

---

## 9.1. Presentación del capítulo

---

Este capítulo trata las relaciones entre objetos y las relaciones entre clases. El capítulo se organiza bajo tres enfoques diferentes. Tratamos los vínculos (relaciones entre objetos), las asociaciones (relaciones entre clases) y, por último, las dependencias (relaciones generales).

## 9.2. ¿Qué es una relación?

---

Las relaciones son conexiones semánticas (significativas) entre elementos de modelado; son la forma de UML de conectar elementos juntos. Ya ha visto algunos tipos de relaciones:

- Entre actores y casos de uso (asociación).
- Entre casos de uso y casos de uso (generalización, <<include>>, <<extend>>).
- Entre actores y actores (generalización).

En este capítulo exploramos conexiones entre objetos y conexiones entre clases. Empezamos con vínculos y asociaciones y, luego, en el capítulo 10, examinamos la generalización y la herencia.

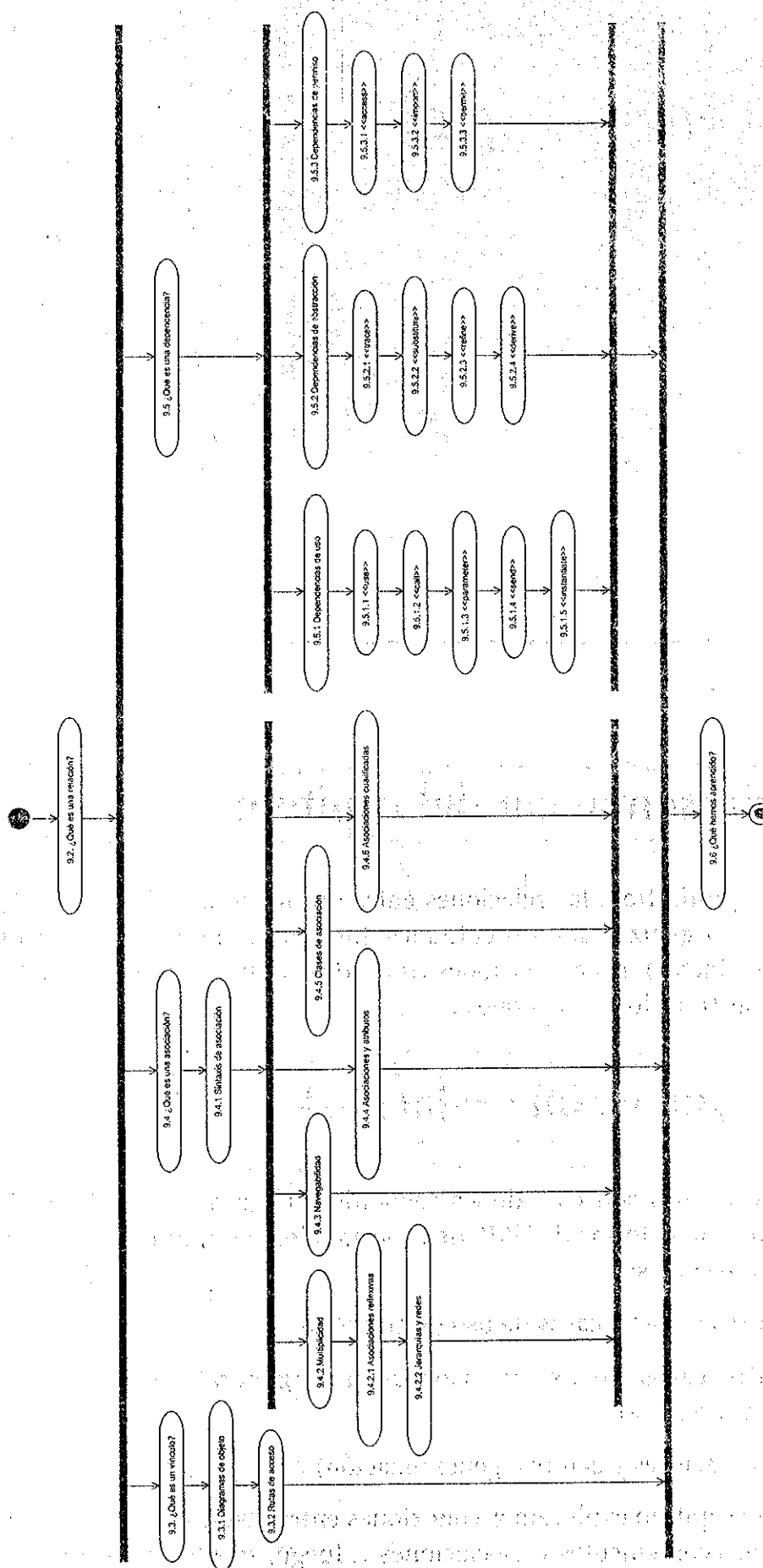


Figura 9.1.

Para crear un sistema orientado a objetos que funcione, no puede dejar que los objetos permanezcan solos, aislados. Necesita conectarlos para que puedan realizar un trabajo de utilidad para beneficiar a los usuarios del sistema. Las conexiones entre objetos se denominan vínculos, y cuando los objetos funcionan conjuntamente se dicen que colaboran.

Si existe un vínculo entre dos objetos, debe existir alguna conexión semántica entre sus clases. Esto es sentido común; para que los objetos se comuniquen directamente entre sí, las clases de esos objetos deben saber del resto de alguna forma. Las conexiones entre clases se conocen como asociaciones. Los vínculos entre objetos son instancias de las asociaciones entre sus clases.

### 9.3. ¿Qué es un vínculo?

Para crear un programa orientado a objetos, los objetos se tienen que comunicar entre sí. De hecho, un programa orientado a objetos ejecutable es una comunidad armoniosa de objetos que cooperan.

Un vínculo es una conexión semántica entre dos objetos que permite enviar mensajes de un objeto a otro. Un sistema orientado a objetos ejecutable contiene muchos objetos que van y vienen y muchos vínculos (que también van y vienen) que unen esos objetos. Los mensajes se pasan entre los objetos por estos vínculos. Al recibir un mensaje, un objeto invocará su operación correspondiente.

Los vínculos se implementan de formas diferentes por diferentes lenguajes orientados a objetos. Java implementa los vínculos como referencias de objeto; C++ puede implementar los vínculos como referencias o por una inclusión directa de un objeto por otro.

Cualquiera que sea el enfoque, un requisito mínimo para un vínculo es que al menos uno de los objetos tiene que tener una referencia de objeto al otro.

#### 9.3.1. Diagramas de objeto

Un diagrama de objeto es un diagrama que muestra objetos y sus relaciones en un punto en el tiempo. Es como una instantánea de parte de un sistema ejecutable orientado a objetos en un instante determinado, mostrando los objetos y vínculos entre ellos.

Los objetos que están conectados por vínculos pueden adoptar varios roles relativos entre ellos. En la figura 9.2 puede ver que el objeto `ila` adopta el rol de director en su vínculo con el objeto `clubLibro`. Indica esto en el diagrama del objeto al situar el nombre del rol en el extremo apropiado del vínculo. Puede situar nombres de rol en uno o ambos extremos de un vínculo. En este caso, el objeto `clubLibro` siempre desempeña el rol de "club" por lo que no hay necesidad de mostrarlo en el diagrama; no añadirá nada a nuestro entendimiento de las relaciones de objetos.

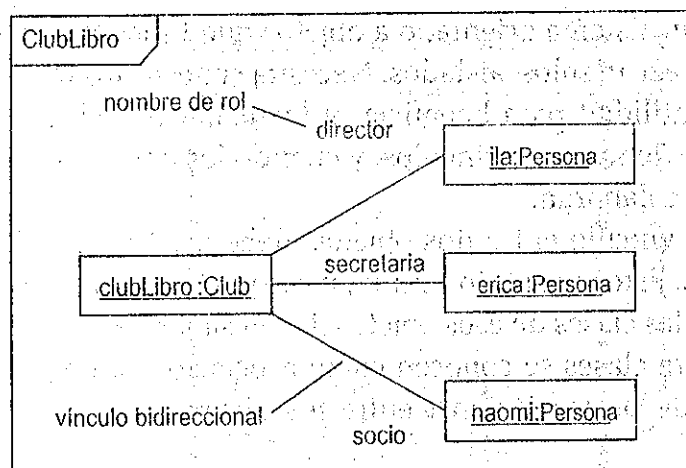


Figura 9.2.

La figura 9.2 nos dice que en un punto determinado en el tiempo, el objeto `ila` desempeña el rol de `director`. Sin embargo, es importante darse cuenta que los vínculos son conexiones dinámicas entre objetos. Es decir, no están necesariamente fijas en el tiempo. En este ejemplo, el rol `director` puede pasar en algún momento a `erika` o `Naomi` y podríamos crear un diagrama de objeto para mostrar este nuevo estado. Normalmente, un solo vínculo conecta exactamente dos objetos como se muestra en la figura 9.2. Sin embargo, UML permite que un solo vínculo conecte más de dos objetos. Esto se conoce como un vínculo *n-ario* y se muestra con una rúta a cada objeto participante. Muchos modeladores (nosotros incluidos) consideran esto innecesario. Raramente se utiliza y las herramientas de modelado UML no siempre lo soportan, por lo que no decimos nada más de esto aquí. En la figura 9.2 en más detalle, puede ver que existen tres vínculos entre cuatro objetos:

- Un vínculo entre `clubLibro` e `ila`.
- Un vínculo entre `clubLibro` y `erika`.
- Un vínculo entre `clubLibro` y `naomi`.

