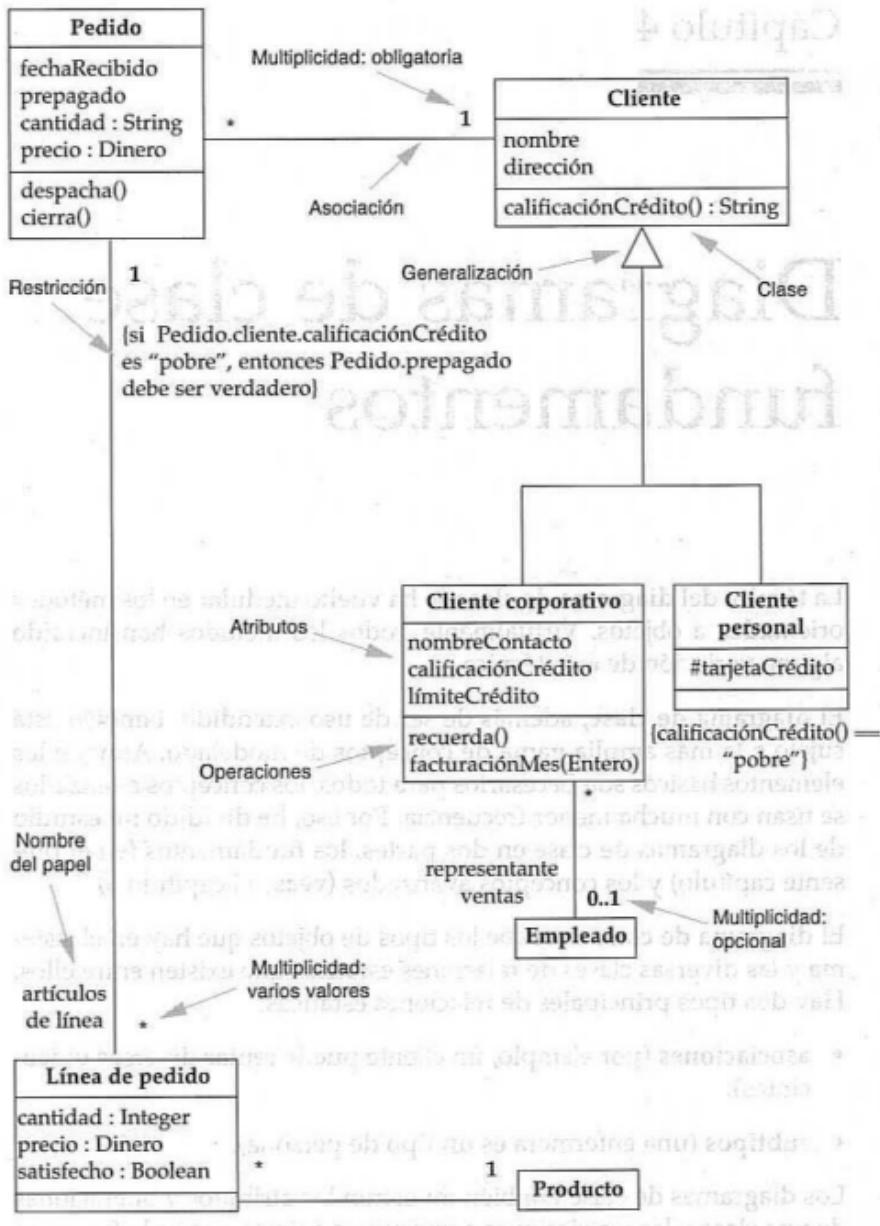
La técnica del **diagrama de clase** se ha vuelto medular en los métodos orientados a objetos. Virtualmente, todos los métodos han incluido alguna variación de esta técnica.

El **diagrama de clase**, además de ser de uso extendido, también está sujeto a la más amplia gama de conceptos de modelado. Aunque los elementos básicos son necesarios para todos, los conceptos avanzados se usan con mucha menor frecuencia. Por eso, he dividido mi estudio de los diagramas de clase en dos partes; los fundamentos (en el presente capítulo) y los conceptos avanzados (véase el capítulo 5).

El **diagrama de clase** describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos. Hay dos tipos principales de relaciones estáticas:

- **asociaciones** (por ejemplo, un cliente puede rentar diversas video-cintas).
- **subtipos** (una enfermera es un tipo de persona).

Los diagramas de clase también muestran los atributos y operaciones de una clase y las restricciones a que se ven sujetos, según la forma en que se conecten los objetos.



Los diversos métodos OO utilizan terminologías diferentes (y con frecuencia antagónicas) para estos conceptos. Se trata de algo sumamente frustrante pero inevitable, dado que los lenguajes OO son tan desconsiderados como los métodos. Es en esta área que el UML aportará algunos de sus mayores beneficios, al simplificar estos diferentes diagramas.

La figura 4-1 muestra un diagrama de clase típico.

Perspectivas

Antes de empezar a describir los diagramas de clase, quisiera señalar una importante sutileza sobre el modo en que se usan. Tal sutileza generalmente no está documentada, pero tiene sus repercusiones en el modo en que debe interpretarse un diagrama, ya que se refiere a lo que se va a describir con un modelo.

Siguiendo la recomendación de Steve Cook y John Daniels (1994), considero que hay tres perspectivas que se pueden manejar al dibujar diagramas de clase (o, de hecho, cualquier modelo, aunque esta división se advierte de modo especial en relación con los diagramas de clase).

