Práctica 2: Buffy the Vampire Slayer refactored - Parte I

Fecha de entrega:
[Parte I, refactorización] 30 de Noviembre de 2020, 9:00.
[Parte II, extensiones de juego] 14 Diciembre de 2020, 9:00

Objetivo: Herencia, polimorfismo, clases abstractas e interfaces

1. Introducción

El objetivo de esta práctica consiste, fundamentalmente, en aplicar los mecanismos que ofrece la POO para mejorar el código desarrollado hasta ahora. En particular, en esta versión de la práctica incluiremos las siguientes mejoras:

- Primero, refactorizamos[1] el código de la práctica anterior. Para ello, modificaremos parte del controlador distribuyendo su funcionalidad entre un conjunto de clases.
- Vamos a hacer uso de la herencia para reorganizar los objetos de juego. Hemos visto que hay mucho código repetido en los distintos tipos de objetos (Vampiros y Slayers). Por ello, vamos a crear una estructura de clases que nos permita extender fácilmente la funcionalidad del juego.
- Una vez refactorizada la práctica, vamos a añadir nuevos objetos al juego.
- La herencia también nos va a permitir redefinir cómo almacenamos la información del estado del juego. En la práctica anterior, al no usar herencia, debíamos tener una lista para cada conjunto de objetos. Sin embargo, en esta versión de la práctica, podremos usar una sola estructura de datos para todos los objetos de juego.

Todos los cambios comentados anteriormente se llevarán a cabo de forma progresiva. El objetivo principal es extender la práctica de una manera robusta, preservando la funcionalidad en cada paso que hagamos y modificando el mínimo código para ampliarla.

[1] Refactorizar consiste en cambiar la estructura del código (se supone que para mejorarlo) sin cambiar su funcionalidad.

2. Refactorización de la solución de la práctica anterior

Hay una regla no escrita en programación que dice *Fat models and skinny controllers*. Lo que viene a decir es que el código de los controladores debe ser mínimo y, para ello, deberemos llevar la mayor parte de la funcionalidad a los modelos. Una manera de adelgazar el controlador es utilizando el patrón *Command* que, como veremos, permite encapsular acciones de manera uniforme y extender el sistema con nuevas acciones sin modificar el controlador.

El cuerpo del método run del controlador va a tener - más o menos - este aspecto. Tu código no tiene que ser exactamente igual, pero lo importante es que veas que se asemeja a esta propuesta.

```
while (!game.isFinished()){
        if (refreshDisplay) printGame();
        refreshDisplay = false;
      System.out.println(prompt);
      String s = scanner.nextLine();
      String[] parameters = s.toLowerCase().trim().split(" ");
      System.out.println("[DEBUG] Executing: " + s);
       Command command = CommandGenerator.parse(parameters);
       if (command != null) {
            refreshDisplay = command.execute(game);
        }
        else {
              System.out.println("[ERROR]: "+ unknownCommandMsg);
        }
  }
```

Básicamente, mientras el juego no termina, leemos un comando de la consola, lo parseamos, lo ejecutamos y, si la ejecución es satisfactoria y ha cambiado el estado del juego, lo repintamos. Este mismo controlador nos valdría para diferentes versiones del juego o incluso para diferentes juegos. En la próxima sección vamos a ver cómo funciona el patrón *Command*.

2.1. Patrón Command

El patrón *Command* es un patrón de diseño[2] muy conocido. En esta práctica no necesitas conocer de este patrón más de lo que se explica aquí. Para aplicarlo, cada comando del juego se representa por una clase diferente, que llamamos AddCommand, UpdateCommmand, ResetCommand, HelpCommand,... y que heredan de una clase abstracta Command. Las clases concretas invocan métodos de la clase Game para ejecutar los comandos respectivos.

