CAPITULO 7. DISEÑAR CLASES

7.1 Introducción

Es posible implementar una aplicación y lograr que realice su tarea mediante un diseño de clases mal logrado. El problema surge, típicamente, cuando un programador de mantenimiento quiere hacer algunos cambios en una aplicación existente. Si por ejemplo, el programador intenta solucionar un fallo o quiere agregar funcionalidad a un programa, una tarea que debiera ser fácil y obvia con un buen diseño de clases, podría resultar muy difícil de manejar y consumir una gran cantidad de trabajo si las clases están mal diseñadas.

En las aplicaciones grandes, este efecto ya ocurre durante la implementación original. Si la implementación comienza con una mala estructura, su finalización puede volverse muy compleja y puede que no se termine de completar el programa, o que contenga fallos o que su construcción tome más tiempo de lo necesario.

El diseño de mala calidad no es, generalmente, consecuencia de tener un problema difícil para resolver. La mala calidad del *diseño* tiene más que ver con las decisiones que se toman cuando se resuelve un problema en especial. No podemos usar el argumento de que no había otra manera de resolver el problema como una excusa para un diseño de mala calidad.

7.3 Introducción al acoplamiento y a la cohesión

El término **acoplamiento** describe la interconectividad de las clases. Nos esforzamos por lograr aclopamiento débil en un sistema, es decir, un sistema en el que cada clase es altamente independiente y se comunica con otras clases mediante una pequeña interfaz bien definida.

El grado de acoplamiento determina el grado de dificultad de realizar modificaciones en una aplicación. En una estructura de clases fuertemente acopladas, un cambio en una clase hace necesario también cambiar otras varias clases. Este hecho es el que tratamos de evitar porque el efecto de hacer un pequeño cambio puede rápidamente propagarse a la aplicación completa. Además, encontrar todos los lugares en que resulta necesario hacer los cambios y realmente llevar a cabo estos cambios puede ser dificultoso y consumir demasiado tiempo.

Por otro lado, en un sistema débilmente acoplado, podemos con frecuencia modificar una clase sin tener que realizar cambios en ninguna otra y la aplicación continúa funcionando.

El término **cohesión** describe cuánto se ajusta una unidad de código a una tarea lógica o a una entidad. En un sistema altamente cohesivo cada unidad de código (método, clase o módulo) es responsable de una tarea bien definida o de una entidad. Un diseño de clases de buena calidad exhibe un alto grado de cohesión. La cohesión es relevante para unidades formadas por una sola clase y para métodos individuales, así como para módulos o paquetes.

Idealmente, una unidad de código debiera ser responsable de una tarea cohesiva, es decir, una tarea que pueda ser vista como una unidad lógica. Un método debiera implementar una operación lógica y una clase debiera representar un tipo de entidad. La razón principal que subyace al principio de cohesión es la reusabilidad: si un método de una clase es responsable de una única cosa bien definida es más probable que pueda ser usado nuevamente en un contexto diferente. Una ventaja complementaria, consecuencia de este principio, es que, cuando se requiere un cambio de un aspecto de una aplicación, probablemente encontremos todas las piezas de código relevantes ubicadas en la misma unidad.

7.4 Duplicación de código

La **duplicación de código**, es decir, tener el mismo segmento de código en una aplicación más de una vez es una señal de mal diseño y debe ser evitada. Generalmente, la duplicación de código es un síntoma de mala cohesión.

7.5 Hacer extensiones

Ampliar el código de una aplicación sin un buen nivel de cohesión implica tener que realizar modificaciones en más sitios.

7.6 Acoplamiento

Idealmente, cuando sólo se cambia la implementación de una clase, las restantes clases no debieran verse afectadas por el cambio. Este sería un caso de acoplamiento débil. En cambio un alto acoplamiento obliga a modificar varias clases.

7.6.1 Usar encapsulamiento para reducir el acoplamiento

El uso de campos públicos expuestos en la interfaz de una clase no solo aporta la información que contiene la clase, sino también cómo se almacena exactamente la información de esos campos. Esto rompe uno de los principios fundamentales del diseño de clases de buena calidad: el *encapsulamiento*.

Una pauta para el encapsulamiento (ocultar la información de la implementación) sugiere que solamente la información sobre lo *que* puede hacer una clase debe estar visible desde el exterior, pero no *cómo* lo hace. Esto tiene una gran ventaja: si ninguna otra clase conoce cómo está almacenada nuestra información entonces podemos cambiar fácilmente la forma de almacenarla sin romper otras clases.