- **Conceptual.** Si se adopta la perspectiva conceptual, se dibuja un diagrama que represente los conceptos del dominio que se está estudiando. Estos conceptos se relacionan de manera natural con las clases que los implementan, pero con frecuencia no hay una correlación directa. De hecho, los modelos conceptuales se deben dibujar sin importar (o casi) el software con que se implementarán, por lo cual se pueden considerar como independientes del lenguaje. (Cook y Daniels llaman perspectiva esencial a esto; por mi parte, empleo el término "conceptual", pues se ha usado durante mucho tiempo.)
- **Especificación.** Ahora estamos viendo el software, pero lo que observamos son las interfaces del software, no su implementación. Por tanto, en realidad vemos los tipos, no las clases. El desarrollo orientado a objetos pone un gran énfasis en la diferencia entre in-

terfaz e implementación, pero esto con frecuencia se pasa por alto en la práctica, ya que el concepto de clase en un lenguaje OO combina tanto la interfaz como la implementación. Así, a menudo se hace referencia a las interfaces como tipos y a la implementación de esas interfaces como clases. Influidos por este manejo del lenguaje, la mayor parte de los métodos han seguido este camino. Esto está cambiando (Java y CORBA tendrán aquí cierta influencia), pero no con suficiente rapidez. Un tipo representa una interfaz que puede tener muchas implementaciones distintas debido, por ejemplo, al ambiente de implementación, características de desempeño y proveedor. La distinción puede ser muy importante en diversas técnicas de diseño basadas en la delegación; véase el estudio de este tema en Gamma *et al.* (1994).

- **Implementación.** Dentro de esta concepción, realmente tenemos clases y exponemos por completo la implementación. Ésta es, probablemente, la perspectiva más empleada, pero en muchos sentidos es mejor adoptar la perspectiva de especificación.

La comprensión de la perspectiva es crucial tanto para dibujar como para leer los diagramas de clase. Desafortunadamente, las divisiones entre las perspectivas no son tajantes y la mayoría de los modeladores no se preocupa por definirlas con claridad cuando las dibujan. A medida que ahonde en mi exposición de los diagramas de clase, enfatizaré cómo cada elemento de la técnica depende en gran medida de la perspectiva.

Cuando usted dibuje un diagrama, hágalo desde el punto de vista de una sola perspectiva clara. Cuando lea un diagrama, asegúrese de saber desde qué perspectiva se dibujó. Dicho conocimiento es esencial si se quiere interpretar correctamente el diagrama.

La perspectiva no es parte del UML formal, pero considero que es extremadamente valiosa al modelar y revisar los modelos. El UML se puede utilizar con las tres perspectivas. Mediante el etiquetado de las clases con un estereotipo (véase la página 86), se puede dar una indicación clara de la perspectiva. Las clases se marcan con <<clase

Tabla 4-1: Terminología de diagramas de clase

UML	Clase	Asociación	Generalización	Agregación
Booch	Clase	Usa	Hereda	Contiene
Coad	Clase y objeto	Conexión de instancias	Espec-gen	Parte-todo
Jacobson	Objeto	Asociación por reconocimiento	Hereda	Consiste en
Odell	Tipo de objeto	Relación	Subtipo	Composición
Rumbaugh	Clase	Asociación	Generalización	Agregación
Shlaer/Mellor	Objeto	Relación	Subtipo	N/A

de implementación>> para mostrar la perspectiva de implementación y con <<tipo>> para el caso de la perspectiva de especificación y la conceptual. La mayor parte de los usuarios de los métodos OO adopta una perspectiva de implementación, lo cual es de lamentar, porque las otras perspectivas con frecuencia son más útiles.

La tabla 4-1 lista cuatro términos del UML que aparecen en la figura 4-1 y sus términos correspondientes en otras metodologías bien establecidas.

Asociaciones

La figura 4-1 muestra un modelo de clases simple que no sorprenderá a nadie que haya trabajado con un proceso de pedidos. Describiré sus partes y hablaré sobre las maneras de interpretarlas, desde las distintas perspectivas.

Comenzaré con las asociaciones. Las **asociaciones** representan relaciones entre instancias de clases (una persona trabaja para una empresa; una empresa tiene cierta cantidad de oficinas).

Desde la perspectiva **conceptual**, las asociaciones representan relaciones conceptuales entre clases. El diagrama indica que un Pedido debe venir de un solo Cliente y que un Cliente puede hacer varios Pedidos en un periodo de tiempo. Cada uno de estos Pedidos tiene varias instancias de Línea de pedido, cada una de las cuales se refiere a un solo Producto.

Cada asociación tiene dos **papeles**; cada papel es una dirección en la asociación. De este modo, la asociación entre Cliente y Pedido contiene dos papeles: uno del Cliente al Pedido; el segundo del Pedido al Cliente.

Se puede nombrar explícitamente un papel mediante una etiqueta. En este caso, el papel en sentido Pedido a Línea de orden se llama Artículos de línea. Si no hay etiqueta, el papel se puede nombrar de acuerdo con la etiqueta de la clase; de esta forma, el papel de Pedido a Cliente se llamará Cliente (en este libro, me refiero a la clase de donde parte el papel como **origen** y a la clase a donde va el papel como **destino**. Esto significa que hay un papel de Cliente cuyo origen es Pedido y cuyo destino es Cliente).

Un papel tiene también una **multiplicidad**, la cual es una indicación de la cantidad de objetos que participarán en la relación dada. En la figura 4-1, la * entre Cliente y Pedido indica que el primero puede tener muchas Órdenes asociadas a él; el 1 indica que un Pedido viene de un solo Cliente.