En la práctica anterior, para saber qué comando se ejecutaba, el método run del controlador contenía un switch - o una serie de if's anidados - cuyas opciones correspondían a los diferentes comandos. Con la aplicación del patrón *command*, para saber qué comando ejecutar, el método run del controlador divide en palabras el texto proporcionado por el usuario (*input*) a través de la consola, para a continuación invocar un método de la *clase utilidad*[3] CommandGenerator, al que se le pasa el *input* como parámetro. Este método le pasa, a su vez, el *input* a un objeto *comando* de cada una de las clases concretas (que,

[2]Los patrones de diseño de software en general, y el patrón *command* en particular, se estudian en la asignatura Ingeniería del Software.

[3]Una clase utilidad es aquella en la que todos los métodos son estáticos.

como decimos, son subclases de Command) para averiguar cuál de ellos lo acepta como correcto. De esta forma, cada subclase de Command busca en el *input* el texto del comando que la subclase representa.

Aquel objeto *comando* que acepte el input como correcto devuelve al CommandGenerator otro objeto *comando* de la misma clase que él [4] . El parseador pasará, a su vez, el objeto *comando* recibido al controlador. Los objetos *comando* para los cuales el input no es correcto devuelven el valor null. Si ninguna de las subclases concretas de comando acepta el input como correcto, es decir, si todas ellas devuelven null, el controlador informa al usuario de que el texto introducido no corresponde a ningún comando conocido. De esta forma, si el texto proporcionado por el usuario corresponde a un comando del sistema, el controlador obtiene del parseador CommandGenerator un objeto de la subclase que representa a ese comando y que puede, a su vez, ejecutar el comando.

Implementación

El código de la clase abstracta Command es el siguiente:

```
package control.Commands;

import logic.Game;

public abstract class Command {

    protected final String name;
    protected final String shortcut;
    private final String details;
    private final String help;

    protected static final String incorrectNumberOfArgsMsg = "Incorrect number of
        arguments";
    protected static final String incorrectArgsMsg = "Incorrect arguments format";

    public Command(String name, String shortcut, String details, String help){
      this.name = name;
      this.shortcut = shortcut;
      this.details = details;
      this.help = help;
    }

    public abstract boolean execute(Game game);

    public abstract Command parse(String[] commandWords);

    protected boolean matchCommandName(String name) {
        return this.shortcut.equalsIgnoreCase(name) ||
            this.name.equalsIgnoreCase(name);
    }

    protected Command parseNoParamsCommand(String[] words) {

        if (matchCommandName(words[0])) {
            if (words.length != 1) {
                System.err.println(incorrectArgsMsg);
                return null;
            }
            return this;
```

[4] Si el comando no tiene parámetros, el objeto devuelto por este objeto comando puede ser él mismo

```
        }

        return null;
    }

    public String helpText(){
      return details + ": " + help + "\n";
    }
}
```

De los métodos abstractos anteriores, execute se implementa invocando algún método con el objeto game pasado como parámetro y ejecutando alguna acción más. El método parse se implementa con un método que parsea el texto de su primer argumento (que es el texto proporcionado por el usuario por consola, dividido en palabras) y devuelve:

- o bien un objeto de una subclase de Command, si el texto que ha dado lugar al primer argumento se corresponde con el texto asociado a esa subclase,
- o el valor null, en otro caso.

La clase CommandGenerator contiene la siguiente declaración e inicialización de atributo:

```
private static Command[] availableCommands = {
      new AddCommand(),
      new HelpCommand(),
      new ResetCommand(),
      new ExitCommand(),
      new UpdateCommand()
  };
```

Este atributo se usa en los dos siguientes métodos de CommandGenerator:

- public static Command parseCommand(String[] commandWords), que, a su vez, invoca el método parse de cada subclase de Command, tal y como se ha explicado anteriormente,
- public static String commandHelp(), que tiene una estructura similar al método anterior, pero invocando el método helpText() de cada subclase de Command. Este método es invocado por el método execute de la clase HelpCommand.

Una de las ventajas de usar el patrón *Command* es que es muy fácil y natural implementar el "Crtl-Z" o undo. Al encapsular los comandos como objetos los podemos apilar y desapilar, yendo para adelante y para atrás en la historia del juego. No lo vamos a implementar en esta práctica pero seguro que durante tu carrera lo utilizas más de una vez.

