UML

[redacted]

CONTROL #3
(CAPS. 7 AL 10)
RUMBAUGH

LENGUAJE UNIFICADO DE MODELADO

Capítulo 7

DIAGRAMAS

En este capítulo

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

El modelado se discute en el Capítulo 1.

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

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

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

Introducción

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

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

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

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

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

En el contexto del software hay cinco vistas complementarias que son las más importantes para visualizar, especificar, construir y documentar una arquitectura software: la vista de casos de uso, la vista de diseño, la vista de interacción, la vista de implementación y la vista de despliegue. Cada una de estas vistas involucra modelado estructural (modelado de cosas estáticas), así como modelado de comportamiento (modelado de cosas dinámicas). Juntas, estas diferentes vistas capturan las decisiones más importantes sobre el sistema. Individualmente, cada una de estas vistas permite centrar la atención en una perspectiva del sistema para poder razonar con claridad sobre las decisiones.

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

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

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

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

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

Términos y conceptos

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

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

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

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

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

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

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

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

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

Los paquetes se discuten en el Capítulo 12.

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

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

Diagramas estructurales

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

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

1. Diagramas de clases.	Clases, interfaces y colaboraciones.
2. Diagramas de componentes.	Componentes.
3. Diagramas de estructura compuesta.	Estructura interna.
4. Diagramas de objetos.	Objetos.
5. Diagramas de artefactos.	Artefactos.
6. Diagramas de despliegue.	Nodos.

Los diagramas de clases se discuten en el Capítulo 8.

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

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

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

Diagramas de estructura compuesta. Un *diagrama de estructura compuesta* muestra la estructura interna de una clase o una colaboración. La diferencia entre componentes y estructura compuesta es mínima, y este libro trata a ambos como diagramas de componentes.

Los diagramas de objetos se discuten en el Capítulo 14.

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

Los diagramas de artefactos se discuten en el Capítulo 30.

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

Los diagramas de despliegue se discuten en el Capítulo 31.

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

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

Diagramas de comportamiento

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

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

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

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

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

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

Los diagramas de secuencia se discuten en el Capítulo 19.

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

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

Los diagramas de comunicación se discuten en el Capítulo 19.

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

Los diagramas de estados se discuten en el Capítulo 25.

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

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

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

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

Técnicas comunes de modelado

Modelado de diferentes vistas de un sistema

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

Para modelar un sistema desde diferentes vistas, es necesario:

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

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

■ Vista de casos de uso.	Diagramas de casos de uso.
■ Vista de diseño.	Diagramas de clases (para modelado estructural).
■ Vista de interacción.	Diagramas de interacción (para modelado del comportamiento).
■ Vista de implementación.	Diagramas de estructura compuesta.
■ Vista de despliegue.	No se requiere.

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

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

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

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

Modelado a diferentes niveles de abstracción

No sólo es necesario ver un sistema desde varios ángulos, sino que habrá situaciones en las que diferentes personas implicadas en el desarrollo necesiten la misma vista del sistema, pero a diferentes niveles de abstracción. Por ejemplo, dado un conjunto de clases que capturen el vocabulario del espacio del problema, un programador podría necesitar una vista detallada a nivel de atributos, operaciones y relaciones de cada clase. Por otro lado, un analista que esté inspeccionando varios escenarios de casos de uso junto a un usuario final, quizás necesite una vista mucho más escueta de las mismas clases. En este contexto, el programador está trabajando a un nivel menor de abstracción, y el analista y el usuario final lo hacen a un nivel mayor, pero todos trabajan sobre el mismo modelo. De hecho, ya que los diagramas son sólo una presentación gráfica de los elementos que constituyen un modelo, se pueden crear varios diagramas para el mismo o diferentes modelos, cada uno ocultando o exponiendo diferentes conjuntos de esos elementos, y cada uno mostrando diferentes niveles de detalle.

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

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

- Hay que considerar las necesidades de las personas que utilizarán el diagrama, y comenzar con un modelo determinado.
- Si se va a usar el modelo para hacer una implementación, harán falta diagramas a un menor nivel de abstracción, que tendrán que revelar muchos detalles. Si se va a usar para presentar un modelo conceptual a un usuario final, harán falta diagramas a un mayor nivel de abstracción, que tendrán que ocultar muchos detalles.
- Según donde uno se ubique en este espectro de niveles de abstracción, hay que crear un diagrama del nivel de abstracción apropiado, ocultando o revelando las cuatro categorías siguientes de elementos del modelo:

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

1. *Bloques de construcción y relaciones*: ocultar los que no sean relevantes para el objetivo del diagrama o las necesidades del usuario.
2. *Adornos*: revelar sólo los adornos de los bloques y relaciones que sean esenciales para comprender el objetivo.
3. *Flujo*: en el contexto de los diagramas de comportamiento, considerar sólo aquellos mensajes o transiciones esenciales para comprender el objetivo.
4. *Estereotipos*: en el contexto de los estereotipos utilizados para clasificar listas de elementos, como atributos y operaciones, revelar sólo aquellos elementos estereotipados esenciales para comprender el objetivo.

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

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

Las dependencias de traza se discuten en el Capítulo 32.

- Considerar las necesidades de las personas que utilizarán el diagrama y decidir el nivel de abstracción que debería ver cada una, creando un modelo separado para cada nivel.
- En general, poblar los modelos a mayor nivel de abstracción con abstracciones simples, y los modelos a un nivel más bajo de abstracción con abstracciones detalladas. Establecer dependencias de traza entre los elementos relacionados de diferentes modelos.
- En la práctica, si se siguen las cinco vistas de una arquitectura, al modelar un sistema a diferentes niveles de abstracción, se producen cuatro situaciones con frecuencia:

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

1. *Casos de uso y su realización*: los casos de uso en un modelo de casos de uso se corresponden con colaboraciones en un modelo de diseño.
2. *Colaboraciones y su realización*: las colaboraciones se corresponden con una sociedad de clases que trabajan juntas para llevar a cabo la colaboración.
3. *Componentes y su diseño*: los componentes en un modelo de implementación se corresponden con los elementos en un modelo de diseño.
4. *Nodos y sus componentes*: los nodos en un modelo de despliegue se corresponden con componentes en un modelo de implementación.