En general, la multiplicidad indica los límites inferior y superior de los objetos participantes. El * representa de hecho el intervalo $0...infinito$: el Cliente no necesita haber colocado un Pedido, y no hay un tope superior (por lo menos en teoría) para la cantidad de pedidos que puede colocar. El 1 equivale a 1..1: cada Pedido debe haber sido solicitado por un solo Cliente.

En la práctica, las multiplicidades más comunes son 1, *, y 0..1 (se puede no tener ninguno o bien tener uno). Para una multiplicidad más general, se puede tener un solo número (por ejemplo, 11 jugadores en un equipo de fútbol), un intervalo (por ejemplo, 2..4 para los participantes de un juego de canasta) o combinaciones discretas de números e intervalos (por ejemplo, 2, 4 para las puertas de un automóvil).

La figura 4-2 muestra las notaciones de cardinalidad del UML y de los principales métodos pre-UML.

Dentro de la perspectiva de la **especificación**, las asociaciones representan responsabilidades.

La figura 4-1 significa que hay uno o más métodos asociados con Cliente que me proporcionarán los pedidos que ha colocado un Cliente dado. Del mismo modo, existen métodos en Pedido que me permitirán saber qué Cliente colocó tal Pedido y cuales son los Artículos de lÍnea que contiene un Pedido.

Si existen convenciones estándar para nombrar **métodos de consulta**, probablemente sus nombres se puedan deducir del diagrama. Por ejemplo, puedo tener una convención que diga que las relaciones de **un solo valor** se implementan con un método que devuelve el objeto relacionado y que las relaciones de varios valores se implementan mediante una enumeración (iterador) en un **conjunto** de objetos relacionados.

Al trabajar con una convención de nombres como ésta en Java, por ejemplo, puedo deducir la siguiente interfaz para una clase de Pedido:

```
class Pedido {  
    public Cliente cliente() ;  
    // Enumeración de líneas de Pedido  
    public Enumeration lineasPedido() ;
```

← Lectura de izquierda a derecha →

	Una A siempre se asocia con una B	Una A siempre se asocia con una o más B	Una A siempre se asocia con ninguna o con una B	Una A siempre se asocia con ninguna, con una o con más B
<i>Booch</i> (1 ^a ed.)				
<i>Booch</i> (2 ^a ed.)*				
<i>Coad</i>				
<i>Jacobson**</i>				
<i>Martin/ Odell</i>				
<i>Shlaer/ Mellor</i>				
<i>Rumbaugh</i>				
<i>Unified</i>				

* puede ser unidireccional

** unidireccional

Figura 4-2: Notaciones de cardinalidad

Las convenciones de programación, por supuesto, variarán de acuerdo con el lugar y no indicarán todos los métodos, pero le serán a usted de gran utilidad para encontrar el camino.

La figura 4-1 implica también cierta responsabilidad al poner al día la relación. Debe haber algún modo de relacionar el Pedido con el Cliente. De nuevo, no se muestran los detalles; podría ser que se especifique el Cliente en el constructor del Pedido. O, tal vez, exista un método

agregar Pedido asociado al Cliente. Esto se puede hacer más explícito añadiendo operaciones al cuadro de clase (tal y como veremos más adelante).

Sin embargo, estas responsabilidades *no* implican una estructura de datos. A partir de un diagrama a nivel de especificación, no puedo hacer suposición alguna sobre la estructura de datos de las clases. No puedo, ni debo poder decir si la clase Pedido contiene un apuntador a Cliente, o si la clase Pedido cumple su responsabilidad ejecutando algún código de selección que pregunte a cada Cliente si se refiere a un Pedido dado. El diagrama sólo indica la interfaz, y nada más.

Si éste fuera un modelo de **implementación**, ahora daríamos a entender con ello que hay apuntadores en ambos sentidos entre las clases relacionadas. El diagrama diría entonces que Pedido tiene un campo que es un conjunto de apuntadores hacia Línea de pedido y también un apuntador a Cliente. En Java, podríamos deducir algo como lo que exponemos a continuación:

```
class Pedido {  
    private Cliente _cliente;  
    private Vector _lineasPedido;  
}  
  
class Cliente {  
    private Vector _pedidos;  
}
```

En este caso, no podemos inferir nada sobre la interfaz a partir de las asociaciones. Esta información nos la darían las operaciones sobre la clase.

Ahora véase la figura 4-3. Es básicamente la misma que la figura 4-1, a excepción de que he añadido un par de flechas a las líneas de asociación. Estas líneas indican **navegabilidad**.

En un modelo de especificación, esto indicaría que un Pedido tiene la responsabilidad de decir a qué Cliente corresponde, pero un Cliente no tiene la capacidad correspondiente para decir cuáles son los pedidos que tiene. En lugar de responsabilidades simétricas, aquí tenemos responsabilidades de un solo lado de la línea. En un diagrama de implementación se indicaría qué Pedido contiene un apuntador a Cliente pero Cliente no apuntaría a Pedido.

