Práctica 2: Space Invaders refactored, parte I

Fecha de entrega: 5 de Diciembre de 2019, 9:00

Objetivo: Herencia, polimorfismo, clases abstractas, interfaces, excepciones y gestión de ficheros

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, como se explica en la sección 2, 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 aliens o entre el misil y las bombas. 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.
- Vamos a incluir la posibilidad de modificar cómo se representa el tablero. Para ello, crearemos otro *printer* con el que podamos ver el juego de forma *serializada*.

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

- Utilizaremos la serialización poder grabar y cargar partidas en disco. Para ello, ampliaremos el juego para gestionar ficheros de entrada y salida.

- También vamos a incluir el manejo y tratamiento de excepciones. En ocasiones, existen estados en la ejecución del programa que deben ser tratados convenientemente. Además, cada estado debe proporcionar al usuario información relevante como, por ejemplo, errores producidos al procesar un determinado comando. En este caso, el objetivo es dotar al programa de mayor robustez, así como mejorar la interoperabilidad con el usuario.

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.

Importante: Hemos divido el enunciado en varias partes que iremos publicando progresivamente, en esta primera parte nos centraremos en el primer item de la lista. Solo habrá una entrega para toda la práctica completa.

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()){
  System.out.println(PROMPT);
  String[]  words = in.nextLine().toLowerCase().trim().split ("\\s+");

  try {
        Command command = CommandGenerator.parse(words);
        if (command != null) {
          if (command.execute(game))
              System.out.println(game);
        }
        else {
          System.out.format(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*. En el bucle principal de juego también gestionaremos dos tipos de excepciones

en los bloques try-catch, dependiendo de si ha habido un problema con la ejecución del comando o con al parseo del mismo. Hablaremos de ello más adelante, ahora nos centraremos en los comandos.

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 MoveCommand, ShootCommand, 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, 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. 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:

```
public abstract class Command {

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


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

  public Command(String name, String shortcut, String details, String help){
```

[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.

```
    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);
  }

  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 ListCommand(),
  new HelpCommand(),
  new ResetCommand(),
  new ExitCommand(),
  new ListCommand(),
  new UpdateCommand(),
  new MoveCommand(),
  new ShockwaveCommand()
};
```

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.