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

Los diagramas de interacción se discuten en el Capítulo 19.

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

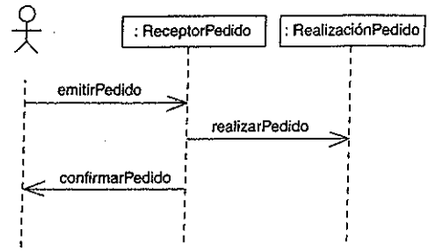


Figura 7.1: Diagrama de interacción a un alto nivel de abstracción.

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

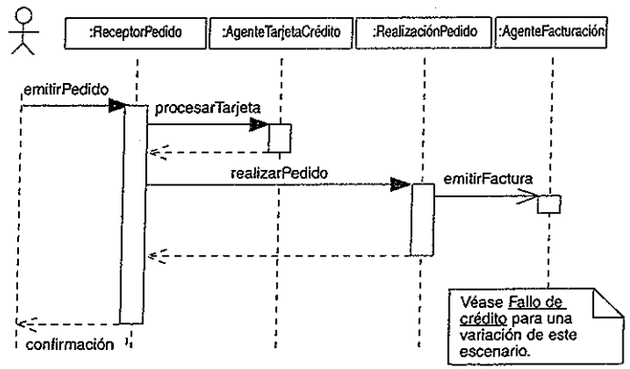


Figura 7.2: Interacción a un nivel de abstracción bajo.

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

Modelado de vistas complejas

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

Para modelar vistas complejas:

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

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

Sugerencias y consejos

Cuando se cree un diagrama:

- Hay que recordar que el propósito de un diagrama en UML no es dibujar bonitas imágenes, sino visualizar, especificar, construir y documentar. Los diagramas son un medio para el fin de implantar un sistema ejecutable.
- No hay por qué conservar todos los diagramas. Debe considerarse la construcción de diagramas sobre la marcha, inspeccionando los elementos en los modelos, y hay que utilizar esos diagramas para razonar sobre el sistema mientras se construye. Muchos de estos diagramas pueden desecharse después de haber servido a su propósito (pero la semántica bajo la que se crearon permanecerá como parte del modelo).
- Hay que evitar diagramas extraños o redundantes. Éstos complican los modelos.
- Hay que revelar sólo el detalle suficiente en cada diagrama para abordar las cuestiones para las que se pensó. La información extraña puede distraer al lector del punto clave que se desea resaltar.
- Por otro lado, no hay que hacer los diagramas minimalistas, a menos que realmente se necesite presentar algo a un nivel muy alto de abstracción. Simplificar en exceso puede ocultar detalles que tal vez sean importantes para razonar sobre los modelos.
- Hay que mantener un equilibrio entre los diagramas de comportamiento y los estructurales en el sistema. Muy pocos sistemas son totalmente estáticos o totalmente dinámicos.
- No hay que hacer diagramas demasiado grandes (los que ocupan varias páginas impresas son difíciles de navegar) ni demasiado pequeños (puede plantearse unir varios diagramas triviales en uno).
- Hay que dar a cada diagrama un nombre significativo que exprese su objetivo claramente.
- Hay que mantener organizados los diagramas. Deben agruparse en paquetes según la vista.
- No hay que obsesionarse con el formato de un diagrama. Hay que dejar que las herramientas ayuden.

Un diagrama bien estructurado:

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

Cuando se dibuje un diagrama:

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

Capítulo 8

DIAGRAMAS DE CLASES

En este capítulo

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

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

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

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

Introducción

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

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

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

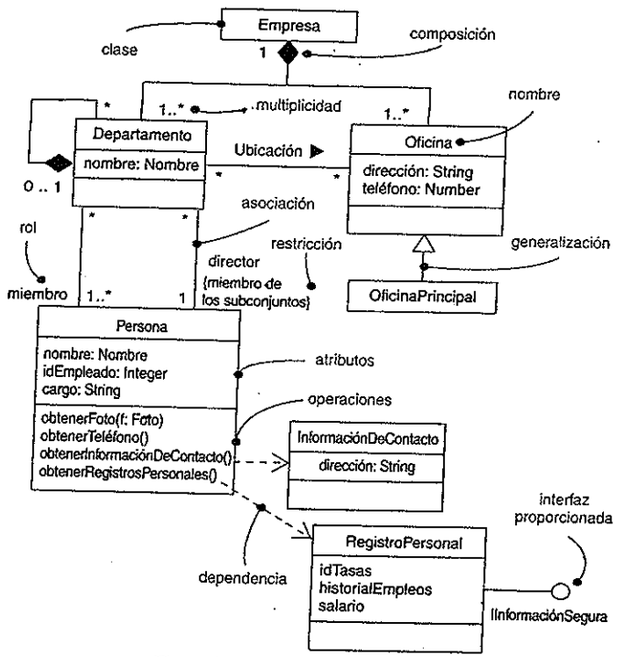


Figura 8.1: Un diagrama de clases.

Términos y conceptos

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

Propiedades comunes

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

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

Contenido

Las clases se discuten en los Capítulos 4 y 9; las interfaces se discuten en el Capítulo 11; las relaciones se discuten en los Capítulos 5 y 10; los paquetes se discuten en el Capítulo 12; los subsistemas se discuten en el Capítulo 32; las instancias se discuten en el Capítulo 13.

Los diagramas de clases contienen normalmente los siguientes elementos:

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

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

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

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

Usos comunes

Las vistas de diseño se discuten en el Capítulo 2.

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

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

1. Para modelar el vocabulario de un sistema.

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

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

Las colaboraciones se discuten en el Capítulo 28.

2. Para modelar colaboraciones simples.

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

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

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

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

Técnicas comunes de modelado

Modelado de colaboraciones simples

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

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

Para modelar una colaboración:

Los mecanismos como este están relacionados a menudo con los casos de uso, como se discute en el Capítulo 17; los escenarios son hilos a través de un caso de uso, como se discute en el Capítulo 16.

