Tecnología de la Programación I

Presentación de la Práctica 2 (parte I)

(Basado en la práctica de Alberto Núñez y Miguel V. Espada)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción

- ✓ La práctica 2 está especialmente dedicada a **herencia**, **excepciones** y **entrada/salida**.
- ✓ La fecha de entrega es: **5 de diciembre** a las 9:00.
- ✓ La entrega debe realizarse utilizando el campus virtual, no mas tarde de la fecha indicada. Debes subir un fichero comprimido (**.zip**) que contenga lo siguiente. No incluyas los ficheros que resultan de la compilación (los del directorio **bin**).
  - Directorio *src* con el código fuente de todas las clases de la práctica.
  - Fichero *alumnos.txt* donde se indica el nombre de los componentes del grupo.
- ✓ No es necesario entregar la documentación *javadoc*.

- ✓ Con esta práctica mejoramos y extendemos la anterior aplicando algunos de los mecanismos que ofrece POO.
- ✓ En particular, en esta práctica incluiremos las siguientes **mejoras**:
  - Primero, **refactorizamos** el código de la práctica anterior modificando parte del controlador, distribuyendo su funcionalidad entre un conjunto de clases.
  - Vamos a hacer uso de la **herencia** para reorganizar los objetos de juego y evitar código repetido; 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 **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.
- Utilizaremos la **serialización** para 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 de **excepciones** tratando, por ejemplo, errores producidos al procesar un determinado comando. El objetivo es dotar al programa de mayor robustez, así como mejorar la interoperabilidad con el usuario.

2. Refactorización

- ✓ *Fat models and skinny controllers*: el código de los controladores debe ser mínimo y, para ello, debemos llevar la mayor parte de la funcionalidad a los modelos.
- ✓ Una manera de adelgazar el controlador es utilizando el **patrón *Command***.
- ✓ Este permite encapsular acciones de manera uniforme y extender el sistema con nuevas acciones sin modificar el controlador.

- El cuerpo del método *run* de *Controller* va a tener - más o menos - este aspecto:

```
while (!game.isFinished()){
    System.out.print(PROMPT);
    String[] words = in.nextLine().toLowerCase().trim().split ("\\s+");
    try {
        Command command = CommandGenerator.parseCommand(words);
        if (command != null) {
            if (command.execute(game))
                System.out.println(game);
        }
        else {
            System.out.format(unknownCommandMsg);
        }
    }
}
```

- ✓ Mientras el juego no termina, leemos un comando de la consola, lo parseamos (lo convertimos en un objeto comando), 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 el bucle principal del juego también gestionamos 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 las excepciones más adelante, ahora nos centraremos en los comandos.

3. Patrón Command

- ✓ El patrón *command* es un patrón de diseño muy conocido.
- ✓ 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.
- ✓ Los métodos *update* y *computerAction* de *Game* también se llaman desde estas clases (ver el nuevo método *run* de *Controller*).

- ✓ En la práctica anterior, para saber qué comando se ejecutaba, el método *run* de *Controller* 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* de *Controller* divide en palabras el texto proporcionado por el usuario (*input*) a través de la consola, para a continuación invocar el método *parseCommand* de la clase utilidad ***CommandGenerator***, al que se le pasa el *input* como parámetro. Esta clase se ve más adelante.
- ✓ Una clase utilidad tiene todos sus métodos estáticos.

- ✓ Este método *parseCommand* (parseador) de *CommandGenerator* 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 a *Controller*.

- ✓ 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 de *CommandGenerator* un objeto de la subclase que representa a ese comando y que puede, a su vez, ejecutar el comando.

- 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){
      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 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 argumento se corresponde con el texto asociado a esa subclase,
  - o el valor null, en otro caso.

- Las subclases que heredan de *Command* dan valor a sus atributos estáticos. Por ejemplo, la clase *HelpCommand* tiene los siguientes atributos:

```
protected final static String name = "help";
private final static String details = "[h]elp";
protected final static String shortcut = "h";
private final static String help = "shows this help.";
```

- Estas subclases implementan los métodos *parse* y *execute*. Por ejemplo, la subclase *UpdateCommand* implementa el siguiente método *execute*:

```
public boolean execute(Game game) {
        game.computerAction();
        game.update();
        return true;
}
```

- ✓ Para implementar el método *parse*, estas subclases de *Command* pueden utilizar el método *matchCommandName* de *Command*. Este método se utiliza para ver si *commandWords*[0] coincide con el atributo *name* o el atributo *shortcut*.

- 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*.

- El método *parseCommand* recorre con un bucle for-each el array de *availableCommands* llamando al método parse de cada clase Command. Si ese método parse devuelve un objeto Command distinto de nulo, entonces se hace break en el bucle for.
- El método *commandHelp* utiliza *StringBuilder*, haciendo *append* de cada *String* devuelto por el método *helpText* de cada objeto *Command* del array *availableCommands*. Por ello es muy útil el bucle for-each para esta implementación.
- Una de las ventajas de usar el patrón *Command* es que es muy fácil y natural implementar el "Crtl-Z" o **undo**.