Programación orientada a objetos

Capítulo 8

Diseño de clases

Tutor: Manuel Fernández Barcell
Centro Asociado de Cádiz
http://prof.mfbarcell.es

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

Tema 6. Se discuten cuestiones relacionadas con la división del problema a tratar con el fin de obtener un conjunto de clases adecuado que facilite su implementación. Se presentan técnicas orientadas a un diseño de clases de buena calidad que incluyen conceptos tales como el diseño dirigido por responsabilidades, acoplamiento, cohesión y refactorización.

Tema 6:

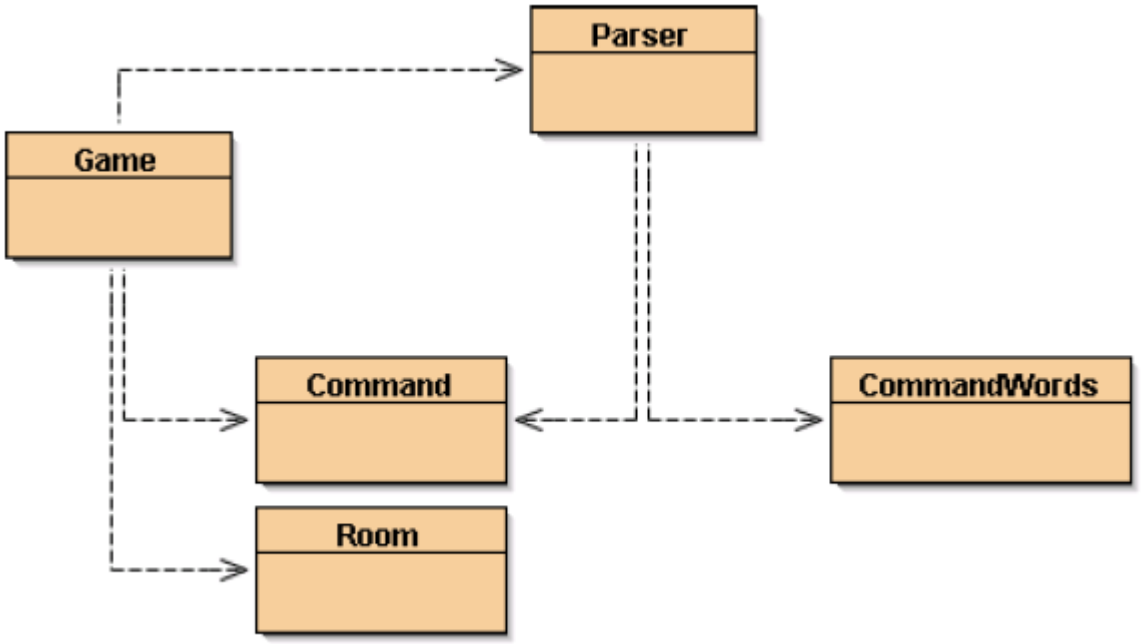
1- Ser capaz de definir un esquema de clases dado un problema, evitando el acoplamiento y la duplicación de código.

6.2 Juego world-of-zuul “malo”

- Juego de aventuras
  - Veremos cómo ir mejorando el diseño inicial
- *Colossal Cave Adventure* (Will Crowder, ’70)
  - Juego que trata de encontrar un camino a través de un complejo sistema de cuevas, ubicar el tesoro escondido, usar palabras secretas y otros misterios, todo ello con el propósito de obtener el máximo número de puntos

Ejemplo de código no muy ejemplar

- chapter06/zuul-bad



Clases de zuul

- Clase Game
  - Clase principal que inicia el juego y permite empezar un ciclo de lectura y ejecución de comandos
  - También tiene el código que implementa cada comando
- Clase CommandWords
  - Define todas las palabras de posibles comandos
- Clase Command
  - Representa un comando introducido por el usuario y permite controlar si es válido y considerar por separado la primera y segunda palabras
- Clase Room
  - Representa una habitación que puede tener varias salidas para ir a otras habitaciones
- Clase Parser
  - Se encarga de interpretar la entrada del usuario y trata de crear los objetos Command correspondientes

Diseño de clases

- Características de un buen programa
  - Fácil de entender
  - Mantenible
  - Reutilizable
- Hay principios de diseño que ayudan en el diseño de las clases y la estructura del programa:
  - Acoplamiento y cohesión
  - Diseño dirigido por responsabilidades
  - Refactorización

8.3 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.