2.2. Herencia y polimorfismo

Quizás la parte más frustrante y mayor fuente de errores de la primera práctica es tener que replicar código en los objetos del juego y en las listas de objetos. Esto lo vamos a resolver usando la herramienta básica de la programación orientada a objetos: la *herencia*. Para ello, se creará una jerarquía de clases que heredan de GameObject.

- La clase abstracta GameObject tendrá los atributos y métodos básicos para controlar la posición en el tablero y una referencia a la clase Game.
- De GameObject heredará Slayer y Vampire. La primera clase representará las defensas del juego, mientras que la segunda representará los objetos que causan daño.

Es importante remarcar que en esta versión de la práctica, la mayor parte de la lógica estará distribuida en los propios elementos del juego, quedando la clase Game liberada de esta tarea. De esta forma se consigue un diseño escalable, permitiendo añadir nuevos elementos en el juego de forma sencilla.

Adicionalmente, podemos utilizar la herencia para refactorizar el código de las listas. En esta práctica, utilizaremos una clase – llamada GameObjectBoard – que será la encargada de gestionar la lista de elementos de tipo GameObject. La clase Game sólo tendrá un atributo de tipo GameObjectBoard. Con este atributo se gestionarán todos los elementos del juego.

El hecho de emplear la palabra *board* no quiere decir que ahora vayamos a utilizar como tablero una matriz para gestionar los objetos del juego, seguiremos usando una lista de GameObjects donde estarán todos los objetos de juego utilizando un ArrayList<GameObject> del package java.util.

Para el desarrollo de la práctica 1 refactorizada vamos a emplear ArrayList en GameObjectList de forma que se unifiquen las dos listas (Vampiros y Slayers) que se hacían de forma independientes con arrays en la practica 1.

```
import java.util . ArrayList;

public class GameObjectList {
   private ArrayList<GameObject> gameobjects;
        ....
}
```

2.3. Detalles de implementación

Hay varios aspectos que van a cambiar radicalmente la estructura del código:

- Sólo tenemos un contenedor para todos los objetos.
- Desde el Game y el board sólo manejamos abstracciones de los objetos, por lo que no podemos distinguir quién es quién.
- Toda la lógica del juego estará en los objetos de juego. Cada clase concreta sabe sus detalles acerca de cómo se mueve, cómo ataca, cómo recibe ataque y cuándo realiza *computer actions*.

Estas refactorizaciones son complejas y os vamos a dar ayuda; empecemos por la clase Game, la cual se encarga de demasiadas tareas en la práctica 1. En esta nueva versión, al igual que hemos hecho con el controlador, vamos a delegar su comportamiento a las demás clases.

Vamos a usar un interfaz para encapsular los métodos relacionados con los *ataques*. La clase GameObject implementa el interfaz IAttack. La idea es que todos los objetos del juego deben tener la posibilidad de atacar o ser atacados. Por ello, tenemos que implementar la posibilidad de recibir ataques.

```
package logic.GameObjects;

public interface IAttack {

    void attack();

    default boolean receiveSlayerAttack(int damage) {return false;};
    default boolean receiveVampireAttack(int damage) {return false;};
    default boolean receiveLightFlash() {return false;};
    default boolean receiveGarlicPush() {return false;};
    default boolean receiveDraculaAttack(){return false;};
}
```

Este tipo de estructuras puede resultar compleja la primera vez que la implementamos, pero tiene bastante lógica veamos por qué. Cuando creamos un objeto de juego (ya sea slayer o vampiro) y lo metemos en el ArrayList de GameObjects dejamos de saber qué tipo de objeto es, solo el propio objeto sabe a qué subclase de GameObject pertenece. La pregunta entonces es cómo implementamos la interacción entre dos objetos que no sabemos qué son exactamente. Tenemos varias opciones:

- Delegar al game, que propague el tipo de ataque desde el objeto atacante al objeto dañado, como hacíamos en la primera práctica. Esto no está del todo mal, pero tiene el problema de que conforme añadamos nuevos tipos de objetos de juego tendremos más y más métodos de delegación.

  ```
  // VAMPIRE
   public void attack() {
            ...
            game.attackSlayerInPosition(x - 1, y, HARM);
            ...
  }
  ```