El **encapsulamiento apropiado** en las clases reduce el acoplamiento y por lo tanto lleva a un mejor diseño.

Podemos reforzar esta separación del *qué* y del *cómo* declarando los campos como privados y usando un método de acceso para acceder a ellos. Así en cualquier momento se puede cambiar la representación interna de los datos sin que se tenga que modificar la interfaz, ni, por lo tanto, el resto de clases que la usen. Es importante notar que jamás podemos desacoplar completamente las clases en una aplicación, de lo contrario, no podrían interactuar entre ellos objetos de diferentes clases. Más bien tratamos de mantener un grado de acoplamiento tan bajo como sea posible.

7.7 Diseño dirigido por responsabilidades

El encapsulamiento no es el único factor que influye en el grado de acoplamiento, otro aspecto se conoce como *diseño dirigido por responsabilidades*. **Diseño dirigido por responsabilidades** es el proceso de diseñar clases asignando responsabilidades bien definidas a cada una. Este proceso puede usarse para determinar las clases que deben implementar una parte de cierta función de una aplicación. Expresa la idea de que cada clase será responsable de manejar sus propios datos. Un buen diseño dirigido por responsabilidades influye en el grado de acoplamiento y por consiguiente, también influye en la facilidad con que una aplicación puede ser modificada o extendida.

7.8 Localización de cambios

Uno de los principales objetivos de un diseño de clases de buena calidad es la **localización de los cambios**: las modificaciones en una clase debieran tener efectos mínimos sobre las otras clases.

Apuntamos a crear un diseño de clases que facilite las modificaciones posteriores mediante la ubicación de los efectos de un cambio determinado. Idealmente, debe cambiarse una única clase para realizar una modificación. Algunas veces, es necesario cambiar varias clases, pero apuntamos a que el cambio afecte a la menor cantidad de clases posible. Además, los cambios que requieran las otras clases debieran ser obvios, fáciles de detectar y fáciles de llevar adelante.

En los proyectos grandes, logramos este objetivo siguiendo las reglas de diseño de buena calidad tales como usar diseño dirigido por responsabilidades y apuntar a un bajo acoplamiento y a una alta cohesión.

7.9 Acoplamiento implícito

Los campos públicos deben evitarse. Sin embargo, existe aún una forma peor de acoplamiento: el *acoplamiento implícito*.

El acoplamiento implícito es una situación en la que una clase depende de la información interna de otra pero esta dependencia no es inmediatamente obvia. El denso acoplamiento en el caso de los campos públicos no era bueno, pero por lo menos era obvio. Si cambiamos los campos públicos en una clase y nos olvidamos de otra, la aplicación no compilará más y el compilador indicará el problema. En los casos de acoplamiento implícito, el omitir un cambio necesario puede no ser detectado. Un buen diseño de clases evitará esta forma de acoplamiento siguiendo la regla de diseño dirigido por responsabilidades

Si nos fijamos en el diagrama de clases de una aplicación, las flechas en el diagrama son un buen primer indicador del grado de intensidad del acoplamiento de un programa: cuantas más flechas, más acoplamiento. Como una aproximación a un buen diseño de clases podemos apuntar a crear diagramas con pocas flechas.

7.10 Pensar en futuro

Una característica de un buen diseñador de software es la habilidad de pensar en el futuro. ¿Qué podría cambiar? ¿Qué podemos asumir con seguridad que permanecerá sin cambios durante la vida del programa?

Encapsular toda la información de la interfaz de usuario en una sola clase o en un conjunto de clases claramente definido forma parte de un buen diseño.

7.11 Cohesión

El principio de cohesión puede aplicarse a clases y a métodos: las clases deben mostrar un alto grado de cohesión y lo mismo ocurre con los métodos.

7.11.1 Cohesión de métodos

Método cohesivo: un método cohesivo es responsable de una y sólo una tarea bien definida. Es más fácil de comprender lo que hace un segmente de código y realizar modificaciones si se usan métodos breves y cohesivos. Estas características representan una ayuda valiosa para un programador de mantenimiento.

7.11.2 Cohesión de clases

La regla de cohesión de clases establece que cada clase debe representar una única entidad bien definida en el dominio del problema. Una clase cohesiva representa una única entidad bien definida.