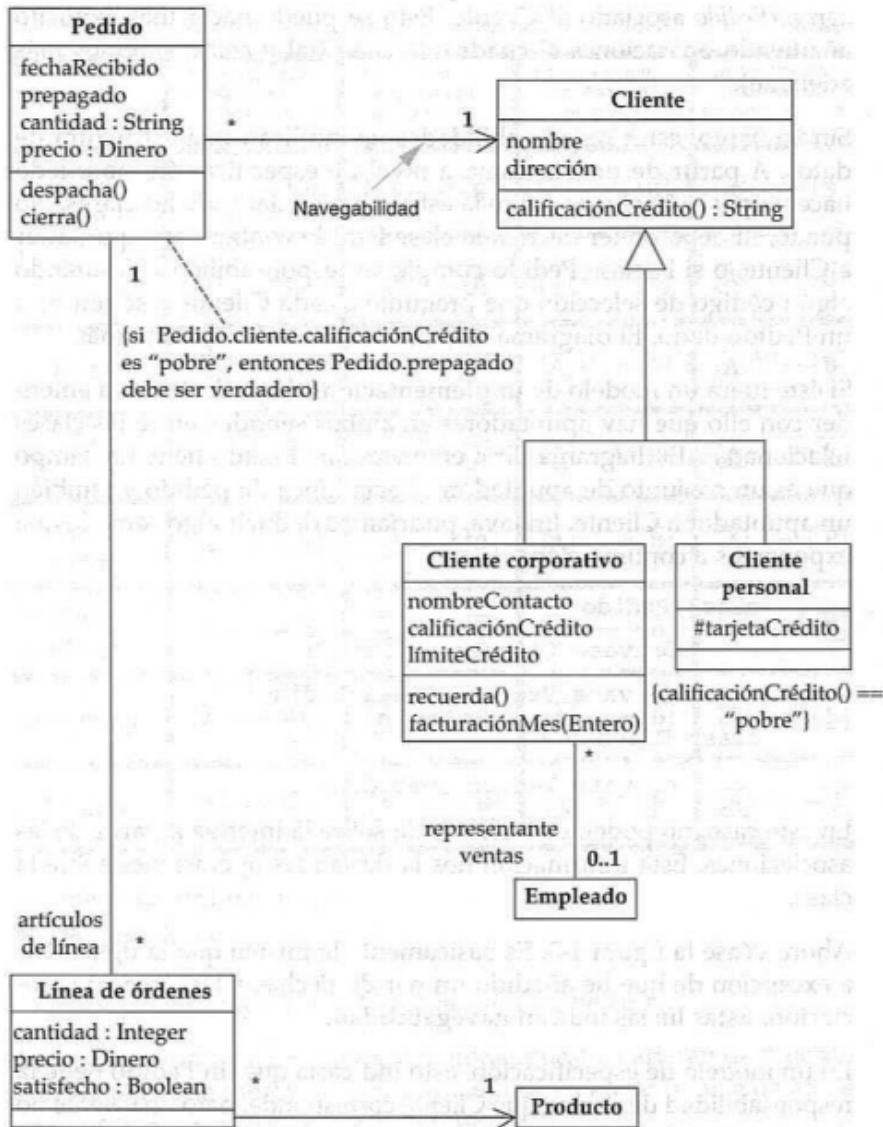


Figura 4-3: Diagrama de clase con navegabilidades

Como se podrá apreciar, la navegabilidad es una parte importante de los diagramas de implementación y especificación. Sin embargo, no pienso que la navegabilidad tenga utilidad en los diagramas conceptuales.

Frecuentemente verá diagramas conceptuales que primero no tendrán navegabilidades. Posteriormente, se agregarán las navegabilidades como parte del cambio a perspectivas de especificación y de implementación. Nótese también, que las navegabilidades posiblemente serán diferentes entre la especificación y la implementación.

Si existe una navegabilidad en una sola dirección, a la asociación se le llama **asociación unidireccional**. Una **asociación bidireccional** contiene navegabilidades en ambas direcciones. El UML dice que las asociaciones sin flechas significan que la navegabilidad es desconocida o que la asociación es bidireccional. El proyecto debe definirse en el sentido de uno u otro de los significados. Por mi parte, prefiero que signifique "sin decidir", en el caso de los modelos de especificación e implementación.

Las asociaciones bidireccionales incluyen una restricción adicional, que consiste en que ambos papeles son inversos entre ellos. Esto es igual al concepto de funciones inversas en las matemáticas. En el contexto de la figura 4-3, esto significa que cada Artículo de línea asociado con un Pedido se debe asociar con el Pedido original. De modo similar, si se toma una Línea de pedido y se busca en los artículos de línea el Pedido asociado, se deberá ver la Línea de pedido original en el conjunto. Esta cualidad se mantiene verdadera en las tres perspectivas.

Hay varias maneras de nombrar las asociaciones. A los modeladores de datos tradicionales les gusta denominar a una asociación con un verbo, de modo que se pueda emplear la relación en una oración. La mayoría de los modeladores de objetos prefieren valerse de sustantivos para denominar uno u otro papel, pues esta forma corresponde mejor a las responsabilidades y operaciones.

Algunos les dan nombres a todas las asociaciones. Yo prefiero nombrar las asociaciones sólo cuando mejora la comprensión. He visto demasiadas asociaciones con nombres como "tiene" o "se relaciona con". Si no existe un nombre para el papel, considero que su nombre es el de la clase señalada, como indiqué antes.

Atributos

Los atributos son muy parecidos a las asociaciones.

Desde un nivel conceptual, el atributo de nombre de un Cliente indica que los Clientes tienen nombres. Desde el nivel de especificación, este atributo indica que un objeto Cliente puede decir su nombre y tiene algún modo de establecer un nombre. En el nivel de implementación, un Cliente tiene un **campo** (también llamado variable de instancia o miembro de datos) para su nombre.

Dependiendo del detalle del diagrama, la notación de un atributo puede mostrar el nombre, el tipo y el valor predeterminado de un atributo (la sintaxis del UML es *visibilidad nombre: tipo = valor por omisión*, donde la *visibilidad* es igual que la de las operaciones, las cuales se describirán en la sección siguiente).