En la figura 9.2 los vínculos son bidireccionales, por lo tanto puede decir correctamente que el vínculo conecta `ila` a `clubLibro` o que el vínculo conecta `clubLibro` a `ila`. Si un vínculo es unidireccional, utiliza la navegabilidad para especificar en qué dirección los mensajes pueden pasar por el vínculo.

Puede mostrar la navegabilidad al situar una punta de flecha (navegable) o cruz (no navegable) en el extremo de un vínculo. Piense en navegabilidad como un sistema de un solo sentido en una ciudad. Los mensajes solamente pueden fluir en la dirección indicada por la punta de flecha. La especificación UML 2 permite tres formas diferentes de modelado para mostrar la navegabilidad, que tratamos en detalle más adelante. Utilizamos la siguiente especificación a lo largo del libro:

- Todas las cruces se suprimen.
- Las asociaciones bidireccionales no tienen flechas.
- Las asociaciones unidireccionales tienen una sola flecha.

La única desventaja real de esto es que no hay forma de indicar que la navegabilidad está sin decidir ya que no navegabilidad significa "no navegable".

Por ejemplo, la figura 9.3 muestra que el vínculo entre `:DetallesPersona` y `:Dirección` es unidireccional. Esto significa que el objeto `:DetallesPersona` tiene una referencia de objeto con el objeto `:Dirección`, pero no viceversa. Los mensajes solamente se pueden enviar de `:DetallesPersona` a `:Dirección`.

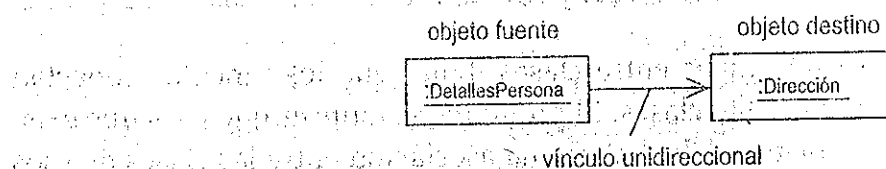


Figura 9.3.

### 9.3.2. Rutas de acceso

Los símbolos UML, como el icono de objeto, el icono de caso de uso y el icono de clase, están conectados con otros símbolos por rutas de acceso. Una ruta de acceso es "una serie conectada de segmentos gráficos" (en otras palabras, una línea) que une dos o más símbolos. Existen tres estilos para dibujar rutas de acceso:

- **Ortogonal:** Donde la ruta consta de una serie de segmentos horizontales y verticales.
- **Oblicuo:** Donde la ruta es una serie de una o más líneas inclinadas.
- **Curvado:** Donde la ruta es una curva.

Es una cuestión de preferencia personal decidir el estilo que se utiliza, y los estilos se pueden incluso mezclar en el mismo diagrama, si esto hace que el diagrama sea más claro y fácil de leer. Normalmente utilizamos el estilo ortogonal como muchos otros modeladores. En la figura 9.4, hemos adoptado el estilo ortogonal y las rutas se han combinado en un árbol. Solamente puede combinar rutas que tienen las mismas propiedades. En este caso, todas las rutas representan vínculos y por lo tanto podemos combinarlas.

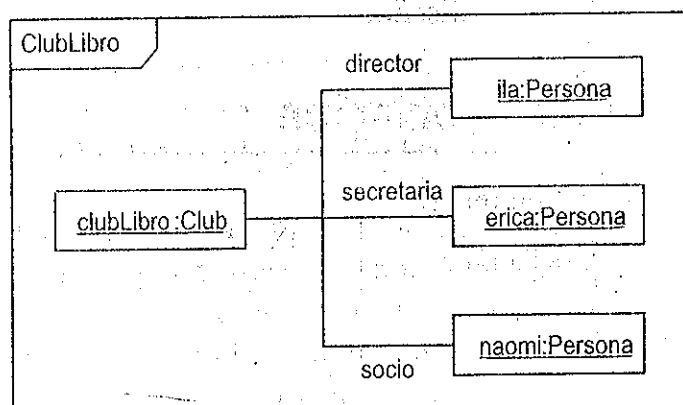


Figura 9.4.

La legibilidad visual y el interés general de los diagramas es de crucial importancia. Recuerde siempre que la mayoría de los diagramas se dibujan para ser leídos por alguien. Como tal, no importa el estilo que adopte, la claridad es vital.

## 9.4. ¿Qué es una asociación?

Las asociaciones son relaciones entre clases. Igual que los vínculos conectan objetos, las asociaciones conectan clases. El punto importante es que para que exista un vínculo entre dos objetos, debe haber una asociación entre las clases de esos objetos. Esto es porque un vínculo es una instancia de una asociación, igual que un objeto es una instancia de una clase. La figura 9.5 muestra la relación entre clases y objetos, y entre vínculos y asociaciones. Puesto que no puede tener un vínculo sin una asociación, está claro que los vínculos dependen de las asociaciones; puede modelar esto con una relación de dependencia (la flecha de puntos) que veremos con más detalle más adelante. Para hacer explícita la semántica de la dependencia entre asociaciones y vínculos, estereotipe la dependencia `<<instantiate>>`.

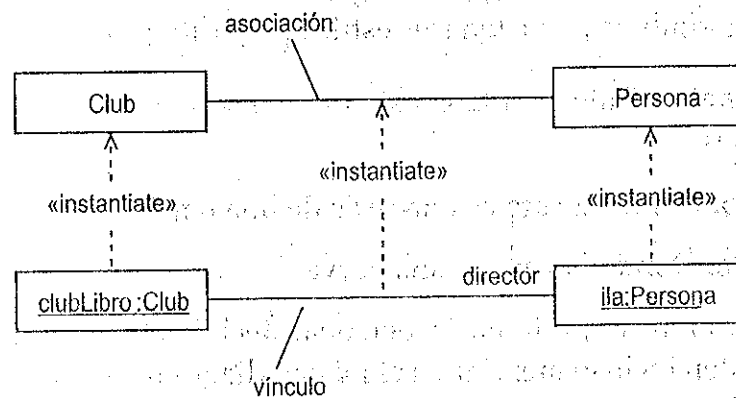


Figura 9.5.

La semántica de la asociación básica es muy sencilla, una asociación entre clases indica que puede tener vínculos entre objetos de esas clases. Existen otras formas más mejoradas de asociación (agregación y composición) que examinaremos en el capítulo 18 en el workflow de diseño.

### 9.4.1. Sintaxis de asociación

Las asociaciones pueden tener:

- Un nombre de asociación.
- Nombres de roles.
- Multiplicidad.
- Navegabilidad.

Los nombres de asociación deberían ser frases verbales porque indican una acción que el objeto fuente está realizando sobre el objeto destino. El nombre también se podría prefijar o postfixar con una pequeña punta de flecha en negro para indicar la dirección en la que se debería leer el nombre de asociación. Los nombres de asociación se escriben poniendo en mayúscula la primera letra de cada palabra con la primera letra de todas en minúscula.

En el ejemplo de la figura 9.6 lee la asociación de la siguiente manera: "una Empresa emplea muchas Personas". Aunque la flecha indica la dirección en la que se debería leer la asociación, siempre puede leer las asociaciones también en la otra dirección. Por lo tanto, en la figura 9.6 puede decir "cada Persona está empleada exactamente por una Empresa" en cualquier momento en el tiempo.

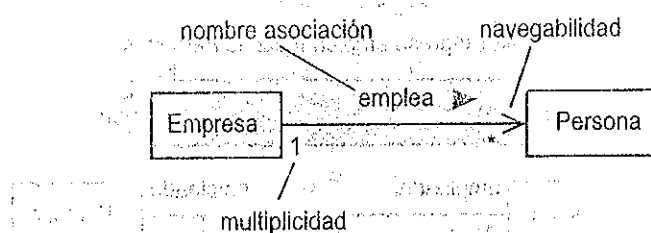


Figura 9.6.

De forma alternativa, puede asignar nombres de rol a las clases en uno o ambos extremos de la asociación. Estos nombres de roles indican los roles que los objetos de esas clases desempeñan cuando están vinculados por instancias de esta asociación. En la figura 9.7 puede ver que un objeto Empresa desempeñará el rol empleador y objetos Persona desempeñará el rol empleado cuando estén vinculados por instancias de esta asociación. Los nombres de roles deberían ser nombres o frases nominales ya que nombran un rol que pueden desempeñar objetos.

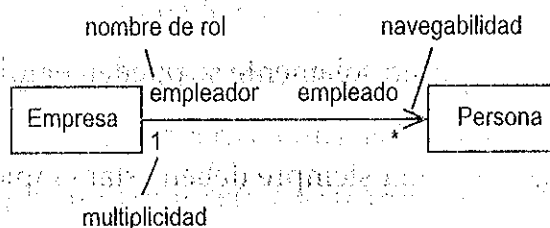


Figura 9.7.

Las asociaciones pueden tener un nombre de asociación o nombres de rol. Situar ambos nombres de rol y nombres de asociación en la misma asociación es teóricamente legal, pero es un mal estilo y peligroso. La clave para buenos nombre de asociación y nombres de roles es que se deberían leer bien.

