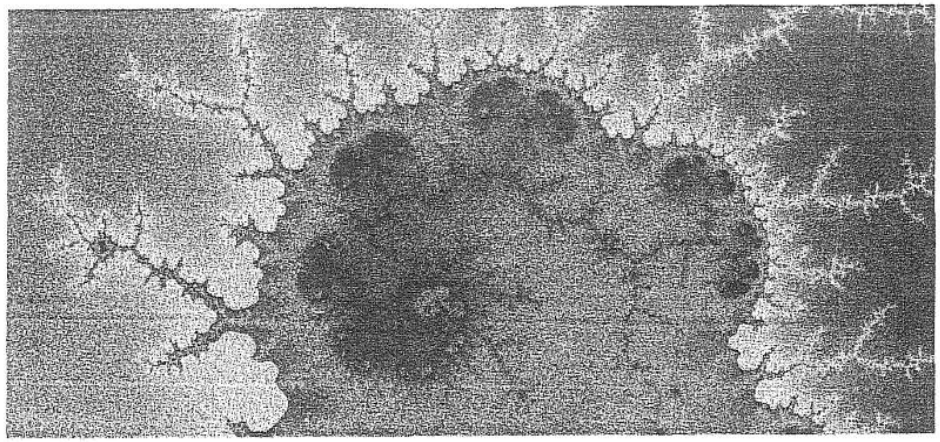

17
Diseñar clases

17.1. Presentación del capítulo

Este capítulo trata el diseño de clases. Éstos son los bloques de construcción del modelo de diseño y son vitales para usted, como diseñador orientado a objetos, para entender cómo modelar estas clases de forma eficaz.

Después de proporcionar el contexto UP, describimos la anatomía de una clase de diseño y luego pasamos a una consideración de lo que conforma una clase de diseño bien creada. Tratamos los requisitos de totalidad, suficiencia, sencillez, cohesión alta, acoplamiento bajo, y la aplicabilidad de agregación vs. herencia.

17.2. Actividad UP: Diseñar una clase

En el detalle del workflow de diseño de UP (véase la figura 16.6), después del diseño de la arquitectura, las siguientes actividades son `Diseñar una clase` y `Diseñar un caso de uso` (véase el capítulo 20). Estas dos actividades ocurren concurrentemente e iterativamente.

En este apartado examinamos la actividad de UP `Diseñar una clase`. Esta actividad se muestra en la figura 17.2. Hemos ampliado la actividad para mostrar `interfaz [completa]` como un resultado explícito de `Diseñar una clase`. El artefacto aparece en gris para mostrar que es una modificación; en la descripción original de la actividad era un resultado implícito.

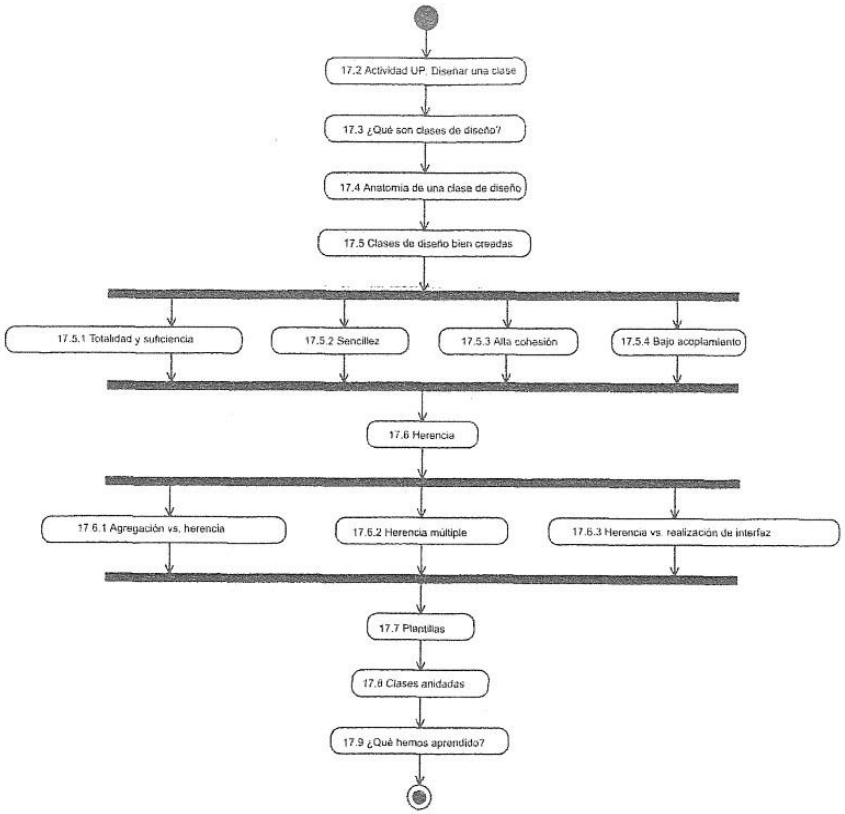

Figura 17.1.

Ha visto cómo crear el artefacto de entrada `clase de análisis [completo]` en la parte de análisis de este libro y por lo tanto no decimos más sobre ella aquí.

Merece la pena considerar el artefacto `diseñar clase [esbozado]` en cierto detalle. Parece por la actividad como si existieran dos artefactos aparte y distintos, `diseñar clase [esbozado]` y `diseñar clase [completo]`. Sin embargo, éste no es el caso. Estos dos artefactos representan el mismo artefacto (una clase de diseño) en diferentes niveles en su evolución.

Si toma una instantánea de los artefactos de un proyecto UP al final de la fase de elaboración o al principio de la de construcción, no encontrará artefactos etiquetados como `diseñar clase [esbozado]` o `diseñar clase [completo]`. En su lugar, simplemente serán clases de diseño y cada una de ellas estará en un nivel diferente en su desarrollo.