Por lo tanto, ¿cuál es la diferencia entre un atributo y una asociación?

Desde la perspectiva conceptual, no hay diferencia; un atributo sólo lleva consigo otro tipo de notación, la cual puede servir si se estima conveniente. Los atributos siempre son de valor único. Por lo general, los diagramas no indican si los atributos son opcionales u obligatorios (aunque, hablando estrictamente, deberían indicarlo).

La diferencia se da en los niveles de especificación y de implementación. Los atributos implican navegabilidad sólo del tipo al atributo. Es más, queda implícito que el tipo contiene únicamente su propia copia del objeto de atributo, lo que implica que cualquier tipo que se emplee como atributo tendrá un valor más que una semántica de referencia.

Hablaré sobre el valor y los tipos referenciales más adelante. Por el momento, lo mejor es considerar que los atributos son clases simples y pequeñas, tales como cadenas, fechas, objetos de moneda y valores no objeto, como *int* y *real*.

Operaciones

Las operaciones son los procesos que una clase sabe llevar a cabo. Evidentemente, corresponden a los métodos sobre una clase. En el nivel de especificación, las operaciones corresponden a los métodos públicos sobre un tipo. En general, no se muestran aquellas operaciones que simplemente manipulan atributos, ya que por lo común, se pueden inferir. Sin embargo, tal vez sea necesario indicar si un atributo dado es de sólo lectura o está **inmutable** congelado (esto es, que su valor nunca cambia). En el modelo de implementación, se podrían mostrar también las operaciones privadas y protegidas.

La sintaxis del UML completa para las operaciones es

visibilidad nombre (lista-de-parámetros) : expresión-tipo-de-dato-a-regresar {cadena-de-propiedades}

donde

- *visibilidad* es + (public), # (protected), o - (private)
- *nombre* es una cadena de caracteres (string)
- *lista-de-parámetros* contiene argumentos (opcionales) cuya sintaxis es la misma que la de los atributos
- *expresión-tipo-de-dato-a-regresar* es una especificación opcional dependiente del lenguaje
- *cadena-de-propiedades* indica valores de propiedad que se aplican a la operación dada

Un ejemplo de operación sería: + *últimaCantidadDe (valorTipoFenómeno) : Cantidad*

Dentro de los modelos conceptuales, las operaciones no deben tratar de especificar la interfaz de una clase. En lugar de ello, deberán indicar las principales responsabilidades de dicha clase, usando tal vez un par de palabras que sintetizan una responsabilidad de CRC (véase el recuadro).

Tarjetas de CRC

A fines de la década de 1980, uno de los centros más grandes de tecnología de objetos era el laboratorio de investigación de Tektronix, en Portland, Oregon, Estados Unidos. Este laboratorio tenía algunos de los principales usuarios de Smalltalk y muchas de las ideas clave de la tecnología de objetos se desarrollaron allí. Dos de sus programadores renombrados de Smalltalk eran Ward Cunningham y Kent Beck.

Tanto Cunningham como Beck estaban y siguen preocupados por cómo enseñar los profundos conocimientos de Smalltalk que habían logrado. De esta pregunta sobre cómo enseñar objetos surgió la sencilla técnica de las tarjetas de Clase-Responsabilidad-Colaboración (CRC).

En lugar de utilizar diagramas para desarrollar modelos, como lo hacían la mayoría de los metodólogos, Cunningham y Beck representaron las clases en tarjetas 4 x 6 [pulgadas]. Y en lugar de indicar atributos y métodos en las tarjetas, escribieron responsabilidades.

Ahora bien, ¿qué es una responsabilidad? En realidad es una descripción de alto nivel del propósito de una clase. La idea es tratar de eliminar la descripción de pedazos de datos y procesos y, en cambio, captar el propósito de la clase en unas cuantas frases. El que se haya seleccionado una tarjeta es deliberado. No se permite escribir más de lo que cabe en una tarjeta (véase la figura 4-4).

Nombre de la clase		
Responsabilidad	Pedido	Colaboración
<i>Revisa si hay elementos en existencia</i>		<i>Línea de pedido</i>
<i>Determina precio</i>		<i>Línea de pedido</i>
<i>Revisa si el pago es válido</i>		<i>Cliente</i>
<i>Despacha a la dirección de entrega</i>		

Figura 4-4: Tarjeta de Clase-Responsabilidad-Colaboración (CRC)

La segunda C se refiere a los colaboradores. Con cada responsabilidad se indica cuáles son las otras clases con las que se tiene que trabajar para cumplirla. Esto da cierta idea sobre los vínculos entre las clases, siempre a alto nivel.

Uno de los principales beneficios de las tarjetas de CRC es que alientan la discusión animada entre los desarrolladores. Son especialmente eficaces cuando se está en medio de un caso de uso para ver cómo lo van a implementar las clases. Los desarrolladores escogen tarjetas a medida que cada clase colabora en el caso de uso. Conforme se van formando ideas sobre las responsabilidades, se pueden escribir en las tarjetas. Es importante pensar en las responsabilidades, ya que evita pensar en las clases como simples depositarias de datos, y ayuda a que el equipo se centre en comprender el comportamiento de alto nivel de cada clase.

Un error común que veo que comete la gente es generar largas listas de responsabilidades de bajo nivel. Este procedimiento es completamente fallido. Las responsabilidades deben caber sin dificultad en una tarjeta. Yo cuestionaría cualquier tarjeta que tenga más de tres responsabilidades. Plantéese la pregunta de si se deberá dividir la clase y si las responsabilidades se podrían indicar mejor integrándolas en enunciados de un mayor nivel.