En la figura 9.6 una Empresa emplea muchas Personas, se lee muy bien. Leer la asociación al contrario puede decir que una Persona está empleada exactamente por una Empresa en cualquier momento en el tiempo; se sigue también leyendo bien. De forma similar, los nombres de rol en la figura 9.7 indican claramente los roles que los objetos de estas clases desempeñarán cuando se vinculan de esta forma en particular.

### 9.4.2. Multiplicidad

Las restricciones son uno de los tres mecanismos de extensibilidad de UML y multiplicidad es el primer tipo de restricción que hemos visto. También es el tipo de restricción más común. La multiplicidad restringe el número de objetos de una clase que se pueden implicar en una relación determinada en cualquier momento en el tiempo. La frase "en cualquier momento en el tiempo" es vital para entender las multiplicidades. Considerando la figura 9.8, puede ver que en cualquier momento en el tiempo un objeto Persona está empleado por exactamente un objeto Empresa. Sin embargo, con el tiempo un objeto Persona podría estar empleado por una serie de objetos Empresa.

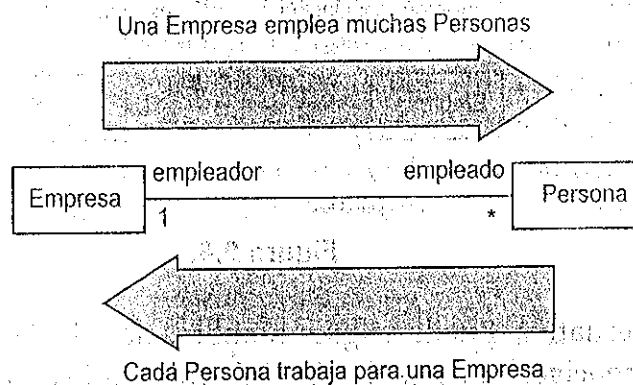


Figura 9.8.

Examinando la figura 9.8, puede ver algo más que es interesante. Un objeto Persona nunca puede estar desempleado; siempre está empleado por exactamente un objeto Empresa. La restricción por lo tanto incorpora dos reglas de negocio de este modelo:

- Que los objetos Persona solamente se pueden emplear por una Empresa de cada vez.
- Que los objetos Persona siempre deben estar empleados.

Que éstas sean restricciones razonables o no depende por completo de los requisitos del sistema que esté modelando, pero esto es lo que dice el modelo. Puede ver que las limitaciones de multiplicidad son muy importantes; pueden codificar reglas clave de negocio en su modelo. Sin embargo, estas reglas están "enterradas" en los detalles del modelo. Algunos modeladores llaman a esta ocultación de reglas y requisitos de negocio clave "trivialización". Para una explicación mucho más detallada de este fenómeno, véase [Arlow 1]. La multiplicidad se especifica como una lista de intervalos separados por coma, donde cada intervalo es de la forma:

mínimo..máximo

mínimo y máximo pueden ser enteros o cualquier expresión que tenga un resultado entero.



Si la multiplicidad no se indica explícitamente, está pendiente, no existe multiplicidad "predeterminada" en UML. De hecho, es un error común de modelado UML asumir que una multiplicidad sin decidir por defecto indica una multiplicidad de 1. Algunos ejemplos de sintaxis de multiplicidad se facilitan en la tabla 9.1.

Tabla 9.1.

Adorno	Semántica
0..1	Cero ó 1
1	Exactamente 1
0..*	Cero ó más
*	Cero ó más
1..*	1 ó más
1..6	1 a 6
1..3, 7..10, 15, 19..*	1 a 3 ó 7 a 10 ó 15 exactamente, ó 19 a muchos

El ejemplo en la figura 9.9 ilustra que la multiplicidad es una potente restricción que a veces tiene un gran efecto sobre la semántica de negocio del modelo. Si lee el ejemplo detenidamente ve que:

- Una Empresa puede tener exactamente siete empleados.
- Una Persona puede estar empleada por exactamente una Empresa (en este modelo una Persona no puede tener más de un trabajo a la vez).
- Una CuentaBancaria puede tener exactamente un propietario.
- Una CuentaBancaria puede tener uno o muchos operadores.
- Una Persona puede tener cero a muchas CuentasBancarias.
- Una Persona puede operar cero a muchas CuentasBancarias.

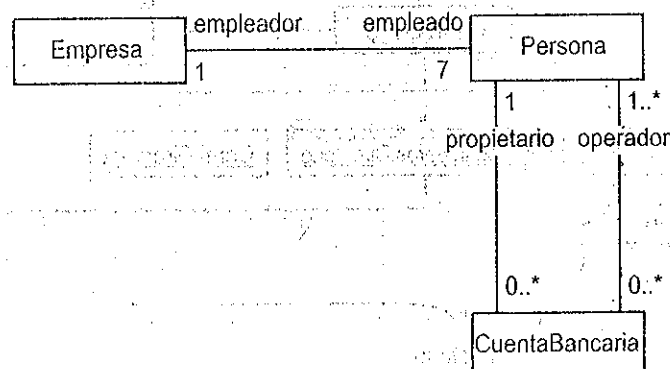


Figura 9.9.

Cuando se lee un modelo UML, es muy importante averiguar exactamente qué dice el modelo en lugar de realizar cualquier asunción o inventar la semántica. A esto lo llamamos "leer el modelo literalmente".

Por ejemplo, la figura 9.9 indica que una Empresa puede tener exactamente 7 empleados, ni más y ni menos. La mayoría de las personas considerarían que esta semántica es bastante extraña o incluso incorrecta (a menos que sea una empresa muy extraña), pero esto es lo que dice el modelo. Nunca debe olvidar esto.

Existe mucho debate acerca de si la multiplicidad se debería mostrar en modelos de análisis. Pensamos que se debería hacer porque la multiplicidad describe reglas de negocio, requisitos y restricciones y puede exponer suposiciones injustificadas realizadas sobre el negocio. Dichas asunciones se deben exponer tan pronto como sea posible.

#### 9.4.2.1. Asociaciones reflexivas

Es bastante común para una clase tener una asociación consigo misma. Esto se denomina una asociación reflexiva y significa que los objetos de esa clase tienen vínculos a otros objetos de la misma clase. Un buen ejemplo de una asociación reflexiva se muestra en la figura 9.10. Todo objeto Directorio puede tener vínculos a cero ó más objetos Directorio que desempeñan el rol subdirectorio y a cero ó un objeto Directorio que desempeña el rol padre. Además, todo objeto Directorio está asociado con cero ó más objetos Archivo. Esto modela una estructura de directorio genérica bastante bien, aunque merece la pena mencionar que sistemas específicos de archivo (como Windows) pueden tener diferentes restricciones de multiplicidad para este modelo.

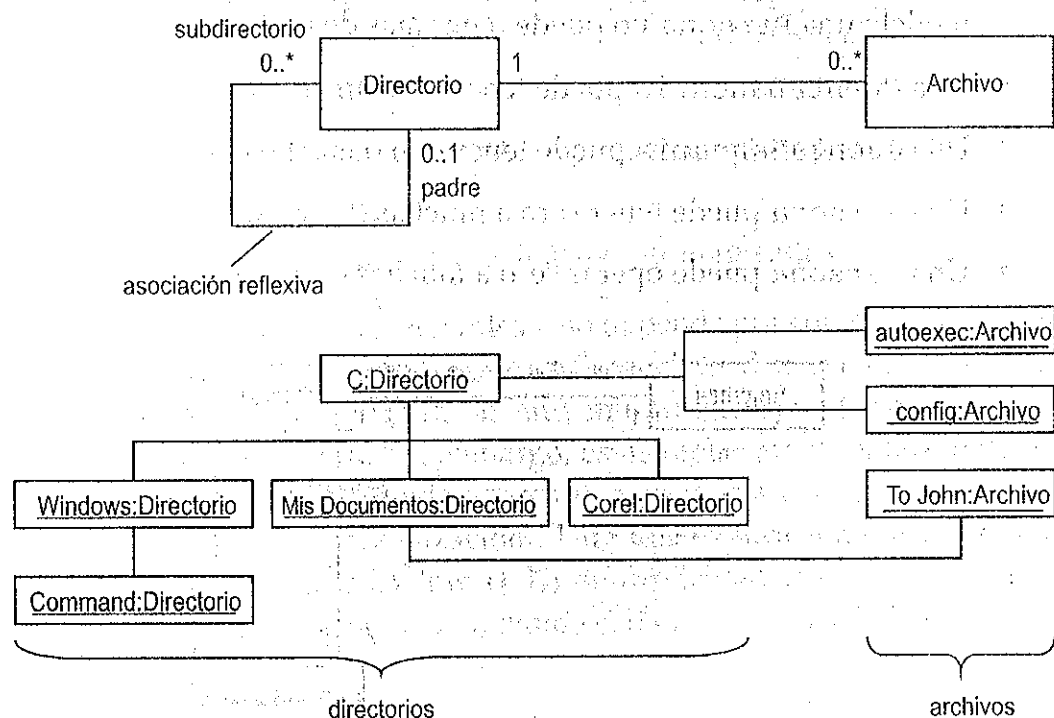


Figura 9.10.

La mitad superior de la figura 9.10 muestra el diagrama de clase y la mitad inferior muestra un diagrama de objeto de ejemplo que concuerda con ese diagrama de clase.