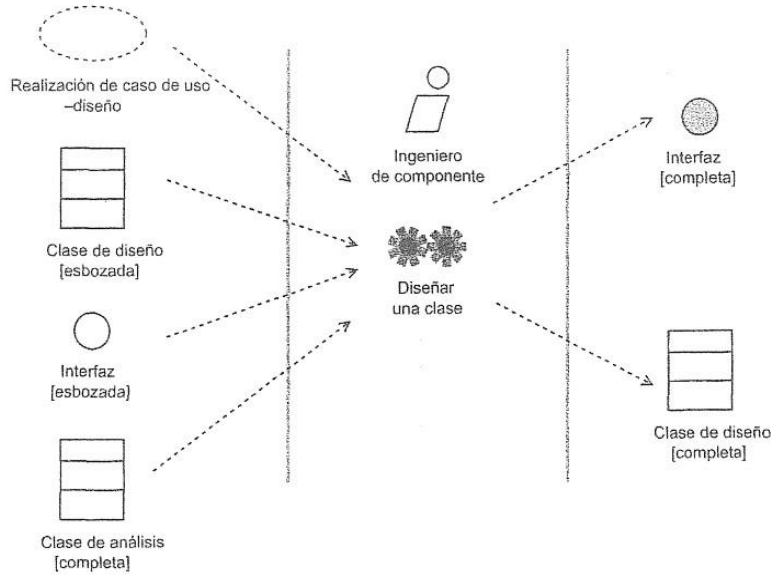

Figura 17.2. Adaptada de la figura 9.39 [Jacobson 1] con permiso de Addison-Wesley.

Una clase de diseño "completa" desde la perspectiva UP es una que está suficientemente detallada para servir como una buena base para crear código fuente. Éste es el punto clave y uno que los modeladores principiantes a menudo olvidan. Las clases de diseño solamente se tienen que modelar en suficiente detalle para que el código se pueda desarrollar a partir de ellas, por lo que raramente se modelan de manera exhaustiva.

El nivel necesario de detalle depende de su proyecto. Si va a generar código directamente del modelo, sus clases de diseño se tendrán que modelar en gran detalle. Por otro lado, si simplemente van a servir como anteproyecto para programadores, se pueden modelar en menos detalle.

En este capítulo, mostramos cómo modelar clases de diseño con suficiente detalle para cualquier proyecto. Las consideraciones para `diseñar clase [esbozado]` y `diseñar clase [completo]` también se aplican para `interfaz [esbozada]` e `interfaz [completa]`.

El artefacto de entrada `realización de caso de uso - diseño` es simplemente una realización de caso de uso en un punto tardío en su ciclo de vida. Aunque se muestra fluyendo en `Diseñar una clase`, incluye clases de diseño como parte de su estructura y se desarrolla en paralelo con ellas. Dejamos su explicación hasta el capítulo 20 porque encontramos más eficaz tratar sus partes componentes primero.

17.3. ¿Qué son clases de diseño?

Las clases de diseño son clases cuyas especificaciones se han completado hasta tal nivel que se pueden implementar.

En análisis, el origen de las clases es el ámbito del problema. Éste es el conjunto de requisitos que describe el problema que está tratando de solucionar. Ha visto que los casos de uso, especificaciones de requisitos, glosarios y cualquier otra información pertinente se puede utilizar como una fuente de clases de análisis.

Las clases de diseño proceden de dos lados:

- **El ámbito del problema vía una mejora de las clases de análisis:** Esta mejora implica añadir detalles de implementación. A media que hace esto, encuentra que necesita desglosar una clase de análisis de alto nivel en dos o más clases detalladas de diseño. Existe una relación `<<trace>>` entre una clase de análisis y una o más clases de diseño que describen su implementación.
- **El ámbito de solución:** El ámbito de solución es el ámbito de las librerías de clases de utilidad y componentes reutilizables como `Time`, `Date`, `String`, colecciones, etc. Middleware como bases de datos (tanto relacionales como de objeto), frameworks de componentes como .NET, CORBA y Enterprise JavaBeans, viven aquí también. El ámbito de la solución también contiene entornos de trabajo GUI. Este ámbito proporciona las herramientas técnicas que le permiten implementar un sistema.

Esto se ilustra en la figura 17.3.

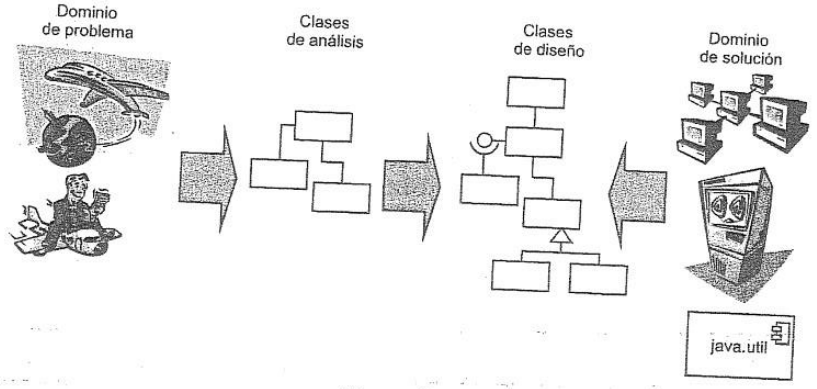

Figura 17.3.

El análisis trata sobre modelar lo que el sistema debería hacer. El diseño es sobre modelar cómo el comportamiento se puede implementar.

¿Por qué puede una clase de análisis convertirse en una o más clases de diseño o interfaces? Una clase de análisis se especifica a un nivel muy alto de abstracción. No se preocupa por el conjunto completo de atributos y el conjunto de operaciones es solamente un esbozo que captura los servicios clave ofrecidos por la clase.