Cuándo usar las tarjetas de CRC

Algunos consideran maravillosas las tarjetas de CRC; en cambio, a otros, esta técnica los deja indiferentes.

Yo considero definitivamente que se deberían probar, a fin de saber si al equipo de trabajo le gusta trabajar con ellas. Se deben usar, en particular, si el equipo se ha empantanado en demasiados detalles o si parecen identificar clases apelmazadas y carentes de definiciones claras.

Se pueden emplear diagramas de clase y diagramas de interacciones (véase el capítulo 6) para captar y formalizar los resultados del modelado CRC en un diseño con notación de UML. Asegúrese de

que cada clase en su diagrama de clase tiene un enunciado de sus responsabilidades.

Para mayor información

Desafortunadamente, Cunningham y Beck no han escrito un libro sobre las CRC, pero usted puede encontrar su artículo original (Beck y Cunningham 1989) en la Web (<http://c2.com/doc/oopsla89/paper.html>). En general, el libro que mejor describe esta técnica y, de hecho, todo el concepto del uso de responsabilidades es el de Rebeca Wirfs-Brock (1990). Es un libro relativamente viejo según las normas de la OO, pero se ha añejado bien.

Con frecuencia encuentro útil hacer la distinción entre aquellas operaciones que cambian el estado de una clase y aquellas que no lo hacen. Una **consulta** es una operación que obtiene un valor de una clase sin que cambie el estado observable de tal clase. El **estado observable** de un objeto es el estado que se puede determinar a partir de sus consultas asociadas.

Considérese un objeto de Cuenta que calcula su balance a partir de una lista de entradas. Para mejorar el desempeño, Cuenta puede poner en un campo caché el resultado del cálculo del balance, para consultas futuras. Por tanto, si el caché está vacío, la primera vez que se llama a la operación "balance", pondrá el resultado en el campo caché. La operación "balance" cambia así el estado real del objeto Cuenta, pero no su estado observable, pues todas las consultas devuelven el mismo valor, esté o no lleno el campo caché. Las operaciones que sí cambian el estado observable de un objeto se llaman **modificadores**.

Considero útil tener perfectamente clara la diferencia entre consultas y modificadores. Las consultas se pueden ejecutar en cualquier orden, pero la secuencia de los modificadores es más importante. Yo tengo como política evitar que los modificadores devuelvan valores, con el fin de mantenerlos separados.

Otros términos que algunas veces se presentan son: métodos de obtención y métodos de establecimiento. El **método de obtención** (*get-*

ting) es un método que devuelve el valor de un campo (sin hacer nada más). El **método de establecimiento** (*setting*) pone un valor en un campo (y nada más). Desde afuera, el cliente no debe poder saber si una consulta es un método de obtención ni si un modificador es un método de establecimiento. El conocimiento sobre los métodos de obtención y establecimiento está contenido completamente dentro de la clase.

Otra distinción es la que se da entre operación y método. Una **operación** es algo que se invoca sobre un objeto (la llamada de procedimiento), mientras que un **método** es el cuerpo del procedimiento. Los dos son diferentes cuando se tiene polimorfismo. Si se tiene un supertipo con tres subtipos, cada uno de los cuales suplanta la operación “foo” del supertipo, entonces lo que hay es una operación y cuatro métodos que la implementan.

Es muy común que operación y método se empleen indistintamente, pero hay veces en que es necesario precisar la diferencia. En algunas ocasiones, la gente distingue entre una y otro mediante los términos llamada a un método, o declaración de método (en lugar de operación), y cuerpo del método.

Los lenguajes tienen sus propias convenciones para los nombres. En C++, las operaciones se llaman funciones miembro; en Smalltalk, operaciones de métodos. C++ también emplea el término miembros de una clase para denominar las operaciones y métodos de una clase.

Generalización

Un ejemplo característico de generalización comprende al personal y las corporaciones clientes de una empresa. Unos y otros tienen diferencias, pero también muchas similitudes. Las similitudes se pueden colocar en una clase general Cliente (el supertipo) con los subtipos siguientes: Cliente personal y Cliente corporativo.

Este fenómeno también está sujeto a diversas interpretaciones, según el nivel de modelado. Por ejemplo, conceptualmente, podemos decir que el Cliente corporativo es un subtipo de Cliente, si todas las instancias de Cliente corporativo, también son, por definición, instancias de

Cliente. Entonces, un Cliente corporativo es un tipo especial de Cliente. El concepto clave es que todo lo que digamos de un Cliente (asociaciones, atributos, operaciones) también vale para un Cliente corporativo.

En un modelo de especificación, la generalización significa que la interfaz del subtipo debe incluir todos los elementos de la interfaz del supertipo. Se dice, entonces, que la interfaz del subtipo se conforma con la interfaz del supertipo.

Otra manera de considerar esto involucra el principio de la **sustituibilidad**. Yo debo poder sustituir a un Cliente corporativo con cualquier código que requiera un Cliente y todo debe operar sin problemas. En esencia, esto significa que, si escribo un código suponiendo que tengo un Cliente, puedo entonces usar con toda libertad cualquier subtipo de Cliente. El Cliente corporativo puede responder a ciertos comandos de manera diferente a otro Cliente (por el principio del polimorfismo), pero el invocador no debe preocuparse por la diferencia.

La generalización desde la perspectiva de implementación se asocia con la herencia en los lenguajes de programación. La subclase hereda todos los métodos y campos de la superclase y puede suplantarlos.