### 9.4.2.2. Jerarquías y redes

Cuando se modela, encontrará que los objetos se organizan en jerarquías o redes. Una jerarquía tiene un objeto raíz y cualquier otro nodo en la jerarquía tiene exactamente un objeto directamente sobre él. Los árboles de directorio forman jerarquías. Lo mismo lo hacen elementos en XML y documentos HTML. La jerarquía es una forma ordenada y estructura y en cierto sentido rígida de organizar objetos. Un ejemplo se muestra en la figura 9.11.

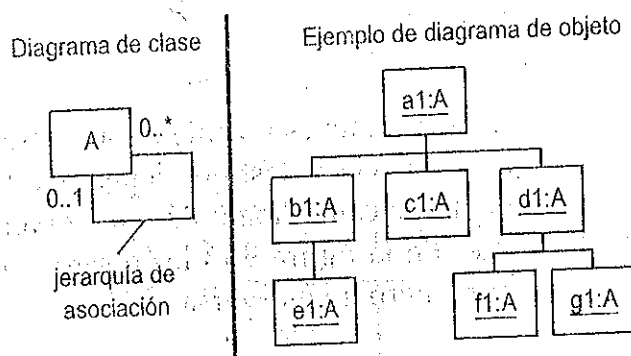


Figura 9.11.

Sin embargo, en la red no hay objetos raíz aunque esto no está excluido. En las redes, cada objeto puede tener muchos objetos directamente conectados a él. No existe un concepto real de "arriba" o "abajo" en una red. Es una estructura mucho más flexible en la que es posible que ningún nodo tenga primacía sobre otro. La World Wide Web forma una red compleja de nodos, como se ilustra de forma sencilla en la figura 9.12.

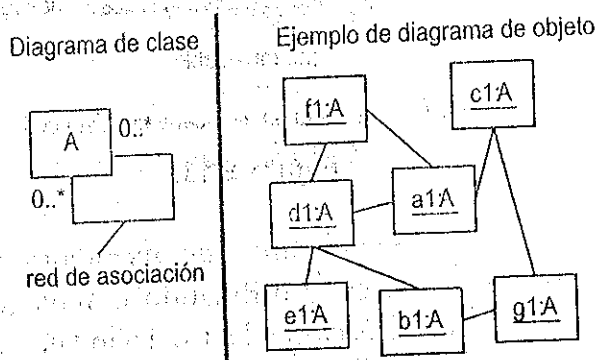


Figura 9.12.

Como ejemplo para ilustrar jerarquías y redes, consideremos productos. Existen dos abstracciones fundamentales:

- TipoProducto, un tipo de producto como una impresora de tinta,
- ElementoProducto, una impresora específica, con número de serie 0001123430.

TipoProducto y ElementoProducto se tratan en detalle en [Arlow 1]. Los TipoProducto a menudo tienden a formar redes, por lo que un TipoProducto como un ordenador puede constar de una CPU, pantalla, teclado, ratón, tarjeta gráfica y otros TiposProducto. Cada uno de estos TiposProducto describe un tipo de producto, no un elemento individual, y estos tipos de producto pueden participar en otros TiposProducto como diferentes paquetes de ordenador.

Sin embargo, si consideramos los ElementosProducto, que son instancias específicas de un TipoProducto, cualquier ElementoProducto, como una CPU específica, solamente se puede vender y distribuir una vez como parte de un paquete de artículos. Los ElementosProducto forman jerarquías.

### 9.4.3. Navegabilidad

La navegabilidad nos muestra que es posible pasar desde un objeto de la clase fuente a uno o más objetos de la clase destino, dependiendo de la multiplicidad. Puede pensar en la navegabilidad como "mensajes que solamente se pueden enviar en la dirección de la flecha". En la figura 9.13 los objetos Pedido pueden enviar mensajes a objetos Producto pero no viceversa.

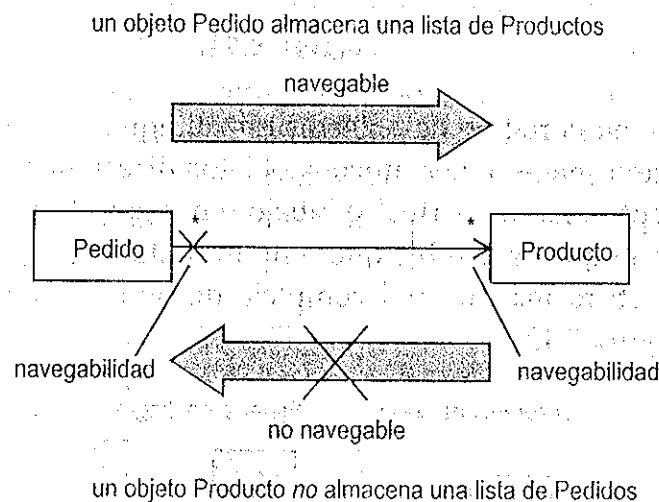


Figura 9.13.

Uno de los objetivos de un buen análisis de diseño orientado a objetos es minimizar el acoplamiento entre clases, y utilizando la navegabilidad, es una buena forma de hacerlo. Al hacer unidireccional la asociación entre Pedido y Producto, puede navegar fácilmente desde objetos Pedido a objetos Producto, pero no hay navegabilidad inversa desde objetos Producto a objetos Pedido. Por lo tanto, los objetos Producto no saben que pueden estar participando en un Pedido en particular y, por lo tanto, no tienen acoplamiento con Pedido.

La navegabilidad se muestra al anexas una cruz o una punta de flecha en un extremo de la relación como se muestra en la figura 9.13.

La especificación UML 2.0 [UML2S] sugiere tres modos de modelado para la utilización de la navegabilidad en sus diagramas:

1. Haga que la navegabilidad sea completamente explícita. Se deben mostrar todas las flechas y cruces.
2. Haga que la navegabilidad sea completamente invisible. No se muestra ninguna flecha o cruz.
3. Suprima todas las cruces. Las asociaciones bidireccionales no tienen flechas. Las asociaciones unidireccionales tienen una sola flecha.

Estos tres modelos se resumen en la figura 9.14.


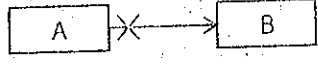
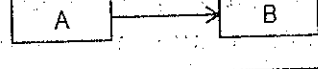
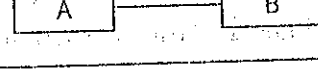
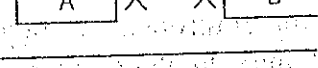
Modelos de navegabilidad UML 2			
Sintaxis UML 2	Modelo 1: Navegabilidad UML2 estricta	Modelo 2: No navegabilidad	Modelo 3: Práctica estándar
	A a B es navegable B a A es navegable		
	A a B es navegable B a A no es navegable		
	A a B es navegable B a A no está definido		A a B es navegable B a A no es navegable
	A a B no está definido B a A no está definido	A a B no está definido B a A no está definido	A a B es navegable B a A es navegable
	A a B no es navegable B a A no es navegable		

Figura 9.14.

El modelo 1 hace que la navegabilidad sea completamente visible, pero puede tender a saturar los diagramas.

El modelo 2 se debería evitar porque oculta demasiada información valiosa.

El modelo 3 es un compromiso razonable. De hecho, el modelo 3 es la opción que se utiliza casi de forma exclusiva en la práctica. Puesto que representa las mejores prácticas, es la opción que utilizamos en este libro. Las principales ventajas de este modelo es que no satura los diagramas con demasiadas flechas o cruces y tiene compatibilidad inversa con versiones anteriores de UML. Sin embargo, presenta desventajas:

- No es posible decir a partir de un diagrama si la navegabilidad está presente o si todavía no se ha definido.
- Cambia el significado de la cabeza de flecha de navegable/indefinido a navegable/no navegable. Es bastante desafortunado, pero así es.

- No puede mostrar asociaciones que no son navegables en ninguna dirección (una cruz en cada extremo). Carecen de utilidad en el modelado diario por lo que no es ningún problema.

Puede ver un resumen de estos tres modelos en la figura 9.15.

Visibilidad modo 3 se utiliza casi de forma exclusiva en la práctica

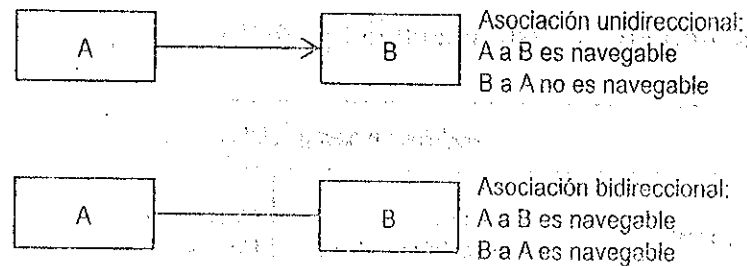


Figura 9.15.

Incluso si una asociación no es navegable en una determinada dirección, puede ser posible atravesar la relación en esa dirección. Sin embargo, el coste computacional será muy alto. En el ejemplo en la figura 9.13, aunque no puede navegar de forma inversa desde Producto a Pedido, puede encontrar el objeto Pedido asociado con un determinado objeto Producto al buscar por todos los objetos Pedido en turno. Ha atravesado entonces una relación no navegable, pero a un coste computacional alto. La navegabilidad en un sentido es como una calle de un solo sentido; podría llegar al final de ésta por medio alguna otra ruta (más larga).