Cuando mueve esta clase al diseño, debe especificar totalmente todas las operaciones y atributos, por lo que es bastante común encontrar que la clase se ha hecho muy grande. Si esto sucede, debería desglosarla en una o más clases más pequeñas. Recuerde que siempre debería tratar de diseñar clases que son pequeñas unidades cohesivas que realizan una o dos tareas realmente bien. Debería evitar a toda costa el tipo de clase "navaja suiza" que trata de hacerlo todo.

El método elegido de implementación determina el grado de totalidad que necesita en las especificaciones de la clase de diseño. Si el modelo de la clase de diseño se proporcionara a programadores que lo utilizaran como una guía para escribir código, entonces las clases de diseño solamente necesitan estar lo suficientemente completas para permitirles realizar esa tarea de forma eficaz. Esto depende de la habilidad que tengan los programadores y lo bien que entiendan los ámbitos del problema y la solución; tiene que averiguar esto para su proyecto en particular.

Sin embargo, si trata de generar código desde las clases de diseño con una herramienta de modelado apropiada, las especificaciones de clase de diseño deben estar completas en todos los aspectos ya que un generador, a diferencia de un programador, no puede llenar los huecos. Durante el resto de este capítulo asumimos que necesita un nivel muy alto de totalidad.

17.4. Anatomía de una clase de diseño

Con las clases de análisis está tratando de capturar el comportamiento requerido del sistema sin preocuparse de por cómo se va a implementar este comportamiento. Con las clases de diseño tiene que especificar exactamente cómo cada clase completará sus responsabilidades. Para hacer esto, debe realizar lo siguiente:

- Complete el conjunto de atributos y especifíquelos incluyendo nombre, tipo, visibilidad y (opcionalmente) un valor predeterminado.
- Complete el conjunto de operaciones y especifíquelas incluyendo nombre, lista de parámetro y tipo de retorno.

Este proceso se ilustra con un ejemplo sencillo en la figura 17.4. Como vio en el capítulo 8, una operación en una clase de análisis es una especificación lógica de alto nivel de una pieza de funcionalidad ofrecida por una clase. En las clases de diseño correspondientes, cada operación de clase de análisis se convierte en una o más operaciones detalladas que se pueden implementar como código fuente. Por lo tanto, una operación de análisis de alto nivel se puede resolver en una o más operaciones de diseño implementables. Estas operaciones detalladas a nivel de diseño se conocen como métodos.

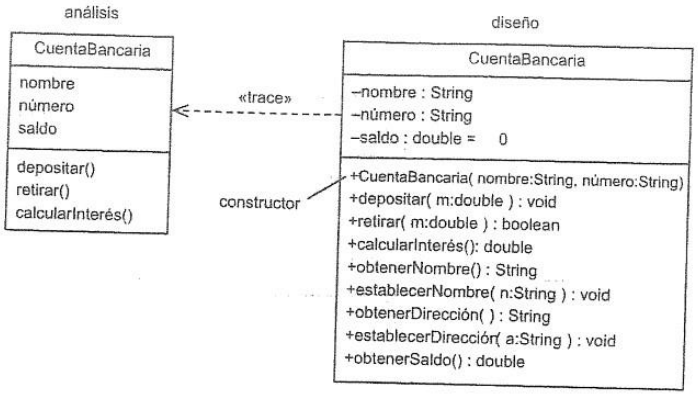

Figura 17.4.

Para ilustrar esto, considere el siguiente ejemplo. En un sistema de facturación de una aerolínea debe especificar en análisis una operación de alto nivel denominada `facturar()`. Sin embargo, como sabría si alguna vez hubiera esperado para facturar para un vuelo, la facturación es un proceso de negocio bastante complejo que implica recopilar y verificar cierta información del pasajero, coger el equipaje y asignar un asiento en un avión. Por lo tanto, es razonable suponer que la operación de análisis de alto nivel `facturar()` se desglosará en una cascada de operaciones de bajo nivel cuando realiza el diseño detallado del proceso. Podría ser que sigue manteniendo una operación `facturar()` de alto nivel, pero en diseño, esta operación invocará una cascada de operaciones de ayuda para desempeñar su responsabilidad. Puede ser incluso que el proceso de facturación sea lo suficientemente complejo para requerir nuevas clases de ayuda que no se identificaron en el análisis.

17.5. Clases de diseño bien creadas

El modelo de diseño se pasará a los programadores para generar código fuente, o el código se puede generar directamente desde el propio modelo si la herramienta de modelado soporta esto. Por lo tanto, las clases de diseño necesitan estar especificadas suficientemente y parte de este proceso de especificación es decidir si las clases están "bien creadas" o no.

Cuando se crea una clase de diseño, siempre es importante examinar la clase desde el punto de vista de sus clientes potenciales. ¿Como ven ellos la clase?, ¿es demasiado compleja?, ¿falta algo?, ¿está muy acoplada a otras clases?, ¿realiza lo que podría esperarse por su nombre? Éstas son consideraciones importantes y se pueden resumir en las siguientes cuatro características que debe tener una clase de diseño para que se considere bien creada.

- Completa y suficiente.
- Sencilla.
- Alta cohesión.
- Bajo acoplamiento.

17.5.1. Totalidad y suficiencia

Las operaciones públicas de una clase definen un contrato entre la clase y los clientes de esa clase. Igual que un contrato de negocios es importante que este contrato sea claro, bien definido y aceptable por todas las partes.