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

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

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

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

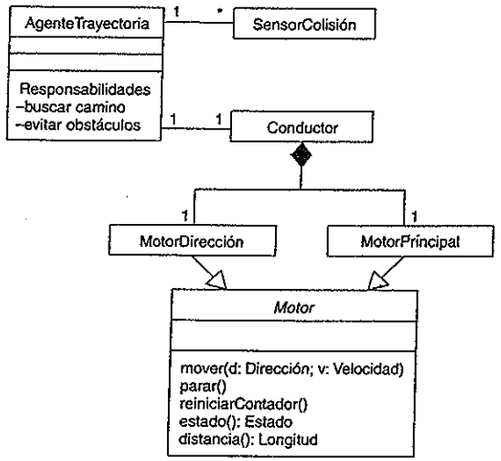


Figura 8.2: Modelado de colaboraciones simples.

Modelado de un esquema lógico de base de datos

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

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

Los diagramas de clases de UML son un superconjunto de los diagramas entidad-relación (E-R), una herramienta de modelado para el diseño lógico de bases de datos utilizada con mucha frecuencia. Mientras que los diagramas E-R clásicos se centran sólo en los datos, los diagramas de clases van un paso más allá, pues permiten el modelado del comportamiento. En la base de datos física, estas operaciones lógicas normalmente se convierten en disparadores (*triggers*) o procedimientos almacenados.

Para modelar un esquema:

Los estereotipos se discuten en el Capítulo 6.

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

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

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

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

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

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

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

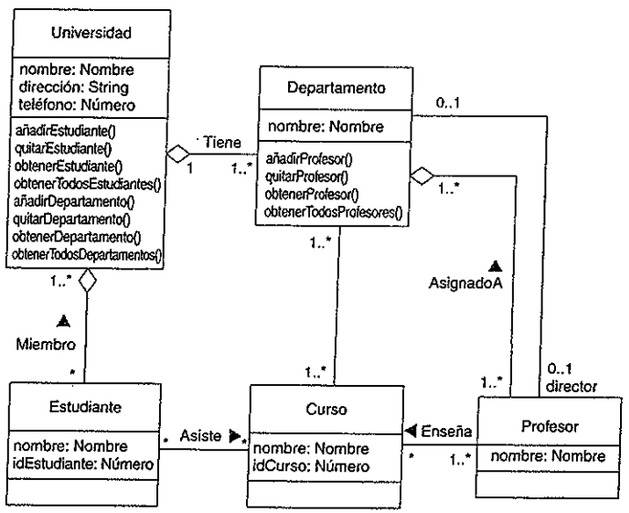


Figura 8.3: Modelado de un Esquema.

Ingeniería directa e inversa

La importancia del modelado se discute en el Capítulo 1.

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

Los diagramas de actividades se discuten en el Capítulo 20.

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

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

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

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

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

Para hacer ingeniería directa con un diagrama de clases:

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

Los patrones se discuten en el Capítulo 29.

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

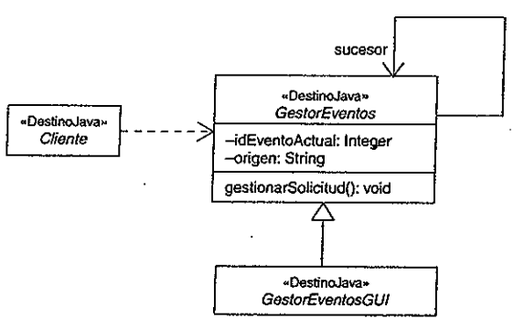


Figura 8.4: Ingeniería directa.

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

```
public abstract class GestorEventos {

  GestorEventos sucesor;
  private Integer idEventoActual;
  private String origen;

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

}
```

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

Para hacer ingeniería inversa sobre un diagrama de clases:

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

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

Sugerencias y consejos

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

Un diagrama de clases bien estructurado:

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

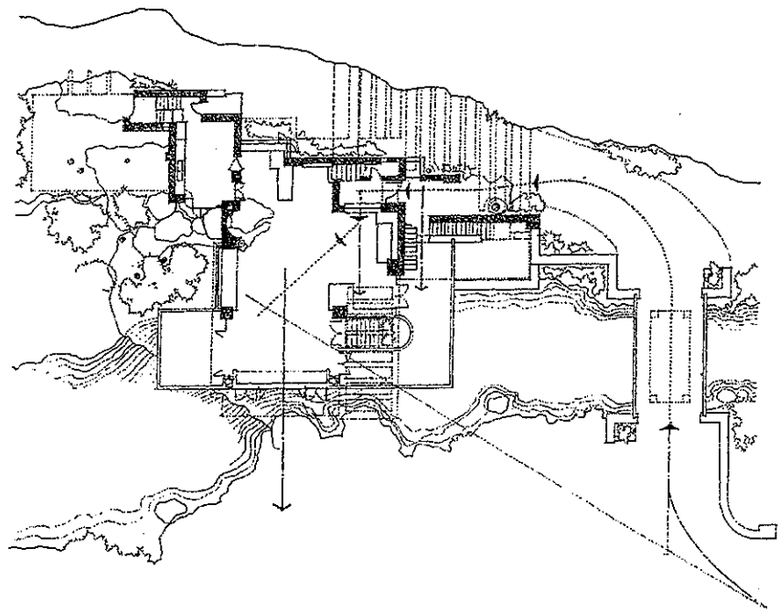
Cuando se dibuje un diagrama de clases:

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

LENGUAJE UNIFICADO DE MODELADO

Parte 3

MODELADO ESTRUCTURAL AVANZADO



LENGUAJE UNIFICADO DE MODELADO

Capítulo 9

CARACTERÍSTICAS AVANZADAS DE LAS CLASES

En este capítulo

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

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

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

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

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

Introducción

La arquitectura se discute en el Capítulo 2.

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

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

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

Las responsabilidades se discuten en el Capítulo 6.

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

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

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

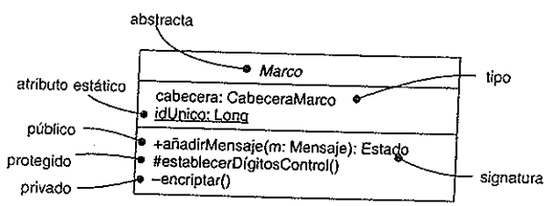


Figura 9.1: Características avanzadas de las clases.