Si existe un nombre de rol en el extremo destino de la relación, los objetos de la clase origen pueden hacer referencia a objetos de la clase destino al utilizar este nombre de rol.

En términos de implementación en lenguajes orientados a objetos, la navegabilidad implica que el objeto origen contiene una referencia de objeto al objeto destino. El objeto fuente puede utilizar esta referencia de objeto para enviar mensajes al objeto destino. Podría representar eso en un diagrama de objeto como un vínculo unidireccional con mensaje asociado.

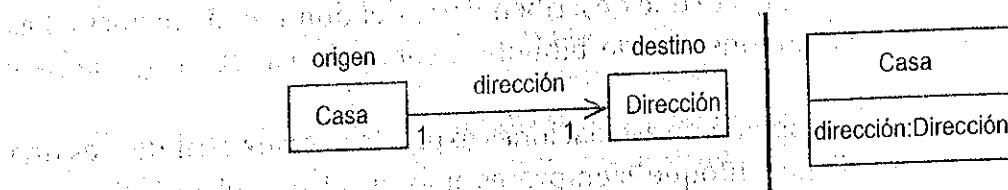
#### 9.4.4. Asociaciones y atributos

Existe un vínculo muy próximo entre asociaciones de clase y atributos de clase.

Una asociación entre una clase origen y una clase destino significa que los objetos de una clase origen pueden albergar una referencia de objeto a objetos de la clase destino. Otra forma de ver esto es que una asociación es equivalente a la clase origen que tiene un pseudo atributo de la clase destino. Un objeto de la clase origen puede hacer referencia a un objeto de la clase destino al utilizar este pseudo atributo; véase figura 9.16.

No existe ningún lenguaje de programación orientado a objetos comúnmente utilizado que tenga un constructor específico de lenguaje para soportar asociacio-

nes. Por lo tanto, cuando el código se genera automáticamente desde un modelo UML, las asociaciones uno a uno se convierten en atributos de la clase origen.



Si una relación navegable tiene un nombre de rol, entonces es como si la clase origen tiene un pseudoatributo con el mismo nombre que el nombre del rol y el mismo tipo que la clase destino.

Figura 9.16.

En la figura 9.17 el código generado tiene una clase Casa que contiene un atributo denominado dirección que es del tipo Dirección. Observe cómo el nombre de rol proporciona el nombre de atributo y la clase en el extremo de la asociación proporciona la clase de atributo. El código Java a continuación se ha generado a partir del modelo en la figura 9.17.

```

public class Casa
{
    private Dirección dirección;
}
  
```

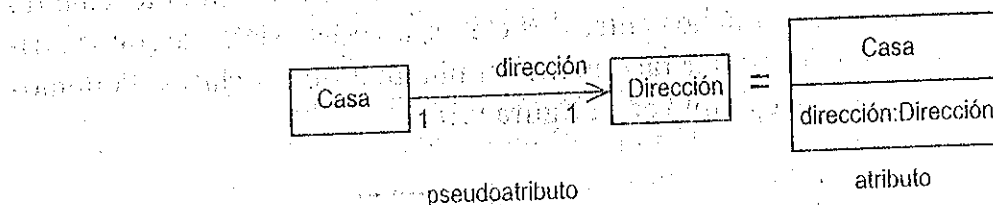


Figura 9.17.

Puede ver que existe una clase Casa que tiene un atributo denominado dirección que es del tipo Dirección. Observe que el atributo dirección tiene visibilidad privada; ésta es la opción predeterminada para la mayor parte de la generación de código.

Las multiplicidades destino mayores que 1 se implementan como:

- Un atributo del tipo array (un constructor que se soporta en la mayoría de los lenguajes), o bien
- Un atributo de algún tipo que es una colección.

Las colecciones son clases cuyas instancias tienen el comportamiento especializado de poder almacenar y recuperar referencias a otros objetos. Un ejemplo Java común de una colección es un Vector, pero existen muchas más. Tratamos las colecciones en más detalle en el capítulo 18.

Esta noción de pseudo atributos está bien para relaciones uno a uno y uno a muchos, pero empieza a desglosarse cuando considera relaciones muchos a muchos. Verá cómo están implementados en el capítulo 18.

Utilice asociaciones solamente cuando la clase destino es una parte importante del modelo. De lo contrario, modele la relación al utilizar atributos. Las clases importantes son clases de negocio que describen parte del dominio de negocio. Las clases importantes son componentes de biblioteca como clases String, Date y Time.

Hasta cierto sentido, la opción de asociaciones explícitas versus atributos es una cuestión de estilo. El mejor enfoque siempre es uno en el que el modelo y los diagramas expresan el problema claramente y de forma precisa. A menudo, es más claro mostrar una asociación a otra clase que modelar la misma relación como un atributo que sería más difícil de ver. Cuando la multiplicidad destino es mayor que 1, esto es una buena indicación de que el destino es importante para el modelo, y por lo tanto utiliza asociaciones para modelar la relación.

Si la multiplicidad destino es exactamente 1, el objeto destino puede ser una parte de la fuente y por lo tanto no merece la pena mostrarse como una asociación, se puede modelar mejor como un atributo. Esto es especialmente cierto si la multiplicidad es exactamente 1 en ambos extremos de la relación (como en la figura 9.17) donde ni la fuente ni el destino pueden existir solos.

### 9.4.5. Clases de asociación

Un problema común en el modelado orientado a objetos es el siguiente: cuando tiene una relación muchos a muchos entre dos clases, a veces existen algunos atributos que no se pueden acomodar fácilmente en ninguna de las clases. Podemos ilustrar esto al considerar el ejemplo en la figura 9.18.

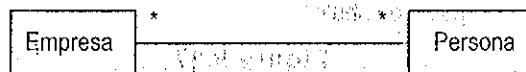


Figura 9.18.

A primera vista esto parece un modelo bastante inocuo:

- Todo objeto Persona puede trabajar para muchos objetos Empresa.
- Todo objeto Empresa puede emplear muchos objetos Persona.

Sin embargo, ¿qué sucede si añade la regla de negocio que cada Persona tiene un sueldo por cada Empresa en la que esté empleado? ¿Dónde se debería grabar el sueldo, en la clase Persona o en la clase Empresa?

No puede hacer que el sueldo de Persona sea un atributo de la clase Persona, ya que cada instancia Persona puede trabajar para muchas Empresas y puede tener un sueldo diferente con cada Empresa. De forma similar, no puede hacer que el sueldo de Persona sea un atributo de Empresa, ya que cada instancia Empresa emplea muchas Personas, todas con sus sueldos diferentes. La respuesta es que el sueldo es una propiedad de la propia asociación. Para cada asociación de empleo que un objeto Persona tiene con un objeto Empresa, existe un sueldo específico.



UML le permite modelar esta situación con una clase de asociación como se muestra en la figura 9.19. Es importante entender esta sintaxis; muchas personas creen que la clase de asociación es un cuadro que cuelga de la asociación. Sin embargo, nada más lejos de la realidad. La clase de asociación es en realidad la línea de asociación (incluidos todos los nombres de rol y multiplicidades), la línea de puntos que desciende y el cuadro de clase en el extremo de la línea de puntos. Es decir, todo el conjunto, todo lo que se muestra en el área indicada.

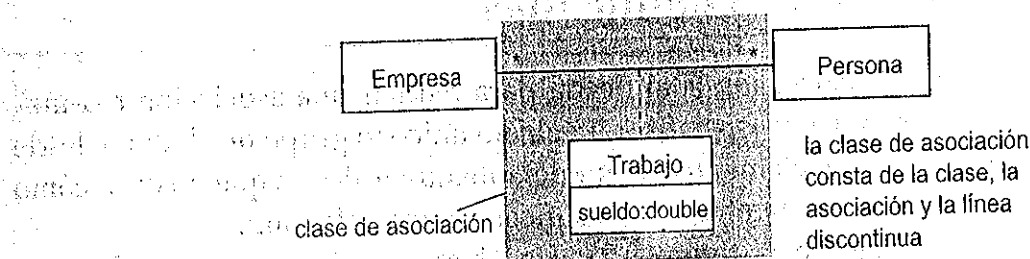


Figura 9.19.

De hecho, una clase de asociación es una asociación que es también una clase. No solamente conecta dos clases como una asociación, sino que define un conjunto de características que pertenecen a la propia asociación. Las clases asociación pueden tener atributos, operaciones y otras asociaciones.

Las instancias de la clase de asociación son vínculos que tienen atributos y operaciones. La identidad única de estos vínculos está determinada exclusivamente por las identidades de los objetos en cada extremo. Este factor restringe la semántica de la clase de asociación; solamente puede utilizarla cuando existe un solo vínculo único entre dos objetos en cualquier momento en el tiempo. Esto es simplemente porque cada vínculo, que es una instancia de la clase de asociación, debe tener su propia identidad única. En la figura 9.19, utilizar la clase de asociación significa que restringe el modelo como el de un objeto Persona y un objeto Empresa donde solamente puede haber un objeto Trabajo. Es decir, cada Persona puede tener solamente un Trabajo con una Empresa dada.

Sin embargo, si tiene la situación donde un objeto Persona dado puede tener más de un Trabajo con un objeto Empresa dado, entonces no puede utilizar una clase asociación, la semántica no coincide.

