Práctica 2: Plantas vs Zombis refactored

Fecha de entrega: 19 de Noviembre de 2018, 09:00

Objetivo: Herencia, polimorfismo, clases abstractas e interfaces.

1. Introducción

En esta práctica vamos mejorar y extender la práctica anterior de la siguientes formas:

- Primero, como se explica en la sección 2, refactorizamos[1] el código de la práctica anterior, eliminando parte del código del método run del controlador y distribuyendo su funcionalidad entre un conjunto de clases. Esto supone aplicar el que se conoce como patrón *command* en combinación el patrón *factory*.
- Nuevos objetos de juego. Una vez refactorizada la práctica vamos a añadir nuevos *personajes* al juego: nuevas plantas y nuevos zombis.
- Por último, vamos a dar la posibilidad de modificar cómo se pinta el tablero, creando dos modos de renderizado, el modo Debug y el modo Release (que es el que vimos en la práctica anterior).

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.

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

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() && !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();
   }
}
```

Básicamente, mientras el juego no termina (por orden del usuario o causas del juego), pintamos el juego, leemos un comando de la consola, lo parseamos y lo ejecutamos. El noPrint y setNoPrintGameState nos ayudan a controlar cuándo repintar el tablero después de la ejecución de un comando y cuándo no. 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] CommandParser, 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 CommandParser otro objeto *comando* de la misma clase que él. El parseador pasará, a su vez, el objeto

[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.

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 CommandParser 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 {

  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];
  }

  /*
  Some commands may generate an error in the execute or parse methods.
  In the absence of exceptions, they must the tell the controller not to print the board
  */

  public abstract void execute(Game game, Controller controller);
  public abstract Command parse(String[] commandWords, Controller
      controller);

  public String helpText(){return " " + commandText + ": " + helpText
      ;}
}
```

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.

Aquellas subclases de Command que corresponden a comandos sin parámetros, no heredan directamente de Command, sino de una clase intermedia NoParamsCommand. Esta clase implementa el método parse haciendo uso del atributo commandName de la clase Command. Por ello, las clases que derivan de NoParamsCommand sólo necesitan implementar el método execute. Las subclases de Command que corresponden a comandos con

parámetros derivan directamente de Command y necesitan atributos para guardar el valor de sus parámetros.

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(),
};
```

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

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

La razón de que se pase el controlador, como argumento, al método parseCommand y este, a su vez, al método parse, así como al método execute, es poder invocar el método setNoPrintGameState() del controlador, cuando sea necesario.

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. Patrón Factory

El patrón *Factory* es otro de los más utilizados. Básicamente, este patrón es "responsable de crear objetos evitando exponer la lógica de instanciación al cliente". En la primera versión de la práctica, la lógica de creación de plantas está fuertemente acoplada con el controlador de la aplicación. La forma de incorporar una nueva planta es la de incluir un nuevo bloque al *switch* o *if's* que tenemos en el método run. Seguramente tu código se parece a este:

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

  case "peashooter":
  case "p":
    result = game.addPeashooter(x, y);
    break;
    default:
    System.err.println(invalidObjectMsg);
    noPrint = true;
```

En nuestra nueva versión queremos que se parezca a esto:

```
Plant plant = PlantFactory.getPlant(plantName);
game.addPlantToGame(plant, x, y);
```

Haciendo uso del patrón *Factory*, podemos extraer la lógica de creación a una clase dedicada exclusivamente a ello. De esta manera añadir o eliminar una planta de la lista es tan sencillo como crear la clase correspondiente y modificar la *Factory*. Así, los cambios en la lista ya no afectarán al controlador o al juego. Con esta propuesta, la lógica de creación esta desacoplada de la lógica del juego y puede evolucionar de forma independiente.

Implementación

En patrón *Factory* se combina muy bien con el patrón *Command*, ya que cuando ejecutamos un comando add plant x y podemos delegar la creación de la planta a la factoría. De la misma manera, cuando ejecutamos un comando list para saber las plantas disponibles, podemos preguntar a la factoría qué AvailablePlants tiene. El siguiente código muestra el esqueleto de la factoría:

```
public class PlantFactory {

  private static Plant[] availablePlants = {
    ...
  };

  public static Plant getPlant(String plantName){
     ....
  }

  public static String listOfAvilablePlants() {
     ...
  }
```

En la primera versión de la 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 con diferentes características. Aunque no es necesario, también puedes crear un ZombieFactory; ahora mismo es el juego es el que se encarga de la creación de zombis, pero si quieres desacoplar la creación de la lógica del juego puedes crear una factoría y un nuevo comando addZombie. Esta opción te puede ser muy útil para depurar, ya que podrás colocar los zombis a tu antojo o aleatoriamente.

El método listOfAvilablePlants lo usará el comando listCommand para mostrar la información de las plantas disponibles.

2.3. 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 de juego y en las listas de objetos. Esto lo vamos a resolver usando la herramienta básica de la programación orientada a objetos: la *herencia*. Por ello, se creará una jerarquía de clases que heredan de GameObject. Todos los zombis y las plantas son objetos de juego.