Términos y conceptos

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

Clasificadores

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

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

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

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

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

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

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

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

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

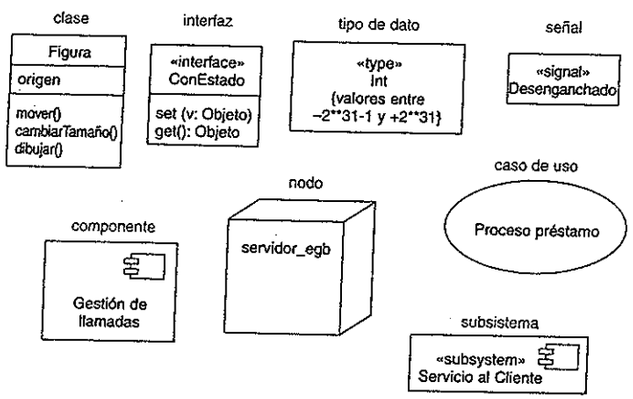


Figura 9.2: Clasificadores.

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

Visibilidad

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

Un clasificador puede ver a otro clasificador si éste está en su alcance, y si existe una relación implícita o explícita hacia él; las relaciones se discuten en los Capítulos 5 y 10; los descendientes provienen de las relaciones de generalización, como se discute en el Capítulo 5; los permisos permiten a un clasificador compartir sus características privadas, como se discute en el Capítulo 10.

1. `public` Cualquier clasificador externo con visibilidad hacia el clasificador dado puede utilizar la característica; se especifica precediéndola del símbolo +.
2. `protected` Cualquier descendiente del clasificador puede utilizar la característica; se especifica precediéndola del símbolo #.
3. `private` Sólo el propio clasificador puede utilizar la característica; se especifica precediéndola del símbolo –.
4. `package` Sólo los clasificadores declarados en el mismo paquete pueden utilizar la característica; se especifica precediéndola del símbolo ~.

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

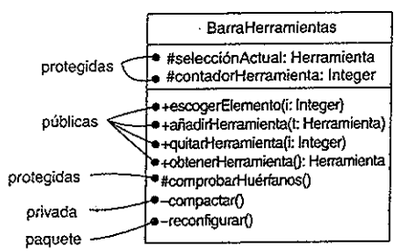


Figura 9.3: Visibilidad.

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

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

Alcance de instancia y estático

Las instancias se discuten en el Capítulo 13.

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

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

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

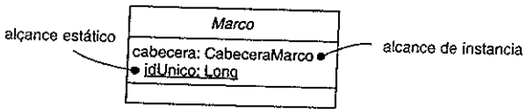


Figura 9.4: Alcance de propiedad.

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

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

El alcance estático funciona de manera ligeramente distinta para las operaciones. Una operación de instancia tiene un parámetro implícito que se corresponde con el objeto que se está manipulando. Una operación estática no tiene ese parámetro; se comporta como un procedimiento global tradicional, sin un objeto destinatario. Las operaciones estáticas se utilizan para operaciones que crean instancias u operaciones que manipulan atributos estáticos.

Elementos abstractos, hojas y polimórficos

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

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

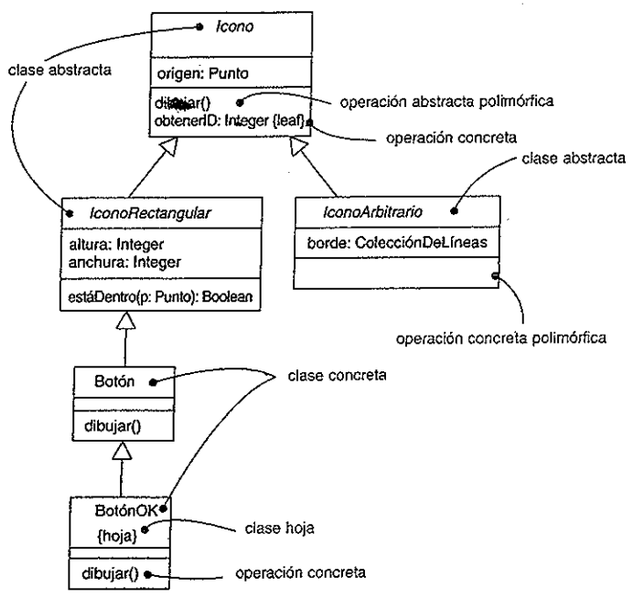


Figura 9.5: Clases y operaciones abstractas y concretas.

Los mensajes se discuten en el Capítulo 16.

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

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

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

Multiplicidad

Las instancias se discuten en el Capítulo 13.

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

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

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

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

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

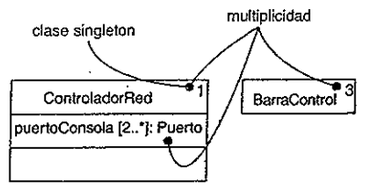


Figura 9.6: Multiplicidad.

Los clasificadores estructurados se discuten en el Capítulo 15.

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

Atributos

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

También se pueden utilizar los estereotipos para designar conjuntos de atributos relacionados, como los atributos de mantenimiento, como se discute en el Capítulo 6.

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

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

Por ejemplo, las siguientes son declaraciones legales de atributos:

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

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

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

Operaciones

Las señales se discuten en el Capítulo 21.

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

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

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

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

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

Por ejemplo, las siguientes son declaraciones legales de operaciones:

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

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

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

Dirección puede tomar uno de los siguientes valores:

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

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

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

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

3. `guarded` La semántica e integridad del objeto se garantizan en presencia de múltiples flujos de control por medio de la secuenciación de todas las llamadas a todas las operaciones `guarded` del objeto. Así, se puede invocar exactamente una operación a un mismo tiempo sobre el objeto, reduciendo esto a una semántica secuencial.

4. `concurrent` La semántica e integridad del objeto se garantizan en presencia de múltiples flujos de control, tratando la operación como atómica. Pueden ocurrir simultáneamente múltiples llamadas desde diferentes flujos de control a cualquier operación concurrente, y todas pueden ejecutarse concurrentemente con semántica correcta; las operaciones concurrentes deben diseñarse para ejecutarse correctamente en caso de que haya una operación secuencial concurrente o con guarda sobre el mismo objeto.

5. `static` La operación no tiene un parámetro implícito para el objeto destino; se comporta como un procedimiento global.