Pero aún así, necesita algún lugar donde situar el sueldo para cada combinación Empresa/Trabajo/Persona y hacer real la relación al expresarlo como una clase normal. En la figura 9.20 Trabajo es ahora una clase normal y puede ver que una Persona puede tener muchos Trabajos donde cada Trabajo es para exactamente una Empresa.

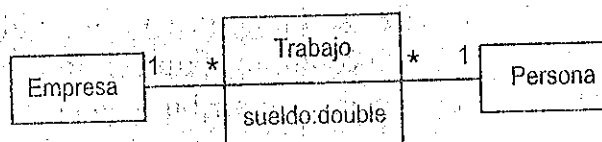


Figura 9.20.

Para ser franco, muchos modeladores de objeto no entienden la diferencia semántica entre clases asociación y relaciones cosificadas y por lo tanto las dos se utilizan de modo indistinto. Sin embargo, la diferencia es muy sencilla: puede utilizar clases de asociación solamente cuando cada vínculo tiene una identidad única. Recuerde que identidad de vínculo está determinada por las identidades de los objetos en los extremos del vínculo.

### 9.4.6. Asociaciones cualificadas

Puede utilizar una asociación cualificada para reducir una asociación n-a-muchas a una asociación n-a-1 al especificar un objeto único (o grupo de objetos) desde el destino establecido. Son elementos de modelado muy útiles ya que ilustran cómo puede buscar o navegar hasta objetos específicos en una colección.

Considere el modelo en la figura 9.21. Un objeto Club está vinculado a un conjunto de objetos Socio y un objeto Socio está de igual forma vinculado a exactamente un objeto Club.

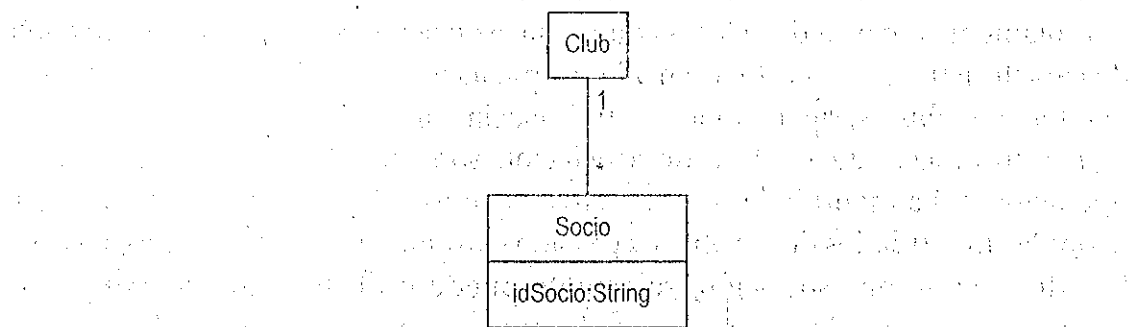


Figura 9.21.

Se plantea la siguiente pregunta: dado un objeto Club que está vinculado a un conjunto de objetos Socio, ¿cómo podría navegar hasta un objeto Socio específico? Claramente, necesita alguna clave única que pueda utilizar para buscar un objeto Socio particular del conjunto. Esto se conoce como calificador. Muchos calificadores son posibles (nombre, número tarjeta crédito, número seguridad social) pero en el ejemplo anterior, todo objeto Socio tiene un valor de atributo idSocio que es único a ese objeto. Esto entonces es la clave de búsqueda en este modelo.

Puede mostrar esto en el modelo al anexar un calificador al extremo Club de la asociación. Es importante reconocer que este calificador pertenece al extremo de la asociación y no a la clase Club. Este calificador especifica una clave única y al hacerlo resuelve la relación uno a muchos a uno a uno, como se muestra en la figura 9.22. Las asociaciones cualificadas son una forma estupenda de mostrar cómo selecciona un objeto específico de un conjunto al utilizar una clave única. Los calificadores normalmente hacen referencia a un atributo en la clase destino, pero puede haber alguna otra expresión siempre que se entienda y seleccione un solo objeto del conjunto.

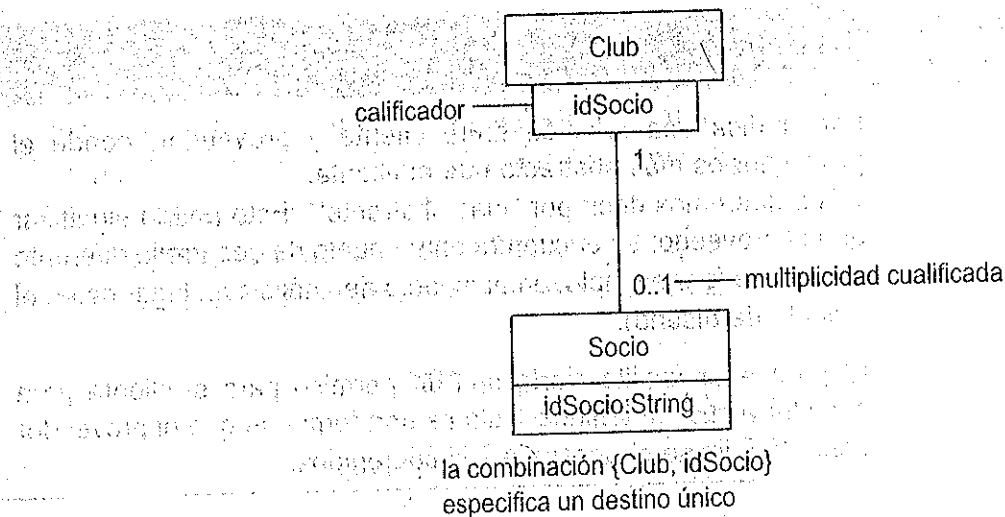


Figura 9.22.

## 9.5. ¿Qué es una dependencia?

Una dependencia indica una relación entre dos o más elementos de modelo donde un cambio en un elemento (el proveedor) puede afectar o proporcionar la información necesaria por el otro elemento (el cliente).

En otras palabras, el cliente depende en cierto sentido del proveedor. Utilizamos dependencias para modelar relaciones entre clasificadores donde un clasificador depende del otro de alguna forma, pero la relación no es realmente una asociación o generalización.

Por ejemplo, puede pasar un objeto de una clase como un parámetro a una operación de un objeto de una clase diferente. Existe cierto tipo de relación entre las clases de esos objetos, pero no es realmente una asociación. Puede utilizar la relación de dependencia (especializada por ciertos estereotipos predefinidos) como algo general para modelar este tipo de relación. Ya ha visto un tipo de dependencia, la relación `<<stantiate>>`, pero existen muchas más. En los siguientes apartados examinaremos los estereotipos de dependencia más comunes.

UML 2 especifica tres tipos básicos de dependencia mostrados en la tabla 9.2. Incluimos una explicación de estos, pero en el modelado del día a día, raramente utilizará cualquier otro elemento que no sea una flecha punteada de dependencia y por lo general no se preocupa en especificar el tipo de dependencia.

Tabla 9.2.

Tipo	Semántica
Uso	El cliente utiliza alguno de los servicios puestos a disposición por el proveedor para implementar su propio comportamiento; éste es el tipo de dependencia utilizado más comúnmente.

Tipo	Semántica
Abstracción	Esto indica una relación entre cliente y proveedor, donde el proveedor es más abstracto que el cliente. ¿Qué queremos decir por "más abstracto"? Esto podría significar que el proveedor se encuentra en un punto de desarrollo diferente del cliente (por ejemplo, en el <u>modelo de análisis</u> en lugar de en el <u>modelo de diseño</u> ).
Permiso	El proveedor facilita cierto tipo de permiso para el cliente para acceder a sus contenidos. Ésta es una forma de que el proveedor controle y limite el acceso a sus contenidos.

Las dependencias no ocurren sólo entre clases. Pueden ocurrir entre:

- Paquetes y paquetes.
- Objetos y clases.

También pueden ocurrir entre una operación y una clase, aunque es bastante extraño mostrar esto explícitamente en un diagrama porque normalmente es un nivel de detalle muy grande. Algunos ejemplos de tipos diferentes de dependencia se muestran en la figura 9.23, y los tratamos en el resto de apartados de este capítulo.

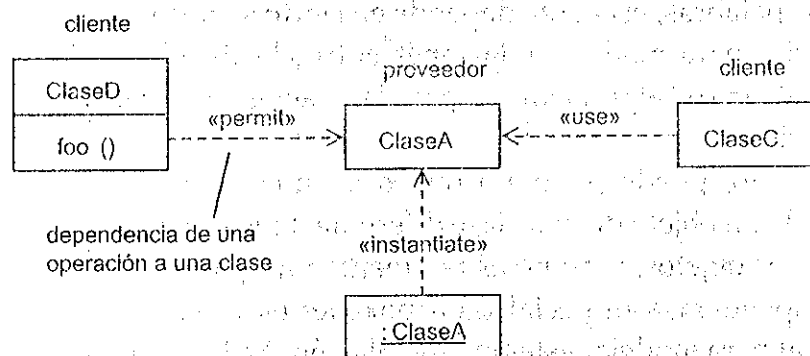


Figura 9.23

La mayor parte del tiempo utiliza solamente una flecha punteada sin adornos para indicar una dependencia y no se preocupa por el tipo de dependencia que es. De hecho, el tipo de dependencia está a menudo claro sin un estereotipo simplemente por el contexto. Sin embargo, si quiere o necesita ser más específico sobre el tipo de dependencia, entonces UML define un rango completo de estereotipos estándar que puede utilizar.