7.11.3 Cohesión para la legibilidad

Hay varias maneras en que un diseño se ve beneficiado por la alta cohesión. Las dos más importantes son la *legibilidad* y la *reusabilidad.* Un programador de mantenimiento fácilmente reconocerá por dónde comenzar a leer el código si necesita realizar un cambio en clases bien cohesionadas

7.11.4 Cohesión para la reusabilidad

La segunda gran ventaja de la cohesión es el alto potencial para la reutilización.

7.12 Refactorización

La **refactorización** es la actividad de reestructurar un diseño existente para mantener un buen diseño de clases cuando se modifica o se extiende una aplicación.

Es frecuente que, durante el tiempo de vida de una aplicación, se le vaya agregando funcionalidad. Un efecto común que se produce de manera colateral es el lento crecimiento de la longitud de los métodos y de las clases.

El agregado de código en reiteradas ocasiones suele tener como consecuencia la disminución del grado de cohesión. Es muy probable que si se agrega más y más código a un método o a una clase, llegue un momento en el que representará más de una tarea claramente definida o más de una entidad.

La refactorización consiste justamente en repensar y rediseñar las estructuras de las clases y de los métodos. El efecto más común es que las clases se abran en dos o que los métodos se dividan en dos o más métodos; la refactorización también incluye la unión de clases o de métodos que da por resultado una sola clase o un solo método, pero este caso es menos frecuente.

7.12.1 Refactorización y prueba

Cuando algo se modifica existe la posibilidad de que se introduzcan errores, por lo tanto, es importante proceder cautelosamente, y antes de llevar a cabo la refactorización debemos asegurarnos de que exista un conjunto de pruebas para la versión actual del programa. Si las pruebas no existen, es prioritario crear algunas pruebas que se adecuen para implementar pruebas regresivas sobre la versión rediseñada. La refactorización debe comenzar sólo cuando existen las pruebas. Idealmente, la refactorización debe seguir dos pasos:

- El primer paso es repensar el diseño de modo que mantenga la misma funcionalidad que la versión original. En otras palabras, reestructuramos el código para mejorar su calidad, no para cambiar o aumentar su funcionalidad. Una vez que este paso está completo, se deben ejecutar las pruebas regresivas para asegurarse de que no se hayan introducido errores no deseados.

- El segundo paso se puede dar, únicamente, una vez que se ha restablecido la funcionalidad básica en la versión refactorizada. En ese momento estamos en una posición segura como para mejorar el programa. Una vez que se ha finalizado con la refactorización, por supuesto que será necesario ejecutar las pruebas en la nueva versión.

La implementación de varios cambios al mismo tiempo (repensar y agregar nuevas características) hace que se vuelva más difícil ubicar la fuente de los problemas, cuando estos ocurran.

Una buena refactorización es tanto una manera de pensar como un conjunto de habilidades técnicas. Mientras realizamos cambios y extensiones en las aplicaciones, regularmente nos debemos preguntar si el diseño original aún representa la mejor

solución. A medida que cambia la funcionalidad, también cambian los argumentos a favor o en contra sobre ciertos di seños. Lo que fue un buen diseño para una aplicación simple podría dejar de serlo cuando se agregan algunas extensiones.

Reconocer estos cambios y realizar efectivamente estas modificaciones de refactorización en el código, generalmente ahorra una gran cantidad de tiempo y de esfuerzo al final. Cuanto antes limpiemos nuestro diseño, más trabajo ahorraremos.

Debemos estar preparados para *refactorizar* métodos (convertir una secuencia de sentencias del cuerpo de un método existente en un método nuevo e independiente) y clases (tomar partes de una clase y crear una nueva clase a partir de ella). Considerar regularmente la refactorización mantiene nuestro diseño de clases limpio y finalmente, nos ahorra trabajo.