En lugar de tener que usar una lista para cada tipo de objeto de juego, vamos a poder usar dos listas, una que almacene todas las plantas y otra que almacene todos los zombis, lo cual va a simplificar la gestión del tablero. Algunas notas sobre estas listas:

- En la primera práctica, el enunciado no decía nada sobre el tamaño del array utilizado para implementar las listas. En esta práctica, decimos explícitamente que o bien hay que eliminar del array a los objetos de juego muertos, o bien hay que rediminsionar el array cuando se queda sin espacio (o incluso las dos cosas).
- En esta versión del juego, el orden del update será primero todas las plantas, luego todos los zombis, y dentro de cada una de estas dos categorías de objeto de juego, los objetos se actualizarán en el orden en el que fueron incorporados al tablero, así que las dos listas de objetos siempre deben estar ordenadas por antigüedad.

2.4. Otras mejoras

Vamos a aprovechar estas refactorizaciones tan grandes para corregir algún error que hemos detectado en la primera práctica, por ejemplo:

- El usuario puede poner una planta en la última columna del tablero bloqueando la aparición de zombis. Lo que vamos a hacer que el usuario solo pueda poner plantas hasta la penúltima columna.

Si has detectado algún *glitch* o inconsistencia en la primera práctica coméntalo con el profesor para que lo podamos incluir en las posibles mejoras.

3. Extensión del juego: básico

3.1. Incorporación de nuevos objetos de juego

Ahora que tenemos nuestra factoría y hemos utilizado herencia para generalizar los objetos de juego va a ser muy sencillo extender el juego con nuevas plantas y zombis. De esta forma, vamos a incluir las siguientes plantas:

- **Petacereza**: es una planta que explota y quita 10 puntos de daño a todos los zombis que están a su alrededor. La planta una vez que explota muere. La explosión ocurre dos ciclos después de ser plantada. Los zombis 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. Está planta no produce ningún daño

En cuanto a los zombies vamos a añadir los siguientes:

- **Caracubo**: es más lento pero más resistente. Camina un paso cada 4 ciclos y tiene resistencia 8.
- **Deportista**: es más rápido pero menos resistente. Camina un paso cada ciclo y tiene resistencia 2.

Los dos nuevos zombis ejercen el mismo daño que el zombi común de la primera práctica. Los diferentes tipos de zombis aparecen con la misma probabilidad.

Para hacer estos cambios puedes extender tu jerarquía de objetos teniendo GameObject, del que heredan Plant y Zombie (que van a ser clases abstractas), de los cuales heredan las clases concretas de plantas y zombies.

Los símbolos que vamos a utilizar son C para Petacereza, N para Nuez, W para Caracubo y X para Deportista. Si has planteado bien la lista de objetos y el juego no deberías tener que hacer ningún cambio importante, lo único que tendrás que hacer es crear las clases y registrar los nuevos objetos en sus respectivas factorías.

Así quedará nuestra 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
```

Aunque imprimir la lista de zombis no es obligatorio, recomendamos crear el comando y la factoría para facilitar la depuración. Así quedará la lista de zombies:

```
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
```

Así es cómo veremos veríamos la pantalla con diferentes tipos de zombi.

```
Command >
Number of cycles: 12
Sun coins: 60
Remaining zombies: 6

   -------------------------------------------------------------
  |       |       |       |       |       | N [4] | X [2] |       |
   -------------------------------------------------------------
  | S [2] |       |       |       |       |       |       |       |
   -------------------------------------------------------------
  |       |       |       |       |       | W [8] |       | X [2] |
   -------------------------------------------------------------
  |       |       |       |       |       |       | X [2] |       |
   -------------------------------------------------------------
```

3.2. Cambiar el modo de pintado

Vamos a permitir dos modos de pintado: Release y Debug. El modo Release es el que vimos en la práctica anterior, mientras que en el modo Debug mostramos más información sobre el estado del juego (nivel y semilla), y además pinta 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[l:2,x:0,y:0,t:2]|W[l:8,x:0,y:4,t:3]|P[l:3,x:1,y:0,t:0]|
 ------------------------------------------------------
```

Como vemos, se muestra la semilla y el nivel en el que estamos (además de la información que teníamos en la versión anterior). A su vez componemos una lista en la que para cada objeto de juego mostramos la siguiente información:

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

Para implementar esta funcionalidad se recomienda hacer los siguientes cambios.

- Vamos a crear un interfaz GamePrinter, con un método printGame.
- Vamos a crear una clase abstracta BoardPrinter la cual contiene la funcionalidad de pintar el tablero, así como el método abstracto encodeGame.
- Vamos a tener las clases concretas DebugPrinter y ReleasePrinter, las cuales heredan de la clase anterior e implementan el método encodeGame.
- DebugPrinter pinta un tablero de una sola dimensión.