La característica de totalidad trata sobre proporcionar a los clientes de una clase lo que podrían esperar. Los clientes realizarán asunciones del nombre de la clase sobre el conjunto de operaciones que debería poner disponible. Para tomar un ejemplo del mundo real, si compra un coche nuevo, espera que tenga ruedas. Sucede lo mismo con las clases. Cuando nombra una clase y describe su semántica, los clientes de la clase inferirán de esta información qué operaciones se deberían encontrar disponibles. Por ejemplo, una clase `CuentaBancaria` que proporciona una operación `retirar(...)` también se esperaría que tuviera una operación `depositar(...)`. Si diseña una clase como un `CatalogoProducto`, cualquier cliente esperaría razonablemente que esta clase le permitiera añadir, eliminar y encontrar productos en el catálogo. Esta semántica está presupuesta por el nombre de la clase. Por lo tanto, esta característica trata sobre asegurarse de que las clases satisfacen todas las expectativas razonables del cliente.

La suficiencia, por otro lado, es asegurarse de que todas las operaciones de la clase están centradas en realizar la finalidad de la clase. Una clase nunca debería sorprender a un cliente. Debería contener exactamente el conjunto esperado de operaciones y nada más. Por ejemplo, un típico error de principiantes es tomar una clase sencilla como `CuentaBancaria` y luego añadir operaciones para procesar tarjetas de crédito o administrar pólizas de seguro. La suficiencia trata sobre mantener la clase de diseño lo más sencilla y centrada posible.

La regla de oro para la totalidad y suficiencia es que una clase debería hacer lo que los usuarios de la clase esperan, ni más ni menos.

17.5.2. Sencillez

Las operaciones se deberían diseñar para ofrecer un solo servicio sencillo. Una clase no debería ofrecer múltiples formas de realizar lo mismo ya que esto es confuso para los clientes de la clase y puede llevar a problemas de coherencia y cargas de mantenimiento.

Por ejemplo, si una clase `CuentaBancaria` tiene una operación sencilla para realizar un solo depósito, no debería tener una operación más compleja que realice dos o más depósitos. Esto es porque puede conseguir el mismo efecto por la aplica-

ción repetida de la operación sencilla. Su objetivo es que las clases deberían siempre poner disponible el conjunto de operaciones más sencillo y pequeño posible.

Aunque la sencillez es una buena regla, existen ocasiones cuando se puede relajar. Una razón común para relajar la restricción de sencillez es mejorar el rendimiento. Por ejemplo, si hubiera un aumento suficiente de rendimiento al realizar depósitos bancarios en un batch, en lugar de individualmente, entonces podría flexibilizar la restricción de sencillez para permitir que una clase `CuentaBancaria` tenga una operación `depositar(...)` más compleja que gestione varias transacciones a la vez. Sin embargo, su punto de partida en diseño debería ser siempre el conjunto de operaciones más sencillo posible. Solamente debería añadir complejidad al relajar la sencillez si existe un caso demostrado para hacerlo.

Éste es un punto importante. Muchas "optimizaciones" de diseño están basadas más en la fe que en hechos reales. Como tal, puede tener poco o ningún impacto sobre el rendimiento real de la aplicación. Por ejemplo, si una aplicación dedica solamente un 1 por ciento de su tiempo en una operación dada, la optimización de esa operación puede solamente agilizar la aplicación en menos de un 1 por ciento. Una regla general de utilidad es que la mayoría de las aplicaciones dedican cerca del 90 por ciento de su tiempo en el 10 por ciento de sus operaciones. Éstas son las operaciones que debe identificar y optimizar para obtener verdaderos aumentos de rendimiento. Este ajuste del rendimiento solamente se puede realizar utilizando una herramienta como jMechanic para Java (http://jmechanic.sourceforge.net) para recopilar métricas de rendimiento al ejecutar código. Ésta es claramente una tarea de implementación que podría impactar en el modelo de diseño.

17.5.3. Alta cohesión

Toda clase debería modelar un solo concepto abstracto y debería tener un conjunto de operaciones que soporten el propósito de la clase; esto es cohesión. Si una clase necesita tener muchas responsabilidades diferentes, puede crear clases "de ayuda" para implementar algunas de éstas. La clase principal puede entonces delegar responsabilidades a sus clases de ayuda.

La cohesión es una de las características más deseables de una clase. Las clases cohesivas son por lo general fáciles de entender, reutilizar y mantener. Una clase cohesiva tiene un pequeño conjunto de responsabilidades que están íntimamente relacionadas. Toda operación, atributo y asociación de la clase está específicamente diseñado para implementar este pequeño conjunto de responsabilidades.

Encontramos el modelo en la figura 17.5 en un sistema de ventas. Existe una clase `HotelBean`, una clase `CocheBean` y una clase `HotelCocheBean` (los "beans" son Enterprise JavaBeans (EJBs)). El `HotelBean` es responsable de vender estancias en hoteles, el `CocheBean` de vender alquiler de coche y el `HotelCocheBean` de vender un paquete de alquiler de coche con estancia en hotel. Claramente este modelo es erróneo desde varias perspectivas:

- Las clases están mal nombradas, `EstanciaHotel` y `AlquilerCoche` serían nombres mucho mejores.

- El sufijo "`Bean`" es innecesario ya que simplemente hace referencia a un detalle específico de implementación.
- La clase `HotelCocheBean` tiene muy mala cohesión, tiene dos responsabilidades principales (vender estancias de hotel y vender alquiler de coche) que ya se llevan a cabo por otras dos clases.
- No es ni un modelo de análisis (contiene información de diseño en los sufijos "`Bean`") ni un modelo de diseño (no está completo).

Desde una perspectiva de cohesión, `HotelBean` y `CocheBean` son bastante posibles (siempre y cuando se renombren) pero `HotelCocheBean` es absurdo.

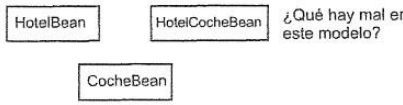

Figura 17.5.