  Lo que queremos es que la funcionalidad esté en los propios objetos de juego, para que sea fácil extenderla y modificarla sin afectar a otros objetos. Veamos otras opciones:

- Pedirle objetos al Game usar instaceOf o getClass para saber qué tipo de objeto es cada uno. Haciendo algo así:

  ```
  // VAMPIRE
   public void attack() {
            ...
            GameObject other = game.getObjectInPosition(x - 1, y);
            if (other != null  && other.getClass() == "Slayer")
                  ((Slayer)  other).decreaseLife(HARM);
            ...
  }
  ```

  Esto va contra todos los principios de la orientación a objetos ya que estamos resolviendo la clase de los objetos en lugar de hacer uso del polimorfismo. Esto está terminantemente prohibido.

- Pedirle objetos al Game e implementar un método isSlayer para saber si es atacable por un vampiro:

  ```
  // VAMPIRE
   public void attack() {
            ...
  ```

```
            GameObject other = game.getObjectInPosition(x - 1, y);
            if (other != null  && other.isSlayer())
                  ((Slayer)  other). decreaseLife (HARM);
             ...
}
```

Esta solución está prohibida como la anterior ya que es lo mismo pero implementada de otra manera.

- Pedirle objetos al Game y usar el interfaz IAttack que hemos presentado más arriba:

```
// VAMPIRE
 public void attack() {
            ...
            GameObject other = game.getObjectInPosition(x - 1, y);
            if (other != null)
                  other. receiveVampireAttack(HARM);
            ...
}
```

Esta solución no está mal porque cada objeto sabe cómo ataca y que otros objetos le hacen daño, el problema es que rompemos la encapsulación al devolver un objeto "valor"de game. Para solucionar el problema debemos hacer que dos objetos sólo se comuniquen a través del interface que es una abstracción o contrato entre ellos.

Las solución que proponemos es la siguiente. Los ataques de los gameObjects como los vampiros y slayers deben de pedirle a su game referencias declaradas como IAttack para desacoplar Game de GameObject. Esto hace referencia al Principio de inversión de la dependencia (Dependency inversion principle) que indica que se debe depender de abstracciones y no de implementaciones. Por ejemplo, el método attack de Vampire podría ser:

```
 public void attack() {
            if (isAlive ()) {
                  IAttack other = game.getAttackableInPosition(x - 1, y);
                  if (other != null)
                        other. receiveVampireAttack(HARM);
             }
}
```

Utilizando interfaces también podemos desacoplar el Game del GamePrinter. Se debe añadir en el *package* view la siguiente interface Iprintable que debe implementar la clase Game:

```
public interface IPrintable {
    String getPositionToString(int x, int y);
    String getInfo();
}
```

De modo que el Gameprinter no necesita un array String [][] board y el constructor de Gameprinter en lugar de recibir un Game podría ser:

```
 public GamePrinter (IPrintable printable, int cols, int rows) {
              this. printable  = printable;
              this. numRows = rows;
              this. numCols = cols;
}
```

De esta manera GamePrinter depende de la interface IPrintable y no de Game.

En la práctica 1 los slayer atacan antes que los vampiros. En la práctica 1 refactorizada todos los GameObject se almacenan en el ArrayList y atacan en el orden en que se almacenan, que es el orden en que se crean, por lo que algún test como el *easy_s5_2.txt* puede tener una salida distinta al de la práctica 1.

IMPORTANTE: La falta de encapsulación, el uso de métodos que devuelvan listas, y el uso de "instanceOf()" o "getClass()" tiene como consecuencia un suspenso directo en la práctica. Es incluso peor implementar un *instanceof* casero, por ejemplo: cada subclase de la clase *GameObject* contiene un conjunto de métodos *esX*, uno por cada subclase X de *GameObject*; el método *esX* de la clase X devuelve *true* y los demás métodos esX de la clase X devuelven *false*.