- ACOPLAMIENTO
  - Interconectividad de las clases
    - Si dos clases dependen mutuamente en muchos detalles se dice que están **fuertemente acopladas**
  - Una estructura de clases **fuertemente acopladas**, un cambio en una clase hace necesario también el cambio en otras varias clases
    - Esto hay que evitarlo
    - Debemos lograr un acoplamiento débil
  - Una **buena** propiedad es el **acoplamiento débil**
    - Que las clases sean lo más independientes posibles
    - Y que se comuniquen a través de pequeñas interfaces bien definidas

Cohesión

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.

- Cohesión
  - Se relaciona con el número y la diversidad de tareas de las que es responsable una sola unidad de la aplicación
    - Cuánto se ajusta una unidad de código a una tarea lógica o a una entidad
      - El número y diversidad de tareas que se asignan a una unidad
- Una **buena** propiedad es una **alto grado de cohesión**
  - Cada unidad de código (método, clase o módulo) es responsable de una tarea bien definida o de una entidad
    - Un método debería implementar una operación lógica
    - Una clase debiera representar a un tipo de entidad
- La razón principal de la cohesión es la reusabilidad
  - Si es responsable de algo bien definido, es probable que pueda ser usado en un contexto diferente

Calidad de diseño

- Un sistema fuertemente acoplado (esto es, con muchas dependencias entre clases)
  - ¿Será más o menos fácil de modificar?
  - ¿Afectarán sus cambios a otras clases?
  - ¿Mejorará su mantenibilidad?
- Si un método es responsable de una única cosa bien definida seguramente podrá ser usado nuevamente en un contexto diferente
  - Más fácil entender lo que hace
  - Asignar nombres más descriptivos
  - Mayor reutilización
- Cohesión de métodos
  - Cada método debe ser responsable de una y solo una tarea bien definida
- Cohesión de clases
  - Cada clase debe representar una sola entidad bien definida

8.4 Duplicación de código

- La duplicación de código es síntoma de mala cohesión
- Es un indicador de mala calidad de diseño
  - Hace más difícil el mantenimiento del código
  - Puede inducir a la introducción de errores durante el mantenimiento
    - Inconsistencias
    - Se cambia el código en un sitio pero no en otro similar
- Ejemplo: métodos *printWelcome()* y *goRoom()* en la clase *Game*
  - Ambos métodos imprimen información pero ninguno puede llamar al otro porque cada uno de ellos, además, hace otras cosas
- ¿La solución?
  - Refactorizar código en un método: *private void printLocation()*

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.

```
private void printLocationInfo()
{
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if(currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if(currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if(currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

Exercise 6.5 Implement and use a separate `printLocationInfo` method in your project, as discussed in this section. Test your changes.

8.5 Hacer ampliaciones

- Supóngase que se quiere añadir la posibilidad de ir en otras direcciones, por ejemplo:
  - *go up* y *go down*
- *¿Qué clases y métodos habrá que tocar?*
  - *Room*
  - *Game*
    - *Están fuertemente acopladas*
- Una solución al acoplamiento fuerte:
  - El **encapsulamiento**

8.6 Acoplamiento

- Usar encapsulamiento para reducir el acoplamiento
  - Los campos de las clases deben ser ”**privados**”
- Encapsulación
  - Ocultar la información de la implementación
    - Solo lo ***qué*** hace una clase debe ser visible, no ***cómo*** lo hace
  - Hay que declarar los campos como privados y usar un método de acceso para acceder a ellos

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

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;
…
}
```