17.5.4. Bajo acoplamiento

Una clase determinada debería estar asociada con suficientes clases para permitirle realizar sus responsabilidades y solamente debería asociar clases si existe un verdadero vínculo semántico entre ellas; esto es bajo acoplamiento.

Uno de los errores comunes de los diseñadores orientados a objetos principiantes es conectar todo de cualquier forma. De hecho, el acoplamiento es su peor enemigo en el modelado de objetos y debería ser proactivo tratando de limitar las relaciones entre clases para minimizar el acoplamiento tanto como pueda.

Un modelo de objeto muy acoplado es el equivalente de "código spaghetti" del mundo no orientado a objetos y llevará a un sistema que es incomprensible y no se puede mantener. Encontrará que los sistemas orientados a objetos muy acoplados a menudo resultan de proyectos en los que no existe actividad formal de modelado, donde el sistema simplemente tiene permitido evolucionar con el tiempo.

Si es un diseñador principiante, debe tener cuidado de no realizar conexiones entre clases simplemente porque una clase tenga cierto código que otra clase pudiera utilizar. Éste es el peor tipo de reutilización ya que sacrifica la integridad arquitectónica del sistema por un pequeño ahorro en tiempo de desarrollo. De hecho, tiene que pensar detenidamente todas las asociaciones entre clases. Muchas de las asociaciones en el modelo de diseño procederán directamente del modelo de análisis, pero existe todo un conjunto de asociaciones que están incorporadas por restricciones de implementación o por el deseo de reutilizar código. Éstas son las asociaciones que necesita examinar más detenidamente.

Por supuesto, cierto acoplamiento es bueno y deseable. Un alto acoplamiento dentro de un subsistema es generalmente aceptable ya que esto indica una alta cohesión dentro del componente. Solamente compromete la arquitectura cuando el

acoplamiento es entre subsistemas y tiene que tratar de reducir este tipo de acoplamiento.

17.6. Herencia

Cuando llega al diseño, tiene que considerar la herencia mucho más que en análisis. En análisis, solamente utilizaría herencia si hubiera una relación clara "es un" entre clases de análisis. Sin embargo, en diseño, también puede utilizar herencia de una forma táctica para reutilizar código. Ésta es una estrategia diferente ya que realmente está utilizando herencia para facilitar la implementación de una clase hijo en lugar de expresar una relación de negocio entre un padre y un hijo.

Examinamos algunas estrategias para utilizar herencia de forma eficaz en el diseño en los siguientes apartados.

17.6.1. Agregación vs. herencia

Herencia es una técnica muy potente, es un mecanismo clave para generar polimorfismos en lenguajes como Java, C# y C++. Sin embargo, los diseñadores y programadores orientados a objetos principiantes a menudo abusan de ello. Debería darse cuenta que la herencia tiene ciertas características no deseables:

- Es la forma mayor de acoplamiento posible entre dos o más clases.
- La encapsulación es débil dentro de una jerarquía de clase. Los cambios en la clase base provocan cambios en las subclases. Esto lleva a lo que se conoce como el problema "clase base frágil" donde los cambios en las clases base tienen un importante impacto sobre otras clases en el sistema.
- Es un tipo muy inflexible de relación. En todos los lenguajes orientados a objetos comúnmente utilizados, las relaciones de herencia se fijan en tiempo de ejecución. Puede modificar las jerarquías de agregación y composición en tiempo de ejecución al crear y destruir relaciones, pero las jerarquías de herencia permanecen fijas. Esto hace que sea el tipo más inflexible de relación entre clases.

El ejemplo en la figura 15.6 es una solución típica de principiante al problema de modelar roles en una organización. A primera vista, parece bastante posible pero tiene problemas. Considere este caso: el objeto `john` es de tipo `Programador` y desea promocionarlo para que sea del tipo `Director`. ¿Cómo puede hacer esto? Ha visto que no puede cambiar la clase de `john` en tiempo de ejecución y por lo tanto, la única forma en que puede conseguir la promoción es crear un nuevo objeto `Director` (denominado `john:Director`) copiar todos los datos relevantes del objeto `john:Programador` y luego eliminar el objeto `john:Programador` para mantener coherencia en la aplicación. Esto es complejo y en absoluto como funciona el mundo real.

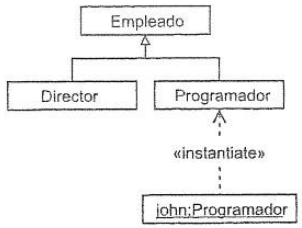

Figura 17.6.

Por supuesto, existe un error semántico fundamental en el modelo en la figura 17.6. ¿Es un empleado su trabajo, o bien es que un empleado tiene un trabajo? Esta pregunta nos lleva a la solución del problema que se muestra en la figura 17.7.

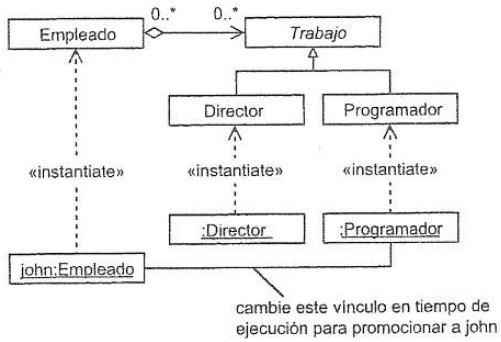

Figura 17.7.

Utilizando agregación obtiene la semántica correcta, un `Empleado` tiene un *Trabajo*. Con este modelo más flexible, los `Empleados` también pueden tener más de un *Trabajo*, si fuera necesario.

Hemos conseguido un modelo mucho más flexible y semánticamente correcto al reemplazar la herencia con agregación como el mecanismo para asignar trabajos a empleados. Existe un principio general importante aquí; las subclases siempre deberían representar "es tipo de" en lugar de "es rol desempeñado por". Cuando pensamos en la semántica de negocio de empresas, empleados y trabajos, está claro que un trabajo es un "rol desempeñado por" un empleado y no indica realmente "un tipo de" empleado.

