



Tecnología de la Programación

Presentación de la Práctica 2

(Basado en la práctica de Miguel V. Espada)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción.
2. Patrón Command.
3. Patrón Factory.
4. Herencia y Polimorfismo.
5. Extensión del Juego: Básico.

1. Introducción

- ✓ Esta práctica está especialmente dedicada a **herencia**.
- ✓ Otros objetivos son: polimorfismo, clases abstractas e interfaces.
- ✓ Fecha de entrega: **19 de Noviembre** a las 9:00.
- ✓ No es necesario entregar la documentación *javadoc*.
- ✓ Con esta práctica mejoramos y extendemos la anterior:
 - Primero refactorizamos el código aplicando el patrón *Command* y el patrón *Factory*. Modificamos el código de tal forma que haga lo mismo pero mejorando su diseño. Usamos soluciones a problemas comunes (patrones).
 - Añadimos nuevos objetos al juego: nuevas plantas y nuevos zombies.
 - Damos la posibilidad de modificar como se pinta el tablero creando el modo *Debug* y el modo *Release* (de la práctica anterior).

2. Patrón Command

- ✓ El patrón *Command* permite reducir el controlador y extender el sistema sin necesidad de modificarlo.
- ✓ Usando este patrón, el bucle del método *run()* queda así:

```
while (!game.isFinished() && !exit) {  
    printGame();  
    noPrint = false;  
    System.out.print(prompt);  
    String[] words = scanner.nextLine().toLowerCase().trim().split("\\s+");  
    Command command = CommandParser.parseCommand(words, this);  
    if (command != null) {  
        command.execute(game, this);  
    } else {  
        System.err.println(unknownCommandMsg);  
        setNoPrintGameState();  
    }  
}
```

- ✓ Con este nuevo bucle, mientras el juego no termina (por orden del usuario o causas del juego), pintamos el juego, leemos el comando de la consola, lo parseamos (obtenemos un objeto *Command*) y lo ejecutamos (llamamos al método *execute()* de esa clase *Command*).
- ✓ El atributo *noPrint* nos sirve para controlar si hay que repintar el tablero después de la ejecución de un comando y cuando no. El método *setNoPrintGameState* pone este atributo a true.
- ✓ El nuevo controlador nos sirve para diferentes versiones del juego o incluso nuevos juegos.

- ✓ El patrón *Command* es un patrón de diseño muy conocido.
- ✓ Para aplicarlo en la práctica, cada comando del juego se representa con una clase diferente: *AddCommand*, *UpdateCommand*, *ResetCommand*, *HelpCommand*, etc.
- ✓ Estas clases heredan (son subclases) de una clase abstracta *Command* e invocan métodos de la clase *Game* para ejecutar los comandos correspondientes.
- ✓ En el bucle de *run()* ya no usamos un switch (o if's anidados). Ahora usamos la clase utilidad (clase cuyos métodos son estáticos) *CommandParser* para encontrar la subclase de *Command* cuyo método *execute()* debe ejecutarse.
- ✓ El código de la clase abstracta *Command* es el siguiente. Como puede verse, todas sus subclases tienen un método *parse()* y un método *execute()*.


```
public abstract class Command {  
    private String helpText;  
    private String commandText;  
    protected final String commandName;  
  
    public Command(String commandText, String commandInfo, String  
helpInfo){  
        this.commandText = commandInfo;  
        helpText = helpInfo;  
        String[] commandInfoWords = commandText.split("\\s+");  
        commandName = commandInfoWords[0];  
    }  
  
    public abstract void execute(Game game, Controller controller);  
    public abstract Command parse(String[] commandWords, Controller  
controller);  
    public String helpText(){return " " + commandText + ": " + helpText  
;}  
}
```

- ✓ El método *parse()* parsea el array de palabras suministradas por el usuario (primer argumento) y devuelve:
 - Un objeto de la subclase de *Command* (*this*) cuando el array de palabras se corresponde con el texto asociado a esa subclase.
 - null en caso contrario.
- ✓ Las subclases de *Command* que no tienen parámetros (por ejemplo, *ExitCommand*) no heredan directamente de *Command* sino de una clase abstracta intermedia llamada *NoParamsCommand* que hereda de *Command*.
- ✓ La subclase *NoParamsCommand* implementa el método *parse()* para sus subclases comparando la posición 0 del array de palabras suministradas por el usuario con el atributo *commandName* de *Command* (como texto asociado).
- ✓ Las subclases de *NoParamsCommand* sólo necesitan implementar el método *execute()*.

- ✓ Las subclases de *Command* que corresponden a comandos con parámetros (por ejemplo, *AddCommand*) necesitan atributos para guardar sus parámetros. Estos atributos (por ejemplo, en *AddCommand*, *plantName*, *x* e *y*) toman valores en el método *parse()* al parsear el array de palabras del usuario.
- ✓ La clase *CommandParser* contiene la siguiente declaración e inicialización de atributo:

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

- ✓ Esta clase contiene dos métodos estáticos:

```
public static Command parseCommand(String[] commandWords, Controller controller)
```

```
public static String commandHelp()
```

- ✓ El primero utiliza un bucle for-each para recorrer el atributo *availableCommands* llamando al método *parse()* sobre cada uno de sus objetos y, si el resultado de este parseado es un objeto *Command* distinto de null, entonces devuelve ese objeto.
- ✓ El controlador se pasa como segundo argumento a este método (y, a su vez, al método *parse()* y al método *execute()*) para poder invocar el método *setNoPrintGameState()* del controlador cuando sea necesario.