### 9.5.1. Dependencias de uso

Existen cinco dependencias de uso: `<<use>>`, `<<call>>`, `<<parameter>>`, `<<send>>`, `<<instantiate>>`. Examinamos cada una de éstas en los siguientes subapartados.

### 9.5.1.1. <<use>>

El estereotipo de dependencia más común es <<use>>, que simplemente indica que el cliente hace uso del proveedor de alguna forma. Si simplemente ve una flecha discontinúa de dependencia sin estereotipo, puede estar seguro de que lo que se pretende es <<use>>.

La figura 9.24 muestra dos clases, A y B, que tienen una dependencia <<use>> entre ellas. Esta dependencia está generada por cualquiera de los siguientes casos:

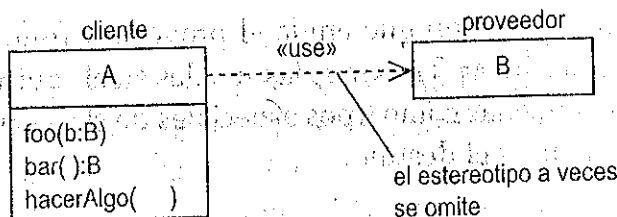


Figura 9.24.

1. Una operación de clase A necesita un parámetro de clase B.
2. Una operación de clase A devuelve un valor de clase B.
3. Una operación de clase A utiliza un objeto de clase B en algún lugar en su implementación, pero no como un atributo.

Los casos 1 y 2 están claros, pero el caso 3 es más interesante. Tendría este caso si una de las operaciones de la clase A creó un objeto transitorio de clase B. Aquí tiene un fragmento de código Java para este caso:

```

class A {
    void hacerAlgo() {
        B miB = new B();
        // Utilizar miB de alguna forma
    }
}
  
```

Aunque puede utilizar una sola dependencia <<use>> como un genérico para los tres casos listados anteriormente, existen otros estereotipos de dependencia más específicos que puede aplicar. Puede modelar los casos 1 y 2 de forma más precisa con una dependencia <<parameter>> y el caso 3 con una dependencia <<call>>. Sin embargo, éste es un nivel de detalle que raramente se requiere en un modelo UML y la mayoría de los modeladores encuentran mucho más sencillo situar una sola dependencia <<use>> entre las clases apropiadas como se ha mostrado.

### 9.5.1.2. <<call>>

La dependencia <<call>> es entre operaciones; la operación cliente invoca la operación proveedor. Este tipo de dependencia no se utiliza mucho en el modelado

UML. Se aplica a un nivel más profundo de detalle de lo que pretenden llegar la mayoría de los modeladores. Igualmente, muy pocas herramientas de modelado soportan en estos momentos dependencias entre operaciones.

#### 9.5.1.3. <<parameter>>

---

El proveedor es un parámetro de la operación cliente.

#### 9.5.1.4. <<send>>

---

El cliente es una operación que envía el proveedor (que debe ser una señal) a algún destino sin especificar. Tratamos las señales en el capítulo 15, pero por ahora, simplemente piense en ellas como tipos especiales de clases utilizados para transferir datos entre el cliente y el destino.

#### 9.5.1.5. <<instantiate>>

---

El cliente es una instancia del proveedor.

### 9.5.2. Dependencias de abstracción

---

Las dependencias de abstracción modelan dependencias entre elementos que se encuentran en diferentes niveles de abstracción. Un ejemplo podría ser una clase en un modelo de análisis, y la misma clase en el modelo de diseño. Existen cuatro dependencias de abstracción: <<trace>>, <<substitute>>, <<refine>> y <<derive>>.

#### 9.5.2.1. <<trace>>

---

Utilice a menudo una dependencia <<trace>> para ilustrar una relación en la que el proveedor y el cliente representan el mismo concepto pero están en diferentes modelos. Por ejemplo, el proveedor y el cliente podrían estar en diferentes niveles de desarrollo. El proveedor podría ser una vista de análisis de una clase y el cliente una vista de diseño más detallada. También podría utilizar <<trace>> para mostrar una relación entre un requisito funcional como "El cajero debería permitir la retirada de efectivo hasta el límite de crédito de la tarjeta" y el caso de uso que soporta este requisito.

#### 9.5.2.2. <<substitute>>

---

La relación <<substitute>> indica que el cliente se puede sustituir por el proveedor en tiempo de ejecución. El proceso de sustitución se basa en que el cliente y el proveedor se ajusten a contratos e interfaces comunes, es decir, tienen que poner disponibles el mismo conjunto de servicios. Observe que esta sustitución no se consigue por medio de relaciones de especialización/generalización entre el cliente y el proveedor (tratamos especialización/generalización en el capítulo 10). De hecho, <<substitute>> está específicamente diseñada para utilizarse en aquellos entornos que no soportan especialización/generalización.

### 9.5.2.3. <<refine>>

Mientras que la dependencia <<trace>> es entre elementos en diferentes modelos, <<refine>> se puede utilizar entre elementos en el mismo modelo. Por ejemplo, puede tener dos versiones de una clase en un modelo, una de las cuales está optimizada para rendimiento. Puesto que la optimización de rendimiento es un tipo de mejora, puede modelar esto como una dependencia <<refine>> entre las dos clases junto con una nota indicando la naturaleza de la mejora.

### 9.5.2.4. <<derive>>

Puede utilizar el estereotipo <<derive>> cuando quiere mostrar explícitamente que un elemento se puede derivar de alguna forma de algún otro elemento. Por ejemplo, si tiene una clase CuentaBancaria y la clase contiene una lista de Transacciones donde cada Transacción contiene una Cantidad de dinero, siempre puede calcular el saldo actual bajo demanda al sumar la Cantidad en todas las Transacciones. Existen tres formas de mostrar que se puede obtener el saldo de la cuenta (una Cantidad). Esto se muestra en la tabla 9.3.

Tabla 9.3.

Modelo	Descripción
<pre> classDiagram     class CuentaBancaria     class Transacción     class Cantidad     CuentaBancaria "1" -- "0..*" Transacción     Transacción "1" -- "1" Cantidad     CuentaBancaria ..&gt; Cantidad : «derive» saldo           </pre>	<p>La clase CuentaBancaria tiene una asociación derivada a Cantidad donde Cantidad desempeña el rol del saldo de la CuentaBancaria. Este modelo enfatiza que saldo se deriva de la colección de Transacciones de CuentaBancaria.</p>
<pre> classDiagram     class CuentaBancaria     class Transacción     class Cantidad     CuentaBancaria "1" -- "0..*" Transacción     Transacción "1" -- "1" Cantidad     CuentaBancaria --&gt; Cantidad : /saldo           </pre>	<p>En este caso, se utiliza una barra inclinada en el nombre del rol para indicar que la relación entre CuentaBancaria y Cantidad está derivada. Esto es menos explícito, ya que no muestra de dónde se deriva el saldo.</p>
<pre> classDiagram     class CuentaBancaria     class Transacción     class Cantidad     CuentaBancaria "1" -- "0..*" Transacción     Transacción "1" -- "1" Cantidad     CuentaBancaria --&gt; Cantidad : /saldo:Cantidad           </pre>	<p>Aquí el saldo se muestra como un atributo derivado; esto se indica por la barra inclinada que prefija el nombre del atributo. Esta es la expresión de dependencia más concisa.</p>

Todas estas formas de mostrar que los saldos se pueden derivar son equivalentes aunque el primer modelo en la tabla 9.3 es el más explícito. Tendemos a preferir los modelos explícitos.



### 9.5.3. Dependencias de permiso

---

Las dependencias de permiso tratan sobre expresar la posibilidad de que un elemento acceda a otro elemento. Existen tres dependencias de permiso: `<<access>>`, `<<import>>` y `<<permit>>`.

#### 9.5.3.1. `<<access>>`

---

La dependencia `<<access>>` es entre paquetes. Los paquetes se utilizan en UML para agrupar elementos. El punto esencial aquí es que `<<access>>` permite que un paquete acceda a todos los contenidos públicos de otro paquete. Sin embargo, cada uno de los paquetes define un espacio de nombres y con `<<access>>` el espacio de nombres permanece separado. Esto significa que los elementos en el paquete cliente deben utilizar nombres de ruta cuando quieren hacer referencia a los elementos en el paquete del proveedor. Véase el capítulo 11 para una explicación más detallada.

#### 9.5.3.2. `<<import>>`

---

La dependencia `<<import>>` es conceptualmente similar a `<<access>>` excepto que el espacio de nombres del proveedor se fusiona en el espacio de nombres del cliente. Esto permite que los elementos en el cliente accedan a elementos en el proveedor sin tener que calificar nombres de elemento con el nombre del paquete. Sin embargo, a veces puede dar lugar a que el espacio de nombres entre en conflicto cuando un elemento en el cliente tiene el mismo nombre que un elemento en el proveedor.

Claramente, en este caso, debe utilizar los nombres de ruta para resolver el conflicto. El capítulo 11 proporciona una explicación más detallada.