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

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

Clases plantilla (*Template*)

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

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

El uso más frecuente de las plantillas es especificar contenedores que se pueden instanciar para elementos específicos, asegurándose de que sólo contendrá elementos del tipo indicado (*type-safe*). Por ejemplo, el siguiente fragmento de código C++ declara una clase parametrizada `Map`.


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

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

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

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

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

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

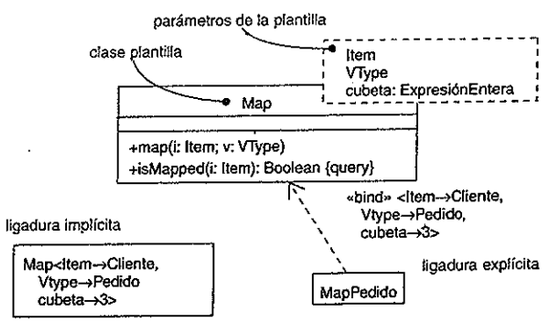


Figura 9.7: Clases plantilla.

Elementos estándar

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

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

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

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

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

Técnicas comunes de modelado

Modelado de la semántica de una clase

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

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

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

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

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

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

Las responsabilidades se discuten en el Capítulo 4.

- Especificar las responsabilidades de la clase. Una responsabilidad es un contrato u obligación de un tipo o clase y se representa en una nota junto a la clase, o en un compartimento extra en el icono de la clase.
- Especificar la semántica de la clase como un todo con texto estructurado, representado en una nota (con estereotipo `semantics`) junto a la clase.

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

- Especificar el cuerpo de cada método usando texto estructurado o un lenguaje de programación, representado en una nota, unida a la operación por una relación de dependencia.

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

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

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

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

Sugerencias y consejos

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

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

Cuando se represente un clasificador en UML:

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

LENGUAJE UNIFICADO DE MODELADO

Capítulo 10

CARACTERÍSTICAS AVANZADAS DE LAS RELACIONES

En este capítulo

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

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

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

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

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

Introducción

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

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

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

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

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

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

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

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

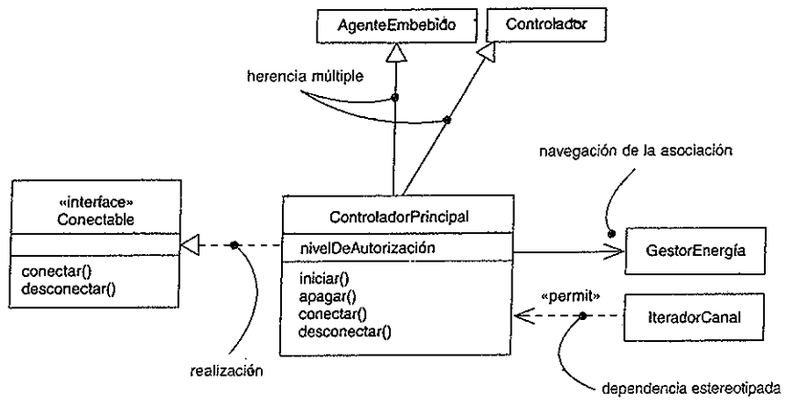


Figura 10.1: Características avanzadas de las relaciones.

Términos y conceptos

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

Dependencia

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

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

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

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

Los diagramas de clases se discuten en el Capítulo 8.

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

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

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

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

2. `derive` — Especifica que el origen puede calcularse a partir del destino.

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

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

3. `permit` — Especifica que el origen tiene una visibilidad especial en el destino.

Las dependencias `permit` se discuten en el Capítulo 5.

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

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

5. `instantiate` — Especifica que el origen crea instancias del destino.

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

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

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

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

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

7. `refine` — Especifica que el origen está a un grado de abstracción más detallado que el destino.

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

8. `use` — Especifica que la semántica del elemento origen depende de la semántica de la parte pública del destino.

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

Los paquetes se discuten en el Capítulo 12.

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

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

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

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

Los casos de uso se discuten en el Capítulo 17.

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

1. `extend` — Especifica que el caso de uso destino extiende el comportamiento del origen.

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

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

Las interacciones se discuten en el Capítulo 16.

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

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

Las máquinas de estados se discuten en el Capítulo 22.

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

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

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

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

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

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

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

Generalización

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

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

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

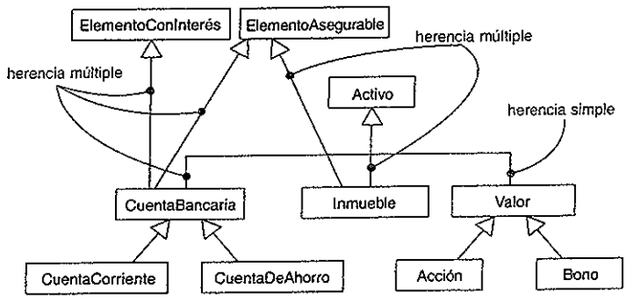


Figura 10.2: Herencia múltiple.

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

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

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

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

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

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

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

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

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

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

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

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

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

Asociación

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

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

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

talles sutiles, como la navegación, la calificación y algunas variantes de la agregación.

Navegación. Dada una asociación simple, sin adornos, entre dos clases, como `Libro` y `Biblioteca`, es posible navegar de los objetos de un tipo a los del otro tipo. A menos que se indique lo contrario, la navegación a través de una asociación es bidireccional. Pero hay ciertas circunstancias en las que se desea limitar la navegación a una sola dirección. Por ejemplo, como se puede ver en la Figura 10.3, al modelar los servicios de un sistema operativo, se encuentra una asociación entre objetos `Usuario` y `Clave`. Dado un `Usuario`, se pueden encontrar las correspondientes `Claves`; pero, dada una `Clave`, no se podrá identificar al `Usuario` correspondiente. Se puede representar de forma explícita la dirección de la navegación con una flecha que apunte en la dirección de recorrido.

Nota: Especificar una dirección de recorrido no significa necesariamente que no se pueda llegar nunca de los objetos de un extremo a los del otro. En vez de ello, la navegación es un enunciado del conocimiento de una clase por parte de otra. Por ejemplo, en la figura anterior, aún sería posible encontrar los objetos `Usuario` asociados con una `Clave` a través de otras asociaciones que involucren a otras clases no mostradas en la figura. Especificar que una asociación es navegable es un enunciado de que, dado un objeto en un extremo, se puede llegar fácil y directamente a los objetos del otro extremo, normalmente debido a que el objeto inicial almacena algunas referencias a ellos.