// utilización:

```
// Try to leave current room.
        Room nextRoom = null;
        if(direction.equals("north")) {
            nextRoom = currentRoom.northExit;
        }
        if(direction.equals("east")) {
            nextRoom = currentRoom.eastExit;
        }
        if(direction.equals("south")) {
            nextRoom = currentRoom.southExit;
        }
        if(direction.equals("west")) {
            nextRoom = currentRoom.westExit;
        }
```

```
public class Room // encapsulando
{
   private String description;
   private Room northExit;
   private Room southExit;
   private Room eastExit;
   private Room westExit;
…
   public Room getExit(String direction) {
        if (direction.equals("north")) return northExit;
        if (direction.equals("south")) return southExit;
        if (direction.equals("east")) return eastExit;
        if (direction.equals("west")) return westExit;
   }
}
```

// utilización:

```
 // Try to leave current room.
        Room nextRoom = currentRoom.getExit(direction);
```

8.7 Diseño dirigido por responsabilidades

Concepto

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.

- Asignar responsabilidades bien definidas a cada clase
  - ¿En qué clase habrá que poner un nuevo método?
  - Cada clase es responsable de gestionar sus propios datos
- La clase que posee unos datos tiene que ser responsable de procesarlos
  - Un buen diseño dirigido por responsabilidades conduce a reducir el grado de acoplamiento

8.8 Localización de cambios

Concepto

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.

- Uno de los objetivos de un buen diseño de clases es facilitar la localización de los cambios
  - Las modificaciones en una clase deberían tener efectos mínimos sobre las otras clases
  - Por eso hay que evitar definir campos públicos y definir claramente la interfaz de uso de la clase como un conjunto de métodos públicos
  - Los métodos públicos de la clase no deberían cambiar
    - Si acaso añadir otros nuevos

8.9 Acoplamiento implícito

- Cuando una clase depende de la información interna de otra pero esta dependencia no es inmediatamente obvia
- El uso de campos públicos puede ser obvio
  - Si se cambian se producirán errores al compilar las clases que los usen
- Pero hay casos más difíciles de detectar
  - Ejemplo: ¿qué pasa si se quieren añadir más comandos en el juego?
  - Ejercicio:
    - Agrega el comando “eat” para que cuando se ejecute aparezca el texto “*I have eaten and I am not more hungry*”

8.10 Planificación por adelantado

- Al diseñar clases hay que pensar cómo podrían ampliarse en el futuro
  - ¿Qué podrá cambiar?
  - La encapsulación ayuda mucho

8.11 Cohesión

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.

- Cohesión de Métodos
  - Cada método debería ser responsable de una y sólo una tarea bien definida
- Cohesión de Clases
  - Cada clase debe representar una única entidad bien definida en el dominio del problema

- La cohesión es importante para la “legibilidad” y la reusabilidad

8.12 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.

- Al ir modificando las clases normalmente se va añadiendo más y más código
  - Las clases y los métodos suelen crecer
  - Habrá que refactorizar para mantener buenos niveles de cohesión y bajo acoplamiento
- La refactorización consiste **en repensar y rediseñar las estructuras de las clases y los métodos**
- Métodos:
  - Convertir una secuencia de sentencias del cuerpo de un método en un método nuevo
    - Un método es demasiado largo si hace más de una tarea lógica
- Clases:
  - Tomar partes de una clase y crear una nueva clase a partir de ella
    - Una clase es demasiado compleja si representa más de una entidad lógica
- Pruebas
  - Es importante realizar pruebas que garanticen el buen funcionamiento de los cambios

8.13.1 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

```
/**
* Representación para todas las palabras comando válidas
del juego.
*
* @author Michael Kölling and David J. Barnes
* @version 2006.03.30
*/
public enum PalabraComando
{
      // Un valor para cada palabra comando, más una para
los comandos no
      // reconocidos.
      IR, SALIR, AYUDA, DESCONOCIDA;
}
```

```
if (palabraComando.equals("ayuda")) {
   imprimirAyuda();
}
```

Si palabraComando se declara de tipo PalabraComando en lugar de tipo String, entonces estas líneas se pueden rescribir así:

```
if (palabraComando == PalabraComando.AYUDA)) {
   imprimirAyuda();
}
```

```
public PalabrasComando()
{
     comandosValidos = new HasMap<String, PalabraComando>;
     comandosValidos.put("ir", PalabraComando.IR);
     comandosValidos.put("ayuda", PalabraComando.AYUDA);
     comandosValidos.put("salir", PalabraComando.SALIR);
}
```

El comando escrito por un usuario ahora puede ser fácilmente convertido a su correspondiente valor de tipo enumerado.