#### 9.5.3.3. `<<permit>>`

---

La dependencia `<<permit>>` permite una violación de encapsulación controlada, pero se debería evitar. El elemento cliente tiene acceso al elemento proveedor, cualquiera que sea la visibilidad declarada del proveedor. Existe una dependencia `<<permit>>` entre dos clases estrechamente relacionadas donde es ventajoso para la clase cliente acceder a los miembros privados del proveedor. No todos los lenguajes informáticos soportan dependencias `<<permit>>`; C++ permite que una clase declare amigos que tienen permiso para acceder a sus socios privados, pero esta característica se ha excluido de Java y C#.

## 9.6. ¿Qué hemos aprendido?

---

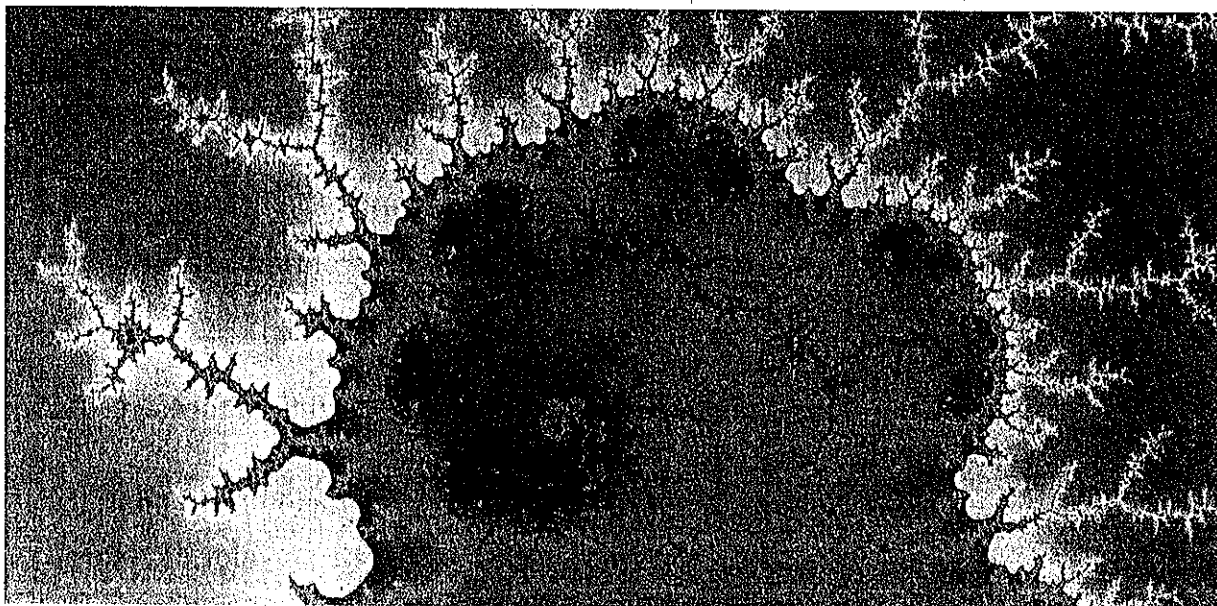
En este capítulo hemos empezado a examinar las relaciones, que son lo importante de los modelos UML. Ha aprendido lo siguiente:



- Las relaciones son conexiones semánticas entre elementos.
- Las conexiones entre objetos se denominan vínculos.
  - Un vínculo ocurre cuando un objeto alberga una referencia de objeto a otro objeto.
  - Los objetos realizan comportamiento del sistema al colaborar:
    - La colaboración ocurre cuando los objetos se envían mensajes por los vínculos.
    - Cuando un mensaje se recibe por un objeto, éste ejecuta la operación apropiada.
  - Diferentes lenguajes orientados a objetos implementan vínculos de formas diferentes.
- Los diagramas de objeto muestran objetos y sus vínculos en un punto determinado en el tiempo.
  - Son instantáneas de un sistema orientado a objetos que se ejecuta en un momento determinado.
  - Los objetos pueden adoptar roles entre ellos, el rol desempeñado por un objeto en un vínculo define la semántica de su parte en la colaboración.
  - Los vínculos n-arios pueden conectar más de dos objetos, se dibujan con una ruta hacia cada objeto pero no se utilizan demasiado.
- Las rutas de acceso son líneas que conectan elementos de modelado UML:
  - Estilo ortogonal, líneas rectas con esquinas rectangulares.
  - Estilo oblicuo, líneas inclinadas.
  - Estilo curvado, líneas curvadas.
- Sea coherente y siga un estilo u otro, a menos que mezclar estilos aumente la legibilidad del diagrama (normalmente no lo hace).
- Las asociaciones son conexiones semánticas entre clases.
  - Si existe un vínculo entre dos objetos, debe haber una asociación entre las clases de esos objetos.
  - Los vínculos son instancias de asociaciones igual que los objetos son instancias de clases.
  - Las asociaciones pueden tener opcionalmente lo siguiente:
    - Nombre de asociación:
      - Pueden estar prefijadas o postfijadas con una pequeña punta de flecha negra para indicar la dirección en la que se debería leer el nombre.
      - Debería ser un verbo o una frase verbal.
      - Se escriben poniendo en mayúscula la primera letra de cada palabra con la primera letra de todas en minúscula.
      - Utiliza un nombre de asociación o nombre de rol, pero no ambos.

- Los nombres de roles en uno o ambos extremos de asociación:
  - Será un nombre o frase nominal que describe la semántica del rol.
  - Se escriben poniendo en mayúscula la primera letra de cada palabra con la primera letra de todas en minúscula.
- Multiplicidad:
  - Indica el número de objetos que se pueden implicar en la relación en cualquier momento en el tiempo.
  - Los objetos pueden ir y venir, pero la multiplicidad restringe el número de objetos en la relación en cualquier punto en el tiempo.
  - La multiplicidad está especificada por una lista de intervalos separada por coma, por ejemplo, 0...1, 3..5.
  - No existe multiplicidad predeterminada. Si la multiplicidad no se muestra explícitamente, está sin decidir.
- Navegabilidad:
  - Mostrada por una punta de flecha en un extremo de la relación; si una relación no tiene puntas de flecha, entonces es bidireccional.
  - La navegabilidad indica que puede atravesar la relación en la dirección de la flecha.
  - También puede atravesarla en el sentido contrario, pero sería computacionalmente caro hacerlo así.
- Una asociación entre dos clases es equivalente a una clase que tiene un pseudo atributo que puede albergar una referencia a un objeto de otra clase.
  - Puede utilizar asociaciones y atributos de forma intercambiable.
  - Utilice asociación cuando tenga una clase importante en el extremo de la asociación que desee enfatizar.
  - Utilice atributos cuando la clase en el extremo de la relación no sea importante (por ejemplo, una clase de biblioteca como String o Date).
- Una clase de asociación es una asociación que también es una clase:
  - Puede tener atributos, operaciones y relaciones.
  - Puede utilizar esta clase cuando existe exactamente un vínculo único entre cualquier par de objetos en cualquier punto en el tiempo.
  - Si un par de objetos puede tener muchos vínculos entre sí en un punto determinado en el tiempo, entonces cosifica la relación al reemplazarla con una clase normal.
- Las asociaciones calificadas utilizan un calificador para seleccionar un objeto único del conjunto destino:
  - El calificador debe ser una clave única en el conjunto destino.
  - Las asociaciones calificadas reducen la multiplicidad de relaciones n a muchas a n a 1.

- Existe una forma de utilidad de llamar la atención sobre los identificadores únicos.
- Las dependencias son relaciones en las que un cambio en el proveedor afecta o proporciona información al cliente.
  - El cliente depende del proveedor de alguna forma.
  - Las dependencias se dibujan como una flecha discontinua desde el cliente al proveedor.
  - Dependencias de uso:
    - `<<use>>`: El cliente hace uso del proveedor de alguna forma; éste es el uso general.
    - `<<call>>`: La operación cliente invoca la operación proveedor.
    - `<<parameter>>`: El proveedor es un parámetro o valor de retorno de una de las operaciones del cliente.
    - `<<send>>`: El cliente envía al proveedor (que debe ser una señal) al destino especificado.
    - `<<instantiate>>`: El cliente es una instancia del proveedor.
  - Dependencias de abstracción:
    - `<<trace>>`: El cliente es un desarrollo histórico del proveedor.
    - `<<substitute>>`: El cliente se puede sustituir por el proveedor en tiempo de ejecución.
    - `<<refine>>`: El cliente es una versión del proveedor.
    - `<<derive>>`: El cliente se puede derivar de alguna forma del proveedor:
      - Puede mostrar relaciones derivadas explícitamente al utilizar una dependencia `<<derive>>`.
      - Puede mostrar relaciones derivadas al prefijar el rol o nombre de relación con una barra inclinada.
      - Puede mostrar atributos derivados al prefijar el nombre del atributo con una barra inclinada.
  - Dependencias de permiso:
    - `<<access>>`: Una dependencia entre paquetes donde el paquete de cliente puede acceder a todos los contenidos públicos del paquete del proveedor; los espacios de nombre de los paquetes permanecen separados.
    - `<<import>>`: Una dependencia entre paquetes donde el paquete de cliente puede acceder a todos los contenidos públicos del paquete proveedor; los espacios de nombre de los paquetes se fusionan.
    - `<<permit>>`: Una violación de encapsulación controlada, donde el cliente puede acceder a los miembros privados del proveedor; esto no se soporta ampliamente y se debería evitar en lo posible.



---

10

# Herencia y polimorfismo

---