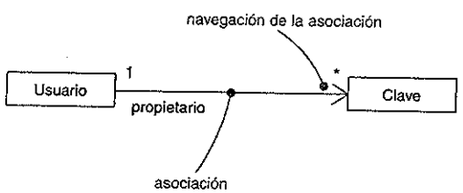


Figura 10.3: Navegación.

Las visibilidades pública, protegida, privada y de paquete se discuten en el Capítulo 9.

Visibilidad. Dada una asociación entre dos clases, los objetos de una clase pueden ver y navegar hasta los objetos de la otra, a menos que se restrinja explícitamente por un enunciado explícito de navegación. Sin embargo, hay circunstancias en las que se quiere limitar la visibilidad a través de esta asociación relativa a los objetos externos a ella. Por ejemplo, como se muestra en la Figura 10.4, hay una asociación entre `GrupoUsuarios` y `Usuario` y otra entre `Usuario` y `Clave`. Dado un objeto `Usuario`, es posible identificar sus correspondientes objetos `Clave`. Sin embargo, una `Clave` es privada a un `Usuario`, así que no debería ser accesible desde el exterior (a menos que el `Usuario` ofrezca acceso explícito a la `Clave`, quizás a través de alguna operación pública). Por lo tanto, como muestra la figura, dado un objeto `GrupoUsuarios`, se puede navegar a sus objetos `Usuario` (y viceversa), pero en cambio no se pueden ver los objetos `Clave` del objeto `Usuario`; son privados al `Usuario`. En UML se pueden especificar tres niveles de visibilidad para el extremo de una asociación, igual que para las características de una clase, adjuntando un símbolo de visibilidad al nombre de un rol. A menos que se indique lo contrario, la visibilidad de un rol es pública. La visibilidad privada indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación; la visibilidad protegida indica que los objetos de ese extremo no son accesibles a ningún objeto externo a la asociación, excepto los hijos del otro extremo. La visibilidad de paquete significa que las clases declaradas en el mismo paquete pueden ver al elemento dado; esto no se aplica a los extremos de asociación.

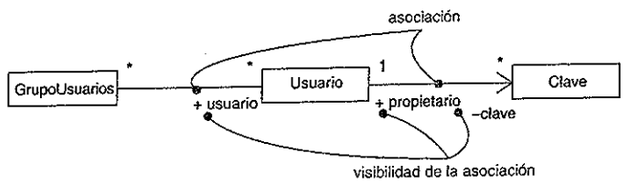


Figura 10.4: Visibilidad.

Los atributos se discuten en los Capítulos 4 y 9.

Calificación. En el contexto de una asociación, uno de los esquemas de modelado más frecuentes tiene que ver con el problema de las búsquedas. Dado un objeto en un extremo de una asociación, ¿cómo identificar un objeto o conjunto de objetos en el otro extremo? Por ejemplo, considérese el problema de modelar una mesa de trabajo en una fábrica, en la cual se arreglan los artículos devueltos. Como se representa en la Figura 10.5, se podría modelar una asociación entre dos clases, `MesaDeTrabajo` y `ArticuloDevuelto`. En el contexto de la `MesaDeTrabajo`, se tendrá un `IdTrabajo` que identificará un `ArticuloDevuelto` particular. En este sentido, `IdTrabajo` es un atributo de la asociación. No es una característica de `ArticuloDevuelto` porque los artículos realmente no tienen conocimiento de cosas como reparaciones o trabajos. Entonces, dado un objeto `MesaDeTrabajo` y dado un `IdTrabajo` particular, se puede navegar hasta cero o un objeto `ArticuloDevuelto`.

En UML se modela este esquema con un calificador, que es un atributo de una asociación cuyos valores identifican un subconjunto de objetos (normalmente un único objeto) relacionados con un objeto a través de una asociación. Un calificador se dibuja como un pequeño rectángulo junto al extremo de la asociación, con los atributos dentro, como se muestra en la figura. El objeto origen, junto a los valores de los atributos del calificador, devuelven un objeto destino (si la multiplicidad del destino es como máximo uno) o un conjunto de objetos (si la multiplicidad del destino es muchos).

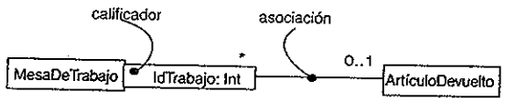


Figura 10.5: Calificación.

La agregación simple se discute en el Capítulo 5.

Composición. La agregación resulta ser un concepto simple con una semántica bastante profunda. La agregación simple es puramente conceptual y no hace más que distinguir un "todo" de una "parte". La agregación simple no cambia el significado de la navegación a través de la asociación entre el todo y sus partes, ni liga las vidas del todo y las partes.

Un atributo es básicamente una forma abreviada para la composición; los atributos se discuten en los Capítulos 4 y 9.

Sin embargo, existe una variación de la agregación simple (la composición) que añade una semántica importante. La composición es una forma de agregación, con una fuerte relación de pertenencia y vidas coincidentes de la parte con el todo. Las partes con una multiplicidad no fijada pueden crearse después de la parte compuesta a la que pertenecen, pero una vez creadas viven y mueren con ella. Tales partes también se pueden eliminar explícitamente antes de la eliminación de la parte compuesta.

Esto significa que, en una agregación compuesta, un objeto puede formar parte de sólo una parte compuesta a la vez. Por ejemplo, en un sistema de ventanas, un `Marco` pertenece exactamente a una `Ventana`. Esto contrasta con la agregación simple, en la que una parte se puede compartir por varios agregados. Por ejemplo, en el modelo de una casa, una `Pared` puede ser parte de uno o más objetos `Habitacion`.

Además, en una agregación compuesta, la parte compuesta es responsable de disponer de las partes, lo que significa que debe gestionar la creación y destrucción de éstas. Por ejemplo, al crear un `Marco` en un sistema de ventanas, debe asignarse a una `Ventana` contenedora. Análogamente, al destruir la `Ventana`, `Ventana` debe a su vez destruir sus partes `Marco`.