- ✓ El método *commandHelp()* utiliza también un bucle for-each llamando al método *helpText()* de cada uno de los objetos del atributo *availableCommands*. Mediante un objeto *StringBuilder* creado antes del bucle for, el método compone el String de todas las ayudas.
- ✓ Este método es invocado por el método *execute()* de la clase *HelpCommand*.

3. Patrón Factory

- ✓ El patrón *Factory* también es muy utilizado.
- ✓ Básicamente, este patrón es “responsable de crear objetos evitando exponer la lógica de instanciación al cliente”.
- ✓ En la primera práctica, la lógica de creación de plantas está fuertemente acoplada con el controlador. Para incluir una nueva planta, es necesario incluir un nuevo bloque al switch (o if's anidados) del método *run()*

```
case "sunflower":  
case "s":  
    result = game.addSunflower(x, y);  
    break;  
case "peashooter":  
case "p":  
    result = game.addPeashooter(x, y);  
    break;
```

- ✓ En la nueva versión, podemos usar el siguiente código en el método *execute()* de la clase *AddCommand*:

```
Plant plant = PlantFactory.getPlant(plantName);  
game.addPlantToGame(plant, x, y);
```
- ✓ Con esta implementación, la lógica de creación de objetos de plantas (antes en el método *addSunflower()* de *Game*) se lleva a la clase *PlantFactory* que se dedica exclusivamente a ello.
- ✓ Ahora, añadir o eliminar nuevas plantas de la lista se reduce a crear la clase de planta correspondiente y modificar *PlantFactory*. Así, los cambios en la lista ya no afectan al controlador o al juego. Y la lógica de creación está desacoplada de la lógica del juego.

- ✓ El patrón *Factory* y el patrón *Command* se combinan muy bien. Hemos visto como implementar en el método *execute()* de *AddCommand* la creación de la planta con la factoría.
- ✓ También podemos usar la factoría en el método *execute()* de *ListCommand* para saber las plantas disponibles. Basta con preguntar a *PlantFactory* que *availablePlants* tiene.
- ✓ El siguiente código muestra el esqueleto de la factoría:

```
public class PlantFactory {  
    private static Plant[] availablePlants = {  
        new Sunflower(), ...  
    };  
    public static Plant getPlant(String plantName){  
        ....  
    }  
    public static String listOfAvilablePlants() {  
        StringBuilder sb = new StringBuilder();  
        ...  
    }  
}
```


- ✓ El método *listOfAvailablePlants()* lo usa el método *execute()* de la clase *ListCommand* para mostrar información de las plantas disponibles.
- ✓ En la primera práctica sólo teníamos dos tipos de plantas y un tipo de zombi. Pero nuestro objetivo es poder extenderla de manera sencilla incorporando nuevos objetos de juego.
- ✓ Aunque **no es necesario**, puedes implementarse una *ZombieFactory* que permita desacoplar la lógica de creación de objetos zombi de la lógica del juego.
- ✓ En la primera práctica, el juego es el que se encarga de crear los objetos zombi. Con *ZombieFactory* y una clase comando *AddZombie* puedes depurar colocando los zombis a tu antojo o aleatoriamente.

4. Herencia y Polimorfismo

- ✓ En la primera práctica hemos replicado código en los objetos de juego y en las listas. Esto lo vamos a resolver usando **herencia**.
- ✓ En esta práctica creamos una jerarquía de clases que heredan de la clase abstracta *GameObject*. En esta clase es interesante definir atributos x e y .
- ✓ Todas las plantas y todos los zombis son objetos de juego.
- ✓ Se puede extender la jerarquía definiendo *Plant* y *Zombi* también como clases abstractas de las que heredan los distintos tipos de plantas y zombis.

- ✓ En lugar de tener que usar una lista para cada tipo de objeto de juego, podemos tener **una lista** que almacene todos ellos. Esto simplifica mucho la gestión del tablero.
- ✓ Algunas características de esta nueva lista son:
 - El tamaño no es fijo como en la primera práctica. Se trata de una **lista dinámica** que puede crecer cuando se queda sin espacio y eliminar objetos de juego muertos.
 - El orden de los **update's** cambia. En esta versión los objetos se actualizan en el orden en que fueron incorporados al tablero. Así que la lista siempre permanece ordenada por antigüedad.
- ✓ Otra mejora de la nueva práctica se refiere a la incorporación de plantas en la última columna que bloquea la salida de zombis. Ahora el jugador sólo va a poder poner plantas hasta la penúltima fila.