Ejemplo de enumerado

```
public enum Demarcacion
{
    PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO
}
```

```
public enum Demarcacion{PORTERO, DEFENSA, CENTROCAMPISTA, DELANTERO}
Demarcacion delantero = Demarcacion.DELANTERO;    // Instancia de un enum de la clase Demarcación
delantero.name();    // Devuelve un String con el nombre de la constante (DELANTERO)
delantero.toString();    // Devuelve un String con el nombre de la constante (DELANTERO)
delantero.ordinal();    // Devuelve un entero con la posición del enum según está declarada (3).
delantero.compareTo(Enum otro);    // Compara el enum con el parámetro según el orden en el que están declarados lo enum
Demarcacion.values();    // Devuelve un array que contiene todos los enum
```

```
delantero.name()= DELANTERO
defensa.toString()= DEFENSA
delantero.ordinal()= 3
delantero.compareTo(defensa)= 2
delantero.compareTo(delantero)= 0
PORTERO - DEFENSA - CENTROCAMPISTA - DELANTERO
```

Instrucción switch

```
switch (commandWord) {
    case UNKNOWN:
        System.out.println("I don't know what you mean...");
        break;
    case HELP:
        printHelp();
        break;
    case GO:
        goRoom(command);
        break;
    case QUIT:
        wantToQuit = quit(command);
        break;
}
```

```java
/**
 * Representations for all the valid command words for the game
 * along with a string in a particular language.
 *
 * @author  Michael Kölling and David J. Barnes
 * @version 2016.02.29
 */
public enum CommandWord
{
    // A value for each command word along with its
    // corresponding user interface string.
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    // The command string.
    private String commandString;

    /**
     * Initialize with the corresponding command string.
     * @param commandString The command string.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * @return The command word as a string.
     */
    public String toString()
    {
        return commandString;
    }
}
```

Pregunta: Según el texto de la bibliografía básica de la asignatura, indique cuál de las siguientes afirmaciones es correcta:

a. El encapsulamiento apropiado en las clases reduce su acoplamiento.
b. El acoplamiento describe el encapsulamiento de las clases.
c. El encapsulamiento apropiado en las clases reduce su cohesión.
d. La cohesión de una unidad de código refleja su acoplamiento.

Pregunta: Según el texto de la bibliografía básica de la asignatura, indique cuál de las siguientes afirmaciones es correcta:

a. Un encapsulamiento apropiado en las clases reduce el acoplamiento.
b. El término acoplamiento describe cuánto se ajusta una unidad de código a una tarea lógica o a una entidad.
c. El acoplamiento describe la conectividad de los propios objetos de una clase.
d. Un sistema débilmente acoplado se caracteriza por la imposibilidad de modificar una de sus clases sin tener que realizar cambios en ninguna otra.

Pregunta: Según el texto de la bibliografía básica de la asignatura, cuando un objeto permite realizar un conjunto de tareas muy relacionadas entre sí, podemos afirmar que:

a. El objeto presenta una alta cohesión.
b. El objeto está muy acoplado.
c. El objeto está poco encapsulado.
d. El objeto presenta una baja cohesión.

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.

Enumerados

- Enumerados
  - https://www.youtube.com/watch?v=DwriSApbm50&list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk&t=0s&index=49
  - https://www.youtube.com/watch?v=js_tdQOYng4
  - https://www.discoduroderoer.es/ejercicios-propuestos-y-resueltos-basicos-java/
  - https://jarroba.com/enum-enumerados-en-java-con-ejemplos/