Como se muestra en la Figura 10.6, la composición es realmente un tipo especial de asociación, y se especifica adornando una asociación simple con un rombo relleno en el extremo del todo.

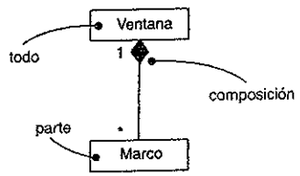


Figura 10.6: Composición.

La estructura interna se discute en el Capítulo 15.

Nota: De forma alternativa, se puede representar la composición anidando los símbolos de las partes dentro del símbolo del compuesto. Esta forma es más útil cuando se quieren destacar las relaciones entre las partes que se aplican sólo en el contexto del todo.

Los atributos se discuten en los Capítulos 4 y 9.

Clases asociación. En una asociación entre dos clases, la propia asociación puede tener propiedades. Por ejemplo, en una relación empleado/patrón entre una `Compañia` y una `Persona`, hay un `Trabajo` que representa las propiedades de esa relación y que se aplican exactamente a un par de `Persona` y `Compañia`. No sería apropiado modelar esta situación con una asociación de `Compañia` a `Trabajo` junto con una asociación de `Trabajo` a `Persona`. Eso no ligaría una instancia específica de `Trabajo` al par específico de `Compañia` y `Persona`.

En UML, esto se modelaría con una clase asociación, la cual es un elemento de modelado con propiedades tanto de asociación como de clase. Una clase asociación puede verse como una asociación que también tiene propiedades de clase, o una clase que también tiene propiedades de asociación. Una clase asociación se dibuja con un símbolo de clase unido por una línea discontinua a una asociación, como se muestra en la Figura 10.8.

Nota: Algunas veces se desea tener las mismas propiedades en varias clases asociación distintas. Sin embargo, una clase asociación no se puede conectar a más de una asociación, ya que una clase asociación es la propia asociación. Para conseguir este propósito, se puede definir una clase (`C`) y luego hacer que cada clase asociación que necesite esas características herede de `C` o utilice a `C` como el tipo de un atributo.

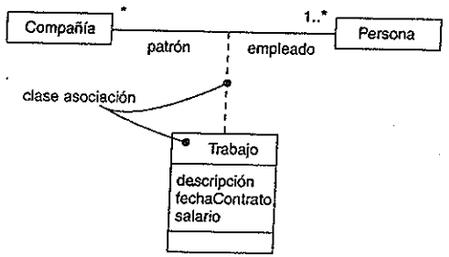


Figura 10.7: Clases asociación.

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

Restricciones. Las propiedades simples y avanzadas de las asociaciones que han sido presentadas hasta ahora son suficientes para la mayoría de las relaciones estructurales que aparecen al modelar. Sin embargo, si se quiere especificar ciertos matices, UML define cinco restricciones que pueden aplicarse a las asociaciones.

En primer lugar, se puede especificar si los objetos de un extremo de la asociación (con una multiplicidad mayor que uno) están ordenados o no.

1. `ordered` — Especifica que el conjunto de objetos en un extremo de una asociación sigue un orden explícito.

Por ejemplo, en una asociación `Usuario/Clave`, las `Claves` asociadas con el `Usuario` podrían mantenerse en orden de menos a más recientemente usada, y se marcarían como `ordered`. Si no existe la palabra clave, los objetos no están ordenados.

Estas propiedades de mutabilidad también se aplican a los atributos, como se discute en el Capítulo 9; los enlaces se discuten en el Capítulo 16.

En segundo lugar, se puede especificar que los objetos en un extremo de la asociación son únicos (es decir, forman un conjunto) o no lo son (es decir, forman una bolsa).

2. `set` — Objetos únicos, sin duplicados.
3. `bag` — Objetos no únicos; puede haber duplicados.
4. `ordered set` — Objetos únicos, pero ordenados.
5. `list` o `sequence` — Objetos ordenados; puede haber duplicados.

Por último, hay una restricción que restringe los cambios posibles de las instancias de una asociación.

6. `readonly` — Un enlace, una vez añadido desde un objeto del otro extremo de la asociación, no se puede modificar ni eliminar. El valor por defecto, en ausencia de esta restricción, es permitir los cambios.

Nota: Para ser precisos, `ordered` y `readonly` son propiedades de un extremo de una asociación. No obstante, se representan con la notación de las restricciones.

Realización

Una *realización* es una relación semántica entre clasificadores, en la cual un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Gráficamente, una realización se representa como una línea dirigida discontinua, con una gran flecha hueca que apunta al clasificador que especifica el contrato.

La realización es lo suficientemente diferente de la dependencia, la generalización y la asociación para ser tratada como un tipo diferente de relación. Semánticamente, la realización es algo así como una mezcla entre dependencia y generalización, y su notación es una combinación de la notación para la dependencia y la generalización. La realización se utiliza en dos circunstancias: en el contexto de las interfaces y en el contexto de las colaboraciones.

Las interfaces se discuten en el Capítulo 11; las clases se discuten en los Capítulos 4 y 9; los componentes se discuten en el Capítulo 15; las cinco vistas de una arquitectura se discuten en el Capítulo 2.

La mayoría de las veces la realización se empleará para especificar la relación entre una interfaz y la clase o el componente que proporciona una operación o servicio para ella. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o componente. Por consiguiente, una interfaz especifica un contrato que debe llevar a cabo una clase o un componente. Una interfaz puede ser realizada por muchas clases o componentes, y una clase o un componente pueden realizar muchas interfaces. Quizás lo más interesante de las interfaces sea que permiten separar la especificación de un contrato (la propia interfaz) de su implementación (por una clase o un componente). Además, las interfaces incluyen las partes lógicas y físicas de la arquitectura de un sistema. Por ejemplo, como se muestra en la Figura 10.8, una clase (como `ReglasNegocioCuenta` en un sistema de entrada de pedidos) en una vista de diseño de un sistema podría realizar una interfaz dada (como `IAgenteDeReglas`). Esta misma interfaz (`IAgenteDeReglas`) también podría ser reali-

zada por un componente (como `reglacuenta.dll`) en la vista de implementación del sistema. Como muestra la figura, una realización se puede representar de dos maneras: de la forma canónica (con el estereotipo `interface` y la línea dirigida discontinua con la flecha hueca) y de la forma abreviada (con la notación en forma de piruleta).