Posiblemente en la práctica anterior el GamePrinter era instanciado y utilizado desde el Game; pero es más interesante desacoplar esta funcionalidad y subirla al controlador. Así pues, el controlador tendrá una instancia de una clase que implemente el interfaz GamePrinter:

```
private GamePrinter gamePrinter;

...

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

Recordamos que estamos usando un booleano para controlar cuándo se repinta el juego y cuándo no.

El cambio de modo de juego se hará con el comando PrintMode, de forma que éste recibe como parámetro el modo de render. Así quedará nuestra lista de comandos:

```
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
```

Es importante remarcar que, cuando cambiamos el modo de juego, se repinta el juego pero no hay un avance de ciclo. Así, podemos cambiar de uno otro para evaluar y depurar una situación concreta sin modificar el juego.

4. Extensión del juego avanzada [OPCIONAL]

El siguiente cambio afecta a más partes del juego, por lo que es más delicado. Vamos a incorporar un nuevo tipo de objeto al juego que son los suns. En la práctica anterior los *sunflowers* actualizaban el número de *suncoins* automáticamente. Ahora el jugador tendrá que cogerlos con el nuevo comando catch x y.

- El *sunflower* coloca el sun en su casilla.
- Además, aleatoriamente cada 5 ciclos se genera un sun en una posición aleatoria del tablero.
- No puede haber dos soles en la misma casilla
- Sí puede haber una planta o un zombie en la misma casilla de un sol.
- El usuario tiene que coger los soles; solo se puede coger un sol por turno. Cada sol suma 10 suncoins como en la práctica anterior.

Los soles se representarán con el símbolo *. Como puede haber una planta o un zombie en la misma casilla de un sol, e incluso los zombies pueden caminar por encima de los soles, a la hora de pintar el tablero se tendrá que generar de la siguiente manera, concatenando el símbolo del sol con el del objeto de juego que haya en la casilla:

```
Command >
Number of cycles: 9
Sun coins: 80
Remaining zombies: 8

 -------------------------------------------------------------
|S [2] *|       |       |       |       |W [8]  |       |       |
 -------------------------------------------------------------
|       |       |       |       |       |X [2] *|       |       |
 -------------------------------------------------------------
|       |       |       |       |       |       |       |       |
 -------------------------------------------------------------
|   *   |       |       |       |       |       |       |       |
 -------------------------------------------------------------
```

Para implementar esta parte vamos a proceder con los siguientes cambios:

- Vamos a añadir una clase Sun.
- Tendremos que volver a reorganizar nuestra jerarquía de objetos de juego. Vamos a tener objetos activos y pasivos. Los activos son las plantas y los zombies, es decir, los que teníamos en la versión anterior, y los pasivos son aquellos que no se actualizan y sobre los que se pueden caminar. Pasivos solo vamos a tener la clase Sun, pero podríamos extender la práctica con más objetos pasivos como lápidas, agua,... como ocurre en el juego original.
- La clase Sun heredará de PasiveGameObject, que a su vez hereda de GameObject.
- Los Sunflowers, en lugar de actualizar el número de *suncoins*, generarán un nuevo Sun.

```
public void update() {
  if (timeToNextAction() == 0) {
    game.addSun(x, y);
    this.nextSun = game.getCycleCount() + SUN_FREQUENCY;
  }
}
```

- El game le delega la creación del Sun al SuncoinManager (que desde ahora vamos a renombrar como SunManager ya que sus reponsabilidades van a aumentar).
- El SunManager mantiene una lista con los suns y su método addSun será así:

```
public void addSun(Sun sun, int x, int y) {
  if(suns.isPositionEmpty(x, y)) {
    sun.setPosition(x, y);
    suns.add(sun);
  }
}
```

- Para la clase que representa una lista de suns deberemos copiar mucho código de la lista de objetos de juego. Hay una funcionalidad de Java que permite definir listas genéricas, lo cual nos permitiría reutilizar nuestra clase de listas tanto para los objetos de juego como para los soles sin duplicar código. Esta funcionalidad la verás más adelante en el curso, pero por el momento deberemos duplicar código.
- La clase SunManager también va ser la encargada de generar un Sun en una posición aleatoria cada 5 ciclos. Para ello vamos a tener un método update que se encarga de ello.
- Para pintar las casillas, el juego concatenará el String de los objetos de juego con el String de los soles en el caso de que lo hubiera. Así que tendremos que definir un método en el game parecido a este:

```
public String positionToString(int i, int j) {
   String sunString = sunManager.positionToString(i, j);

   GameObject object = objectList.getPosition(i, j);

   if(object != null)
     return object.toString() + " " + sunString;
```

```
    return " " + sunString;
}
```

- Crearemos el comando catch que delegará en el juego, que a su vez delegará en el SunManager, en el que tendremos el siguiente método:

```
public boolean catchSun(int x, int y) {
  Sun sun = (Sun) suns.getPosition(x, y);
  if(sun != null){
    sun.catchSun();
    suns.removeDeadGameObjects();
    return true;
  }
  return false;
}
```

Estos cambios deberán ser suficientes para integrar la funcionalidad.

5. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica. Debes subir un fichero comprimido (.zip) que contenga al menos lo siguiente[4]. 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.
- Directorio `doc` con la documentación de la práctica generada con javadoc, si así lo requiere el profesor.
- Fichero `alumnos.txt` donde se indicará el nombre de los componentes del grupo.

[4]Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse.