Programación orientada a objetos

Capítulo 7

Diseñar clases

Tema 7. Diseñar clases. Semana 7	
1- Acoplamiento y cohesión 2- Uso de la encapsulación para reducir el acoplamiento 3- Diseño dirigido por responsabilidades 4- Acoplamiento implícito 5- Refactorización 6- Refactorización para independizarse del idioma 7- Pautas de diseño 8- Ejecución de una aplicación fuera de BlueJ	1- Estudiar el capítulo 7 del libro base para la Unidad Didáctica I 2- Leer el Apéndice E del libro base para la Unidad Didáctica I 3- Realizar los ejercicios en el entorno BlueJ sugeridos en el libro base 4- Revisar el acoplamiento y cohesión de la solución propuesta en la práctica

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 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 **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.

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

Método cohesivo: un método cohesivo es responsable de una y sólo una tarea bien definida.

Clase cohesiva: una clase cohesiva representa una única entidad bien definida.

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.

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 puede 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();
```

Observe que antes del punto se usa el nombre de la clase, no se ha creado ningún objeto

El método “main”

En Java, este problema se resuelve usando una convención: cuando se inicia un programa Java, el nombre de la clase se especifica como un parámetro del comando de inicio y el nombre del método es siempre el mismo, el nombre de este método es «main». Por ejemplo, considere el siguiente comando ingresado en una línea de comando, como si fuera un comando de Windows o de una terminal Unix:

```
java Juego
```

El comando `java` inicia la máquina virtual de Java, que forma parte del kit de desarrollo de Java (SDK) y que debe estar instalado en su sistema. `Juego` es el nombre de la clase que queremos iniciar.

Luego, el sistema Java buscará un método en la clase `Juego` cuya signatura coincida exactamente con la siguiente:

```
public static void main(String[] args)
```

main

En general, el método `main` debe hacer exactamente lo que se hizo interactivamente para iniciar la misma aplicación en BlueJ. Por ejemplo, si para iniciar la aplicación en BlueJ se creó un objeto de la clase `Juego` y se invocó el método de nombre `start`, en el método `main` de la clase `Juego` deberían agregarse las siguientes sentencias:

```
public static void main (String[] args)
{
  Juego juego = new Juego();
  juego.start();
}
```

Desarrollar fuera del BlueJ

Si no quiere solamente ejecutar programas, sino que también quiere desarrollarlos fuera del entorno BlueJ, necesitará editar y compilar las clases. El código de una clase se almacena en un archivo de extensión «.java»; por ejemplo, la clase `Juego` se almacena en un archivo de nombre `Juego.java`. Los archivos fuente pueden editarse con cualquier editor de textos. Existen muchos editores de textos libres o muy baratos. Algunos, como el *Notepad* o el *WordPad* se distribuyen con Windows, pero si en realidad quiere usar un editor para hacer algo más que una prueba rápida, querrá obtener uno mejor. Sin embargo, sea cuidadoso con los procesadores de texto: generalmente los procesadores de texto no graban en formato de texto plano y Java no podrá leerlos.

Los archivos fuente pueden compilarse desde una línea de comando usando el compilador Java que se incluye en el JDK y que se invoca mediante el comando `javac`. Para compilar un archivo fuente de nombre `Juego.java` use el comando

```
javac Juego.java
```

Este comando compilará la clase `Juego` y cualquier otra clase que dependa de ella; creará un archivo denominado `Juego.class` que contiene el código que puede ser ejecutado mediante la máquina virtual de Java. Para ejecutar este archivo use el comando

```
java Juego
```

Observe que este comando no incluye la extensión del archivo «.class».

Términos introducidos en este capítulo

duplicación de código, acoplamiento, cohesión, encapsulamiento, diseño dirigido por responsabilidades, acoplamiento implícito, refactorización, método de clase

Resumen de conceptos

- **acoplamiento** El término acoplamiento describe las interconexiones de las clases. Fomentamos el bajo acoplamiento de un sistema, es decir, un sistema en donde cada clase es bastante independiente y se comunica con otras clases mediante una interfaz pequeña y bien definida.

- **cohesión** La expresión cohesión describe la exactitud con que una unidad de código encaja con una tarea lógica o con una entidad. En un sistema altamente cohesivo cada unidad de código (método, clase o módulo) es responsable de una tarea o entidad bien definida. Un buen diseño de clases exhibe un alto grado de cohesión.

- **duplicación de código** La duplicación de código (tener el mismo segmento de código en una aplicación más de una vez) es una señal de mal diseño. Debe evitarse.

- **Encapsulamiento** El encapsulamiento apropiado de las clases reduce el acoplamiento y conduce a un mejor diseño.

- **diseño dirigido por responsabilidades** Es el proceso de diseñar clases asignando a cada clase responsabilidades bien definidas. Este proceso puede usarse para determinar las clases que implementarán cada parte de una función de una aplicación.

- **localizar cambios** Uno de los principales objetivos de un buen diseño de clases es la localización de los cambios: el hacer cambios en una clase debe tener efectos mínimos en las otras clases.
- **método cohesivo** Un método cohesivo es responsable de una y sólo una tarea bien definida.
- **clase cohesiva** Una clase cohesiva representa una entidad bien definida.
- **refactorización** La refactorización es la actividad de reestructurar un diseño existente para mantener un buen diseño de clases cuando la aplicación se modifica o se extiende.