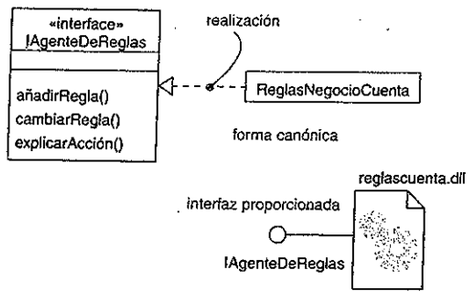


Figura 10.8: Realización de una interfaz.

Los casos de uso se discuten en el Capítulo 17; las colaboraciones se discuten en el Capítulo 28.

Como se muestra en la Figura 10.9, también se utiliza la realización para especificar la relación entre un caso de uso y la colaboración que realiza ese caso de uso. En esta circunstancia, casi siempre se utiliza la forma canónica de la realización.

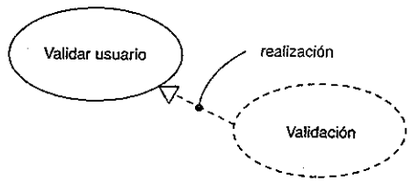


Figura 10.9: Realización de un caso de uso.

Nota: El que una clase o un componente realice una interfaz significa que los clientes pueden confiar en que la clase o el componente llevan a cabo fielmente el comportamiento especificado en la interfaz. Esto quiere decir que la clase o el componente implementan todas las operaciones de la interfaz, responden a todas sus señales, y siempre siguen el protocolo establecido por la interfaz para los clientes que utilizan esas operaciones o envían esas señales.

Técnicas comunes de modelado

Modelado de redes de relaciones

El modelado del vocabulario de un sistema y el modelado de la distribución de responsabilidades en un sistema se discuten en el Capítulo 4.

Al modelar el vocabulario de un sistema aparecen docenas, si no cientos o miles, de clases, interfaces, componentes, nodos y casos de uso. Es difícil establecer una frontera bien definida alrededor de cada una de estas abstracciones. Establecer la miríada de relaciones entre estas abstracciones es aún más difícil: requiere hacer una distribución equilibrada de responsabilidades en el sistema global, con abstracciones individuales muy cohesivas y relaciones expresivas, pero todo débilmente acoplado.

Cuando se modelen estas redes de relaciones:

Los casos de uso se discuten en el Capítulo 17.

- No hay que comenzar de forma aislada. Deben aplicarse los casos de uso y los escenarios para guiar el descubrimiento de las relaciones entre un conjunto de abstracciones.
- En general, hay que comenzar modelando las relaciones estructurales que estén presentes. Éstas reflejan la vista estática del sistema y por ello son bastante tangibles.
- A continuación hay que identificar las posibles relaciones de generalización/especialización; debe usarse la herencia múltiple de forma moderada.
- Sólo después de completar los pasos precedentes se deberán buscar dependencias; normalmente representan formas más sutiles de conexión semántica.
- Para cada tipo de relación hay que comenzar con su forma básica y aplicar las características avanzadas sólo cuando sean absolutamente necesarias para expresar la intención.

Las cinco vistas de una arquitectura se discuten en el Capítulo 2; el Proceso Unificado de Rational se resume en el Apéndice B.

- Hay que recordar que no es deseable ni necesario modelar en un único diagrama o vista todas las relaciones entre un conjunto de abstracciones. En vez de ello, debe extenderse gradualmente el conjunto de relaciones del sistema, considerando diferentes vistas de éste. Hay que resaltar los conjuntos interesantes de relaciones en diagramas individuales.

La clave para tener éxito al modelar redes complejas de relaciones es hacerlo de forma incremental. Hay que incrementar gradualmente las relaciones conforme se añaden a la estructura de la arquitectura de un sistema. Hay que simplificar

esas relaciones conforme se descubren ocasiones de aplicar mecanismos comunes. En cada versión del proceso de desarrollo, se deben revisar las relaciones entre las abstracciones clave del sistema.

Nota: En la práctica (y especialmente si se sigue un proceso de desarrollo incremental e iterativo), las relaciones de los modelos derivarán de decisiones explícitas del modelador, así como de la ingeniería inversa de la implementación.

Sugerencias y consejos

Al modelar relaciones avanzadas en UML, conviene recordar que hay disponible un amplio rango de bloques de construcción, desde asociaciones simples hasta propiedades más detalladas de navegación, calificación, agregación, etc. Se debe elegir la relación y los detalles de la relación que mejor encajen con una abstracción dada. Una relación bien estructurada:

- Sólo muestra las características necesarias para que los clientes usen la relación y oculta las demás.
- No es ambigua en su objetivo ni en su semántica.
- No está tan sobreespecificada que elimine cualquier grado de libertad de los implementadores.
- No está tan poco especificada que quede ambiguo el significado de la relación.

Cuando se dibuje una relación en UML:

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

Capítulo 11

INTERFACES, TIPOS Y ROLES

En este capítulo

- Interfaces, tipos, roles y realización.
- Modelado de las líneas de separación en un sistema.
- Modelado de tipos estáticos y dinámicos.
- Obtener interfaces comprensibles y accesibles.

Las interfaces definen una línea entre la especificación de lo que una abstracción hace y la implementación de cómo lo hace. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o de un componente.

Las interfaces se utilizan para visualizar, especificar, construir y documentar las líneas de separación dentro de un sistema. Los tipos y los roles proporcionan mecanismos para modelar la conformidad estática y dinámica de una abstracción con una interfaz en un contexto específico.

Una interfaz bien estructurada proporciona una clara separación entre las vistas externa e interna de una abstracción, haciendo posible comprender y abordar una abstracción sin tener que sumergirse en los detalles de su implementación.

Introducción

El diseño de casas se discute en el Capítulo 1.

No tendría sentido construir una casa donde hubiese que romper los cimientos cada vez que hubiese que pintar las paredes. Asimismo, nadie desearía vivir en un sitio donde hubiese que reinstalar los cables cada vez que se cambiara una lámpara. Al propietario de un gran edificio no le haría mucha gracia tener que