El punto clave aquí es la diferencia entre generalización desde la perspectiva de la especificación (subtipificación o herencia de interfaces) y desde la perspectiva de la implementación (subclasificación o herencia de implementaciones). La subclasificación (formación de subclases) es una manera de implementar la subtipificación. La subtipificación (formación de subtipos) también se puede implementar mediante delegación; de hecho, muchos de los patrones descritos en Gamma *et al.* (1994) se refieren a maneras de tener dos clases con interfaces similares sin servirse de la subclasificación. También se puede consultar el libro de los "pragmáticos" Martin y Odell (1996) o el de Fowler (1997), para más ideas sobre la implementación de subtipificación.

En el caso de estas dos formas de generalización, se deberá asegurar siempre que también tenga validez la generalización conceptual. He encontrado que, si no se hace lo anterior, se puede uno ver en problemas, debido a que la generalización no es estable cuando hay necesidad de efectuar cambios posteriores.

En algunas ocasiones aparecen casos en los que un subtipo tiene la misma interfaz que el supertipo, pero el subtipo implementa sus operaciones de modo diferente. Si éste es el caso, se podría decidir no mostrar el subtipo en el diagrama de perspectiva de especificación. Así lo hago con frecuencia si para los usuarios de la clase es interesante la existencia de los tipos, pero no lo hago cuando los subtipos varían únicamente debido a razones internas de implementación.

Reglas de restricción

Una buena parte de lo que se hace cuando se dibuja un diagrama de clase es indicar las condiciones limitantes o restricciones.

La figura 4-3 indica que un Pedido únicamente puede ser colocado por un solo Cliente. El diagrama implica también que Artículos de línea se considera por separado: digamos 40 adminículos de color marrón, 40 adminículos azules y 40 adminículos rojos, y no 40 artefactos rojos, azules y marrones. El diagrama dice, además, que los Clientes corporativos tienen límites de crédito, pero que los Clientes personales no los tienen.

Los artificios básicos de la asociación, el atributo y la generalización, colaboran de manera significativa con la especificación de condiciones importantes, pero no pueden indicarlas todas. Estas restricciones aún necesitan ser captadas; el diagrama de clase es un lugar adecuado para hacerlo.

El UML no define una sintaxis estricta para describir las restricciones, aparte de ponerlas entre llaves ({}). Yo prefiero escribir en lenguaje informal, para simplificar su lectura. Puede usarse también un enunciado más formal, como un cálculo de predicados o alguna derivada. Otra alternativa es escribir un fragmento de código de programación.

Idealmente, las reglas se deberían implementar como afirmaciones en el lenguaje de programación. Éstas se corresponden con el concepto de invariantes del Diseño por contrato (véase el recuadro). En lo personal, soy partidario de crear un método *revisaInvariante* e invocarlo desde algún código de depuración que ayude a comprobar los invariantes.

Diseño por contrato

El Diseño por contrato es una técnica diseñada por Bertrand Meyer. Esta técnica es una característica central del lenguaje Eiffel, que él desarrolló. Sin embargo, el Diseño por contrato no es específico de Eiffel; es una técnica valiosa que se puede usar con cualquier lenguaje de programación.

En el corazón del Diseño por contrato se encuentra la afirmación. Una **afirmación** es un enunciado Booleano que nunca debe ser falso y, por tanto, sólo lo será debido a una falla. Por lo común, las afirmaciones se comprueban sólo durante la depuración y no durante la ejecución en producción. De hecho, un programa nunca debe suponer que se están comprobando las afirmaciones.

El Diseño por contrato se vale de tres tipos de afirmaciones: poscondiciones, precondiciones e invariantes.

Las precondiciones y las poscondiciones se aplican a las operaciones. Una **poscondición** es un enunciado sobre cómo debería verse el mundo después de la ejecución de una operación. Por ejemplo, si definimos la operación "cuadrado" sobre un número, la poscondición adoptaría la forma de *resultado* = *éste* * *éste*, donde *resultado* es el producto y *éste* es el objeto sobre el cual se invocó la operación. La poscondición es una forma útil de decir qué es lo que hacemos, sin decir cómo lo hacemos; en otras palabras, de separar la interfaz de la implementación.

La **precondición** es un enunciado de cómo esperamos encontrar el mundo, antes de ejecutar una operación. Podemos definir una precondición para la operación "cuadrado" de *éste* ≥ 0 . Tal precondición dice que es un error invocar a "cuadrado" sobre un número negativo y que las consecuencias de hacerlo son indefinidas.

A primera vista, ésta no parece ser una buena idea, pues deberíamos poner una comprobación en alguna parte para garantizar que se invoque correctamente a "cuadrado". La cuestión importante es quién es responsable de hacerlo.

La precondición hace explícito que quien invoca es el responsable de la comprobación. Sin este enunciado explícito de responsabilidades, podemos tener muy poca comprobación (pues ambas partes suponen que la otra es la responsable) o tenerla en demasía (cuando ambas partes lo hacen). Demasiada comprobación es mala, ya que provoca que se duplique muchas veces el código de comprobación, lo cual puede complicar de manera importante un programa. El ser explícito sobre quién es el responsable contribuye a reducir esta complejidad. Se reduce el peligro de que el invocador olvide comprobar por el hecho de que, por lo general, las afirmaciones se comprueban durante las pruebas y la depuración.

A partir de estas definiciones de precondición y poscondición, podemos ver una definición sólida del término **excepción**, que ocurre cuando se invoca una operación estando satisfecha su precondición y, sin embargo, no puede regresar con su poscondición satisfecha.