5. Extensión del Juego: Básico

- ✓ En esta práctica incorporamos nuevos tipos de plantas y zombies haciendo uso de la factoría y herencia.
- ✓ Nuevos tipos de plantas:
 - **Petacereza.** Es una planta que explota y quita 10 puntos de daño a todos los zombies que están a su alrededor. Cuando explota muere. Y la explosión ocurre dos ciclos después de ser plantada. Los zombies también se pueden comer a la planta que tiene resistencia 2. Su coste es 50 suncoins.
 - **Nuez.** Es una planta que sirve como barrera. Tiene coste 50 suncoins y resistencia 10. Esta planta no produce ningún daño.
- ✓ Nuevos tipos de zombies:
 - **Caracubo.** Camina un paso cada 4 ciclos y tiene resistencia 8.
 - **Deportista.** Camina un paso cada ciclo y tiene resistencia 2.

- ✓ Los dos tipos de zombis ejercen el mismo daño que el **zombi común** de la primera práctica. Además, aparecen con la misma probabilidad.
- ✓ Los símbolos (posible atributo de *GameObject*) que se utilizan son C para Petacereza, N para Nuez, W para Caracubo y X para Deportista.
- ✓ Para implementar todos los tipos de plantas y zombis basta con desarrollar las clases correspondientes y registrar los nuevos objetos en la factoría correspondiente.
- ✓ Así queda la nueva lista de plantas:

```
Command > list
```

```
[S]unflower: Cost: 20 suncoins Harm: 0
```

```
[P]eashooter: Cost: 50 suncoins Harm: 1
```

```
Peta[c]ereza: Cost: 50 suncoins Harm: 10
```

```
[N]uez: Cost: 50 suncoins Harm: 0
```

- ✓ Imprimir la lista de zombis **no es obligatorio**. Pero se recomienda crear el comando y la factoría para facilitar la depuración.
- ✓ Así queda la nueva lista de zombis:

```
Command > zombieList  
[Z]ombie comun: speed: 2 Harm: 1 Life: 5  
[Z]ombie deportista: speed: 1 Harm: 1 Life: 2  
[Z]ombie caracubo: speed: 4 Harm: 1 Life: 8
```
- ✓ En esta práctica también se extiende el juego permitiendo dos tipos de pintado: *Release* y *Debug*.
- ✓ El modo *Release* es el que vimos en la práctica anterior.
- ✓ En el modo *Debug* mostramos más información sobre el estado del juego (nivel y semilla), así como la lista de objetos de juego en orden en lugar del tablero.

Number of cycles: 13

Sun coins: 30

Remaining zombies: 7

Level: INSANE

Seed: 0

|S[1:2,x:0,y:0,t:2]|W[1:8,x:0,y:4,t:3]|P[1:3,x:1,y:0,t:0]|

✓ En este modo componemos una lista que, para cada objeto de juego, incluye la siguiente información:

- Su símbolo
- La vida que le queda
- Su posición x, y
- El número de ciclos que le queda hasta que pueda ejecutar su siguiente acción (caminar, generar soles, etc.).

- ✓ Para implementar esta funcionalidad se recomienda hacer los siguientes cambios:
- Crear una interfaz *GamePrinter* con un método *printGame()*
 - Crear una clase abstracta *BoardPrinter* que contiene un método *boardToString()* que imprime el tablero (según la funcionalidad de la primera práctica) y, un método abstracto *encodeGame()*. Esta clase necesita al menos los siguientes atributos:

```
String[][] board;  
final String space = " ";
```

- Implementar clases concretas para los modos: *DebugPrinter* y *ReleasePrinter* que hereden de *BoardPrinter* e implementen la interfaz *GamePrinter*.
- *DebugPrinter* pinta un tablero de una única dimension. Así que su método *encodeGame()* solo rellena *board[0][j]*

- ✓ En la práctica anterior, el objeto *GamePrinter* era creado dentro del método *toString()* de la clase *Game*. En esta práctica desacoplamos esta funcionalidad y la subimos al controlador.
- ✓ Ahora el controlador tiene una instancia de una clase que implemente la interfaz *GamePrinter* y un método *printGame()* como el que se muestra a continuación:

```
private GamePrinter gamePrinter;  
...  
public void printGame() {  
    if(!noPrint) {  
        System.out.println(gamePrinter.printGame(game));  
    }  
}
```

- ✓ donde el booleano *noPrint* se utiliza para controlar cuando se repinta el juego y cuando no.

- ✓ El cambio de modo de juego se hace con el comando *PrintMode* (clase *PrintModeCommand*) de forma que éste recibe como parámetro el modo de render (*Release* o *Debug*). Con este comando, la lista de *help* queda así:

```
Command > help
```

```
The available commands are:
```

```
[A]dd: add flower.
```

```
[H]elp: print this help message.
```

```
[R]eset: resets game
```

```
[E]xit: terminate the program.
```

```
[L]ist: print the list of available plants.
```

```
zombieList: print the list of zombies.
```

```
[P]rintMode: change print mode [Release|Debug].
```

```
none: skips cycle
```

- ✓ Cuando se cambia el modo de juego se repinta el juego pero no avanza el número de ciclos. Así, podemos cambiar de uno a otro para evaluar y depurar una situación.