7.13 Refactorización para independizarse del idioma

Si queremos que el programa sea independiente del idioma, la situación ideal sería que el texto real de las palabras se almacene en un único lugar del código y que en todas las restantes partes se haga referencia a ellas de manera independiente del idioma. Una característica del lenguaje de programación que torna posible esta solución está dada por los *tipos enumerados* o *enumeraciones.*

7.13.1 Tipos Enumerados

En su forma más simple, una definición de un tipo enumerado consiste en una envoltura exterior que utiliza la palabra enum en lugar de la palabra class, y un cuerpo que es simplemente una lista de nombres de variables que denotan el conjunto de valores que pertenece a este tipo. Por convención, los nombres de estas variables se escriben en mayúsculas. Nunca creamos objetos de un tipo enumerado. En efecto, cada nombre dentro de la definición del tipo representa una única instancia de un tipo enumerado que ya se ha creado para usarla.

```
public enum <NombreEnum>{
    ENUM_OPC1, ENUM_OPC2, ENUM_OPC3, ENUM_OPC4;
}
```

Cada opción representa una instancia, a las que nos referimos de la siguiente manera: NombreEnum.ENUM_OPC1**,** NombreEnum.ENUM_OPC2, etc. A pesar de la simplicidad de su definición, los valores del tipo enumerado son objetos propiamente dichos, por lo tanto, no son iguales que los enteros.

Java permite que las definiciones de los tipos enumerados contengan mucho más que una lista de valores de tipos:

```
public enum <NombreEnum>{
    ENUM_OPC1(“valor1”), ENUM_OPC2(“valor2”), ENUM_OPC3(“valor3”), ENUM_OPC4(“valor4”);
    private String valor;
    NombreEnum(String valor){
        this.valor = valor;
    }
    public String getValor(){
        return this.valor;
    }
}
```

En la declaración de cada tipo del enum se añade un parámetro. La definición del tipo incluye un constructor que no tiene la palabra public en su encabezado. Los constructores de los tipos enumerados nunca son públicos porque no podemos crear instancias de ellos. El parámetro asociado a cada valor se pasa mediante el parámetro del constructor. La definición del tipo incluye un campo, valor. El constructor almacena la cadena pasada como parámetro en este campo. Cada tipo enumerado define un método values que devuelve un array que contiene todos los valores del tipo. NombreEnum.values()

7.14 Pautas de diseño

Un método es demasiado largo si hace más de una tarea lógica. Una clase es demasiado compleja si representa más de una entidad lógica.

7.15 Ejecutar un programa fuera de BlueJ

7.15.1 Métodos de clase

Los *métodos de clase* que en Java se conocen también como ***métodos estáticos***. Hasta ahora, todos lo métodos que hemos visto han sido *métodos de instancia:* se invocan sobre una instancia de una clase. Lo que distingue a los métodos de clase de los métodos de instancia es que los métodos de clase pueden ser invocados sin tener una instancia, alcanza con tener la clase.

Los métodos de clase están relacionados conceptualmente y usan una sintaxis relacionada con las variables de clase (la palabra clave en Java es static). Así como las variables de clase pertenecen a la clase antes que a una instancia, lo mismo ocurre con los métodos de clase. Un método de clase se define agregando la palabra clave static antes del nombre del tipo en la signatura del método:

```
public static int getNumeroDeDiasDeEsteMes ()
{
}
```

Estos métodos pueden ser invocados utilizando la notación usual de punto, especificando el nombre de la clase en que está definido seguido del punto y luego del nombre del método. Si, por ejemplo, el método anterior está declarado en una clase de nombre Calendario, la siguiente sentencia lo invoca: int dias = Calendario. getNumeroDeDiasDeEstemes () ;

7.15.2 El método main

Si queremos iniciar una aplicación Java fuera del entorno BlueJ necesitamos usar un método de clase. En BlueJ, típicamente creamos un objeto e invocamos uno de sus métodos, pero fuera de este entorno una aplicación comienza sin que exista ningún objeto. Las clases son las únicas cosas que tenemos inicialmente, por lo que el primer método que será invocado debe ser un método de clase.

La definición de Java para iniciar aplicaciones es bastante simple: el usuario especifica la clase que será iniciada y el sistema Java luego invocará un método denominado main ubicado dentro de dicha clase. Este método debe tener una signatura específica. Si no existe tal método en esa clase se informa un error. (Apéndice E)

7.15.3 Limitaciones de los métodos de clase

Dado que los métodos de clase están asociados con una clase antes que con una instancia, tienen dos limitaciones importantes. La primera limitación es que un método de clase no podrá acceder a ningún campo de instancia definido en la clase. Esto es lógico ya que los campos de instancia están asociados con objetos individuales. En cambio, los métodos de clase tienen el acceso restringido a las variables de clase de sus propias clases. La segunda limitación es como la primera: un método de clase no puede invocar a un método de instancia de la clase. Un método de clase sólo puede invocar a otros métodos de clase definidos en su propia clase.