Una **invariante** es una afirmación acerca de una clase. Por ejemplo, la clase Cuenta puede tener una invariante que diga que `balance == sum(entradas.cantidad())`. La invariante “siempre” es verdadera para todas las instancias de la clase. En este contexto, “siempre” significa “siempre que el objeto esté disponible para que se invoque una operación sobre ella”.

En esencia, lo anterior significa que la invariante se agrega a las precondiciones y poscondiciones asociadas a todas las operaciones públicas de la clase dada. La invariante puede volverse falsa durante la ejecución de un método, pero debe haberse restablecido a verdadera para el momento en que cualquier otro objeto pueda hacerle algo al receptor.

Las afirmaciones pueden desempeñar un papel único en la subclaseificación.

Uno de los peligros del polimorfismo es que se podrían redefinir las operaciones de una subclase, de tal modo que fueran inconsistentes con las operaciones de la superclase. Las afirmaciones

impiden hacer esto. Las invariantes y poscondiciones de una clase deben aplicarse a todas las subclases. Las subclases pueden elegir el refuerzo de estas afirmaciones, pero no pueden debilitarlas. La precondition, por otra parte, no puede reforzarse, aunque sí debilitarse.

Esto parecerá extraño a primera vista, pero es importante para permitir vínculos dinámicos. Siempre debería ser posible tratar un objeto de subclase como si fuera una instancia de su superclase (de acuerdo con el principio de la sustituibilidad). Si una subclase refuerza su precondition, entonces podría fallar una operación de superclase, cuando se aplicara a la subclase.

En esencia, las afirmaciones sólo pueden aumentar las responsabilidades de la subclase. Las precondiciones son un enunciado para trasladar una responsabilidad al invocador; se incrementan las responsabilidades de una clase cuando se debilita una precondition. En la práctica, todo esto permite un mucho mejor control de la subclasificación y ayuda a asegurar que las subclases se comporten de manera adecuada.

Idealmente, las afirmaciones se deben incluir en el código, como parte de la definición de la interfaz. Los compiladores deben poder encender la comprobación por afirmaciones durante las depuraciones y apagarla durante la operación en producción. Se pueden manejar varias etapas de comprobación de afirmaciones. Las precondiciones a menudo ofrecen las mejores posibilidades de atrapar errores con la menor cantidad de sobrecarga de proceso.

Cuándo utilizar el Diseño por contrato

El Diseño por contrato es una técnica valiosa que siempre se debe emplear cuando se programa. Es particularmente útil para la construcción de interfaces claras.

Sólo Eiffel maneja afirmaciones como parte de su lenguaje, pero desafortunadamente Eiffel no es un lenguaje de amplia difusión. La adición de mecanismos a C++ y Smalltalk que manejen algunas

afirmaciones es directa. Es mucho más engorroso hacer algo similar en Java, pero es posible.

El UML no habla mucho acerca de las afirmaciones, pero se pueden usar sin problemas. Las invariantes son equivalentes a las reglas de condición en los diagramas de clase y se deberán usar tanto como sea posible. Las precondiciones y poscondiciones de operación se deben documentar con las definiciones de las operaciones.

Para mayor información

El libro de Meyer (1997) es una obra clásica (aunque, a estas alturas, muy denso) sobre el diseño OO, donde se habla mucho de las afirmaciones. Kim Walden y Jean-Marc Nerson (1995) y Steve Cook y John Daniels (1994) utilizan ampliamente el Diseño por contrato en sus obras.

También se puede obtener mayor información de ISE (la compañía de Bertrand Meyer).

Cuándo emplear los diagramas de clases

Los diagramas de clase son la columna vertebral de casi todos los métodos OO, por lo cual usted descubrirá que se emplean todo el tiempo. Este capítulo ha abarcado los conceptos básicos; el capítulo 5 analizará muchos de los conceptos más avanzados.

El problema con los diagramas de clase es que son tan ricos que pueden resultar abrumadores. A continuación, damos algunos consejos breves.

- No trate de usar todas las anotaciones a su disposición. Comience con el material sencillo de este capítulo: clases, asociaciones, atributos y generalización. Introduzca otras anotaciones del capítulo 5 sólo cuando las necesite.
- Ajuste la perspectiva desde la cual se dibujan los modelos a la etapa del proyecto.

- Si se está en la etapa del análisis, dibuje modelos conceptuales.
- Cuando se trabaje con software, céntrese en los modelos de especificación.
- Dibuje modelos de implementación sólo cuando se esté ilustrando una técnica de implementación en particular.
- No dibuje modelos para todo; por el contrario, céntrese en las áreas clave. Es mejor usar y mantener al día unos cuantos diseños que tener muchos modelos olvidados y obsoletos.

El principal peligro con los diagramas de clase es que se puede quedar empantanado muy pronto en los detalles de la implementación. Para contrarrestar esto, céntrese en las perspectivas conceptual y de especificación. Si cae en estos problemas, encontrará muy útiles las tarjetas de CRC (véase la página 74).

Para mayor información

Los libros de los tres amigos serán la referencia definitiva para todo lo relacionado con los diagramas de clase. Mi consejo sobre otras fuentes depende de si usted prefiere una perspectiva de implementación o conceptual. Para la perspectiva de implementación, es recomendable Booch (1994); para una perspectiva conceptual son recomendables los libros de "fundamentos" de Martin y Odell (1994). Una vez que haya leído su primera selección, lea la otra, pues ambas perspectivas son importantes. Después, cualquier buen libro sobre OO le dará algunos puntos de vista interesantes. Tengo una particular predilección por el de Cook y Daniels (1994), debido a su manejo de las perspectivas y a la formalidad que introducen los autores.