Como tal, la herencia es la opción errónea para modelar este tipo de relación de negocio. Por otro lado, existen muchos tipos de trabajo en una empresa. Esto indica que una jerarquía de herencia de trabajos (que parten de la clase base abstracta *Trabajo*) es probablemente un buen modelo.

17.6.2. Herencia múltiple

Algunas veces es posible que desee heredar de más de un padre. Esto es herencia múltiple y no se soporta en todos los lenguajes orientados a objetos. Por ejemplo, Java y C# solamente permiten herencia sencilla. En la práctica, esta ausencia de soporte para herencia múltiple no es un problema y siempre se puede reemplazar por herencia sencilla y delegación.

Aunque la herencia múltiple, algunas veces ofrece la solución más elegante a un problema de diseño, solamente se puede utilizar si el lenguaje de implementación de destino lo soporta. Los puntos importantes sobre la herencia múltiple son los siguientes:

- Todas las clases padre implicadas deben estar separadas semánticamente. Si hay algún solapamiento en semántica entre las clases base, existe la posibilidad de interacciones imprevistas entre ellas. Esto podría llevar a un comportamiento extraño en la subclase. Decimos que las clases base deben ser todas ortogonales.
- Los principios de "es tipo de" y "sustitución" deben aplicarse entre la subclase y todas sus superclases.
- Normalmente, las superclases no deberían tener ningún padre en común. De lo contrario, tiene un ciclo en la jerarquía de herencia y puede haber múltiples rutas en donde las mismas características se podrían heredar de clases más abstractas. Los lenguajes que soportan herencia múltiple (como C++) tienen formas específicas, dependientes del lenguaje, de resolver ciclos en la jerarquía de herencia.

Una forma común de utilizar herencia múltiple de forma eficaz es la clase "mezclada". Existen clases que no son clases independientes, sino que están diseñadas específicamente para estar "mezcladas" con otras clases utilizando herencia. En la figura 17.8 la clase `Marcador` es una sencilla mezcla. Todo lo que hace es marcar un número de teléfono y por lo tanto no es demasiado útil por sí sola. Sin embargo, proporciona un paquete cohesivo de comportamiento de utilidad que se puede reutilizar por otras clases por medio de la herencia múltiple. Esta mezcla es un ejemplo de una clase de utilidad general que puede convertirse en parte de una librería de reutilización.

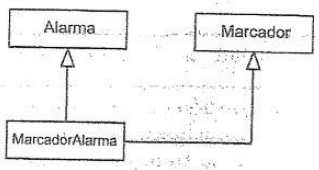

Figura 17.8.

17.6.3. Herencia vs. realización de interfaz

Con la herencia obtiene lo siguiente:

- **Interfaz:** Las operaciones públicas de las clases base.
- **Implementación:** Los atributos, relaciones, las operaciones protegidas y privadas de las clases base.

Con la realización de la interfaz (véase el capítulo 19) obtiene exactamente:

- **Una interfaz:** Un conjunto de operaciones, atributos y relaciones que no tienen implementación.

Herencia y realización de interfaz tienen algo en común ya que ambos mecanismos le permiten definir un contrato que deben implementar las subclases. Sin embargo, las dos técnicas tienen una semántica de utilización muy diferente.

Solamente necesita utilizar herencia cuando está preocupado por heredar algunos detalles de implementación (operaciones, atributos y relaciones) de una superclase. Éste es un tipo de reutilización y de hecho, en los primeros tiempos del modelado orientado a objetos se consideraba el mecanismo principal para la reutilización. Sin embargo, el mundo ha avanzado desde entonces y los diseñadores han reconocido las restricciones inaceptables que algunas veces impone la herencia y han pasado de esta utilización.

La realización de interfaz es de utilidad siempre que desee definir un contrato pero no está preocupado por heredar los detalles de implementación. Mientras que la realización de la interfaz no le proporciona reutilización de código, sí que proporciona un mecanismo para definir contratos y asegurarse de que implementar clases se adapta a esos contratos. Puesto que nada se hereda en realidad en la realización de interfaz, es mucho más flexible y robusta en cierto sentido que la herencia.

17.7. Plantillas

Hasta el momento, cuando hemos definido una clase de diseño, hemos tenido que especificar explícitamente los tipos de atributos, los tipos de retorno de todas las operaciones y los tipos de todos los parámetros de operación. Esto está bien y funciona bien en la mayoría de los casos, pero puede limitar la posibilidad de reutilizar código.

En el ejemplo en la figura 17.9 hemos definido tres clases que son arrays vinculados. Uno es un array de `int`, el siguiente es un array de `double` y el último de `String`. Cuando examina estas clases, ve que son idénticas excepto por el tipo que se almacena en el array. A pesar de esta semejanza, tenemos que definir tres clases diferentes.

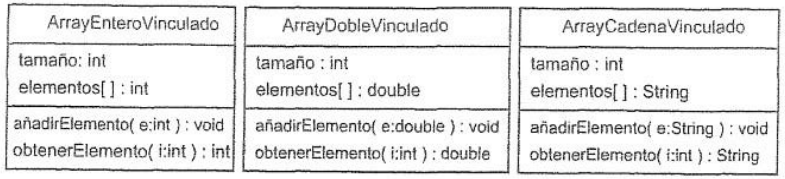

Figura 17.9.

Las plantillas le permiten parametrizar un tipo. Lo que esto significa es que en lugar de especificar los tipos reales de atributos, valores de retorno de operación y parámetros de operación, puede definir una clase en términos de marcadores de posición o parámetros. Éstos se pueden reemplazar por valores reales para crear clases nuevas.

En la figura 17.10 hemos definido la clase `ArrayVinculado` en términos de los parámetros `tipo` (que es por defecto un clasificador) y `tamaño`, que es un `int`. Observe que la plantilla especifica un valor por defecto de `10` para `tamaño`. El valor por defecto se utilizará si `tamaño` no se especifica cuando la plantilla se instancia.

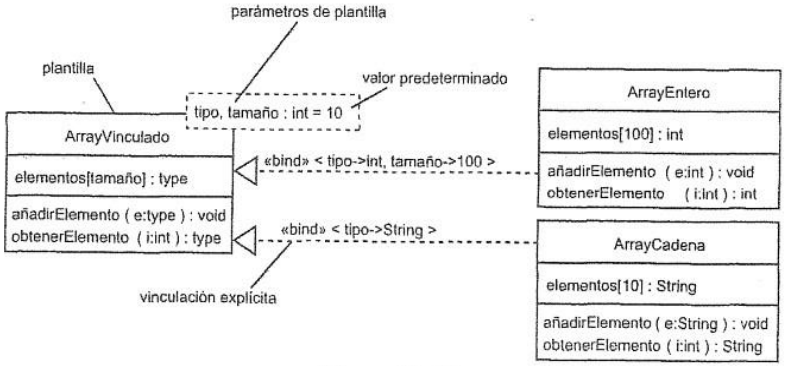

Figura 17.10.

Al vincular valores específicos a estos parámetros formales, puede crear nuevas clases; esto se conoce como instanciar plantillas. Observe que cuando instancia una plantilla, obtiene una clase y esta clase se puede instanciar para conseguir objetos.

Como muestra la figura 17.10, una plantilla se puede instanciar al utilizar una relación de realización estereotipada con `<<bind>>`, esto se conoce como vinculación explícita. Para instanciar una plantilla tiene que especificar los valores reales que se vincularán a los parámetros de plantilla y listar estos entre corchetes detrás del estereotipo `<<bind>>`. Cuando instancia la plantilla, estos valores se sustituyen por los parámetros de plantilla y esto proporciona una nueva clase. Observe la sintaxis para la vinculación de parámetro: la expresión `tipo -> int` significa que el parámetro de plantilla `tipo` se reemplaza con `int` al instanciar. Puede pensar en el símbolo `->` como "reemplazado por" o "vinculado a".

Las plantillas son claramente un mecanismo potente para reutilización; puede definir una clase como una plantilla y luego crear muchas versiones personalizadas de esta clase al vincular los parámetros de plantilla a valores reales apropiados.

En la figura 17.10 utilizamos la vinculación de dos formas. En primer lugar, vinculamos un clasificador al parámetro `tipo`. Cuando un parámetro de plantilla no tiene tipo, por defecto es un clasificador. En segundo lugar, vinculamos un valor entero al parámetro `tamaño`. Esto nos permite especificar la vinculación para el array como un parámetro de plantilla. Los nombres del parámetro de plantilla son locales para una plantilla en particular. Esto significa que si dos plantillas tienen un parámetro denominado `tipo`, es un `tipo` diferente en cada caso.

Existe una variación en la sintaxis de plantilla conocida como vinculación implícita. Aquí no utiliza una realización `<<bind>>` explícita para mostrar instanciación de plantilla, sino que vincula implícitamente al utilizar una sintaxis especial en las clases instanciadas. Para instanciar una plantilla implícitamente, simplemente liste los valores reales entre corchetes detrás del nombre de clase de plantilla como se muestra en la figura 17.11. La desventaja de este enfoque es que no puede asignar a la clase instanciada su propio nombre.

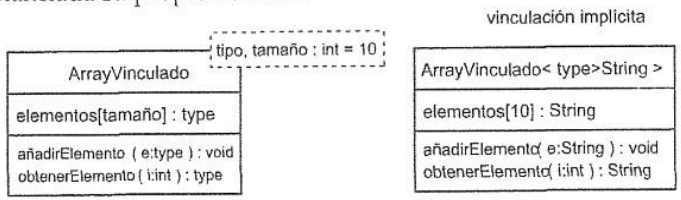

Figura 17.11.

Pensamos que es un mejor estilo utilizar la vinculación explícita de modo que las clases de instanciación de plantilla puedan tener su propio nombre descriptivo.

Mientras que las plantillas son una característica muy potente, en estos momentos C++ y Java son los únicos lenguajes orientados a objetos utilizados comúnmente que los soportan (véase la tabla 17.1). Las plantillas solamente se pueden utilizar en diseño cuando el lenguaje de implementación las soporta.

Tabla 17.1.

Lenguaje	Soporte de plantilla
Java	Sí
C++	Sí
Smalltalk	No
Python	No
Visual Basic	No
C#	No

17.8. Clases anidadas

Algunos lenguajes como Java le permiten situar una definición de clase dentro de otra definición de clase. Esto crea lo que se conoce como una clase anidada. En Java esto también se conoce como una clase interna. Una clase anidada se declara dentro del espacio de nombres de su clase externa y es sólo accesible por esa clase o por los objetos de esa clase. Solamente la clase externa o sus objetos pueden crear y utilizar instancias de la clase anidada. Las clases anidadas tienden a ser un aspecto de diseño ya que normalmente tratan cómo cierta funcionalidad se puede implementar en lugar de qué es la funcionalidad. Por ejemplo, las clases anidadas se utilizan ampliamente en la gestión de eventos de Java. El ejemplo en la figura 17.12 muestra una sencilla clase de ventana denominada `HolaMarco`. Hereda el comportamiento básico de ventana de su clase padre `Marco`. `HolaMarco` tiene una clase anidada denominada `RatónMonitor` que hereda la posibilidad de gestionar eventos de ratón de su clase padre `RatónAdaptador`.

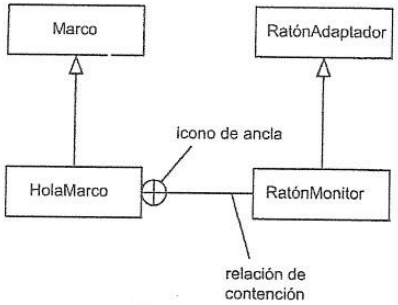

Figura 17.12.

Cada instancia `HolaMarco` utiliza una instancia `RatónMonitor` para procesar sus eventos de ratón. Para conseguir esto, la instancia `HolaMarco` debe:

- Crear una instancia de `RatónMonitor`.
- Establecer este `RatónMonitor` para que sea su evento de escucha del ratón.

Este enfoque tiene mucho sentido. El código de gestión del ratón obtiene su propia clase `RatónMonitor`, separándolo del resto del código `HolaMarco`, y `RatónMonitor` está completamente encapsulado dentro de `HolaMarco`.

17.9. ¿Qué hemos aprendido?

Las clases de diseño son los bloques de construcción del modelo de diseño. Ha aprendido lo siguiente:

- Las clases de diseño se desarrollan durante la actividad UP `Diseñar una clase`.
- Las clases de diseño son clases cuyas especificaciones se han completado hasta un nivel que se puede implementar.
- Las clases de diseño proceden de dos fuentes:
  - El ámbito del problema:
    - Una mejora de las clases de análisis.
    - Una clase de análisis se puede convertir en cero, una o más clases de diseño.
  - El ámbito de solución:
    - Librerías de clase de utilidad.
    - Middleware.
    - Librerías GUI.
    - Componentes reutilizables.
    - Detalles específicos de implementación.
- Las clases de diseño tienen especificaciones completas:
  - Conjunto completo de atributos incluido:
    - Nombre.
    - Tipo.
    - Valor por defecto cuando sea apropiado.
    - Visibilidad.
  - Operaciones:
    - Nombre.
    - Nombres y tipos de todos los parámetros.
    - Valores opcionales de parámetro si es apropiado.
    - Tipo de retorno.
    - Visibilidad
- Las clases de diseño bien formadas presentan ciertas características:
  - Las operaciones públicas de la clase definen un contrato con los clientes de la clase.
  - Completo: La clase no hace menos de lo que pudieran esperar sus clientes.
  - Suficiencia: La clase no hace más de lo que pudieran esperar sus clientes.

- Sencillez: Los servicios deberían ser sencillos y únicos.
- Alta cohesión:
  - Toda clase debería incorporar un solo concepto abstracto bien definido.
  - Todas las operaciones deberían soportar el objetivo de la clase.
- Bajo acoplamiento:
  - Una clase se debería acoplar a suficientes clases para cumplir sus responsabilidades.
  - Solamente acople dos clases cuando exista una verdadera relación semántica entre ellas.
  - Evite acoplar clases simplemente para reutilizar algo de código.
- Siempre valore una clase de diseño desde el punto de vista de los clientes de esa clase.

- Herencia:
  - Solamente utilice herencia cuando exista una relación clara "es un" entre dos clases o para reutilizar código.
  - Desventajas:
    - Es el acoplamiento mayor posible entre dos clases.
    - La encapsulación es débil dentro de una jerarquía de herencia lo que lleva al problema de "clase base frágil"; los cambios en la clase base descienden por la jerarquía.
    - Muy inflexible en la mayoría de lenguajes: la relación se decide en tiempo de compilación y se fija en tiempo de ejecución.
  - Las subclases siempre deberían representar "es tipo de" en lugar de "es rol desempeñado por"; utilice siempre agregación para representar "es rol desempeñado por".
- La herencia múltiple permite que una clase tenga más de un padre.
  - De todos los lenguajes comunes orientados a objetos, solamente C++ tiene herencia múltiple.
  - Directrices de diseño:
    - Las múltiples clases padre deben estar semánticamente separadas.
    - Debe haber una relación "es tipo de" entre una clase y todos sus padres.
    - El principio de sustitución se debe aplicar a la clase y a sus padres.
    - Los padres no deberían tener padre en común.

    - Utilice mezclas, una mezcla es una clase diseñada para mezclarse con otras en herencia múltiple; éste es un elemento seguro y potente.
- Herencia vs. realización de interfaz.
  - Herencia:
    - Obtiene interfaz: operaciones públicas.
    - Obtiene implementación: Los atributos, asociaciones y miembros protegidos y privados.
  - Realización de interfaz: Solamente obtiene la interfaz.
  - Utilice herencia cuando quiera heredar alguna implementación.
  - Utilice realización de interfaz cuando quiera definir un contrato.
- Plantillas.
  - De todos los lenguajes orientados a objetos comúnmente utilizados, solamente C++ y Java soportan plantillas.
  - Las plantillas le permiten "parametrizar" un tipo: cree una plantilla al definir un tipo en términos de parámetros formales, e instancie la plantilla al vincular valores específicos para los parámetros.
    - La vinculación explícita utiliza una dependencia estereotipada <<bind>>:
      - Muestre los valores en la relación.
      - Puede nombrar cada instanciación de plantilla.
    - Vinculación implícita:
      - Especifique los valores reales de la clase dentro de (<>).
      - No puede nombrar las instanciaciones de plantilla, los nombres se construyen a partir del nombre de plantilla y la lista de argumentos.
- Clases anidadas:
  - Define una clase dentro de otra clase.
  - La clase anidada existe en el espacio de nombres de la clase exterior; solamente la clase exterior puede crear y utilizar instancias de la clase anidada.
  - Las clases anidadas se conocen como clases internas en Java y se utilizan ampliamente para gestionar eventos en clases GUI.