Práctica 2 parte I: Road Fighter Refactored

Fecha de entrega: 8 Noviembre 2021, 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 la Práctica 1. En particular, en esta práctica incluiremos las siguientes mejoras:

- en la *Parte I*, refactorizamos[1] el código de la práctica anterior. Así lo preparamos para la *Parte II*. Para ello, modificaremos parte del controlador, distribuyendo su funcionalidad entre un conjunto de clases mejor estructurado para facilitar las extensiones posteriores.
- El objetivo es que, al finalizar la refactorización, la práctica pase los mismos tests que los que se pasaron en la versión anterior, esto es, en la Práctica 1.
- Vamos a hacer uso de la herencia para reorganizar los objetos del juego. Hemos visto que hay mucho código repetido en los distintos tipos de objetos. Por ello, vamos a crear una estructura de clases que nos permita extender fácilmente la funcionalidad del juego.
- En la *Parte II*, una vez refactorizada la práctica, añadiremos nuevos objetos al juego y nuevos comandos de una forma segura, ordenada y fiable, gracias a la nueva estructura del código.
- 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

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

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.

2. Refactorización de la solución de la práctica anterior

2.1. Patrón Command

En la práctica anterior, el usuario podía hacer varias acciones: mover arriba y abajo, avanzar, pedir ayuda o información, etc. El objetivo técnico de esta *Parte I* es poder añadir nuevas acciones sin tener que modificar código ajeno a la nueva acción. Para ello, vamos a ver el patrón de diseño Command[2] que es perfecto para este tipo de estructuras; la idea general es encapsular cada acción del usuario en su propia clase. Cada acción será un comando, de tal manera que el comportamiento de un comando está completamente aislado del resto.

En el patrón Command van a intervenir las siguientes entidades, que explicaremos en varios pasos según profundizamos en los detalles:

- Clase Commmand. Es una clase abstracta que encapsula la funcionalidad común de todos los comandos concretos.
- Comandos concretos. Son las acciones del usuario: MoveUpCommand, HelpCommand, ExitCommand... cada acción va a tener su propia clase.
- Cada comando tiene dos métodos básicos:
  - parse: es el método que comprueba si una acción introducida por teclado corresponde a la del comando.
  - execute: ejecuta la acción del comando, modificando el Game.
- Clase Controller. El controlador en este caso va a ser muy reducido, como veremos más abajo su funcinalidad va ser delegada en los comandos concretos.

Bucle del Juego. 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. En la nueva versión, el 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;
```

[2]Lo que vamos a ver en esta sección no es el patrón Command de manera rigurosa, si no una adaptación que hemos hecho a las necesidades de la práctica.

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

En el bucle, mientras el juego no termine, leemos una acción de la consola, la parseamos para obtener el comando correspondiente, ejecutamos el comando y, si la ejecución es satisfactoria y ha cambiado el estado del juego, lo repintamos. En otro caso, si el comando no es válido, mostramos un error.

Detalles de la Clase Command. En el bucle mostrado, la parte más importante es esta línea:

Command command = Command.getCommand(parameters);.

El punto clave es que el controlador sólo maneja comandos abstractos, por lo que no sabe qué comando concreto se ejecuta y qué es lo que hace exactamente el comando. Este es el mecanismo que nos facilita añadir fácilmente nuevos comandos concretos.

El método getCommand es un método estático de la clase Command, encargado de encontrar qué comando concreto corresponde a la entrada del usuario. Para ello tenemos una lista AVAILABLE_COMMANDS de objetos con los comandos disponibles. El método recorre la lista de comandos para determinar, llamando al método parse de cada comando, cuál corresponde a la entrada del usuario. Cuando lo encuentra, devuelve ese comando al controlador.

El esqueleto del código es este:

```
public abstract class Command {

    private static final String UNKNOWN_COMMAND_MSG = "Unknown command";

    protected static final Command[] AVAILABLE_COMMANDS = {
        new HelpCommand(),
        new InfoCommand(),
        //...
    };

    public static Command getCommand(String[] commandWords) {
        //...
    }

    //...
}
```

El Controller después de recibir un Command, simplemente ejecutará el método command.execute(game).

Detalles de los Comandos Concretos. Todos los comandos tienen un nombre, una información, etc... por ejemplo el comando concreto HelpCommand:

```
public class HelpCommand extends Command {
    private static final String NAME = "help";
    private static final String DETAILS = "[h]elp";
    private static final String SHORTCUT = "h";
    private static final String HELP = "show this help";
    public HelpCommand() {
        super(NAME, SHORTCUT, DETAILS, HELP);
    }
    // ...
```

Los comandos heredan de la clase Command, cuya definición tendrá esta forma:

```
    // ...


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

    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);
    protected abstract Command parse(String[] words);
    // ...
}
```

Como ya mencionamos, la clase Command es abstracta, son los comandos concretos los que implementan la funcionalidad:

- El método execute realiza la acción sobre elgame y devuelve un valor de tipo *boolean* que dice si se debe repintar o no el tablero.
- El método parse devuelve una instancia del comando concreto. Como cada comando se parsea a sí mismo, este método devolverá this o creará una nueva instancia de la misma clase. En el caso de que el texto introducido por el usuario no corresponda con el comando, entonces el método parse devolverá null.

Como hay muchos comandos que tienen el parse similar (hacer la comparación o *match* del name o del shortcut), implementamos un método por defecto en Command para que no lo tengan que reimplementar todos los comandos, sólo aquellos que sean diferentes al ya implementado.

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

protected Command parse(String[] words) {
    if (matchCommandName(words[0])) {
        if (words.length != 1) {
            System.out.format("[ERROR]: Command %s: %s%n%n", name,
                    INCORRECT_NUMBER_OF_ARGS_MSG);
            return null;
        } else {
            return this;
        }
    }
    return null;
}
```

Los comandos que tengan parámetros, como save <filename>, tendrán que sobreescribir el código del parse por defecto (lo haremos en la siguiente práctica).

Comando Reset

Vamos a modificar ligeramente el comportamiento del comando reset de la Práctica 1 con el objetivo de facilitar las pruebas, de modo que sea posible cambiar el nivel y la semilla de juego sin tener que parar y volver a arrancar el juego.

Ten en cuenta que al resetear el juego, también se debe reiniciar la instancia de Random que se utiliza para generar la partida.

2.2. Herencia y polimorfismo

Con el patrón Command el objetivo es poder introducir nuevos comandos sin cambiar el código del controlador. También se busca introducir nuevos objetos de juego sin tener que modificar el resto del código. La clave es que el Game no maneja objetos específicos, sino que maneja objetos de una entidad abstracta que vamos a a llamar GameObject. De esta entidad abstracta heredan el resto de objetos del juego. Como todos los elementos del juego van a ser GameObjects, compartirán la mayoría de atributos y métodos, y cada uno de los objetos concretos será el encargado de implementar su propio comportamiento.

Todos los GameObjects tienen una posición en la carretera y una serie de métodos que llamamos durante cada ciclo del juego, por ejemplo, cuando necesitan hacer algo propio de ese objeto en un momento concreto de su ciclo de vida:

- onEnter: Se llama cuando el objeto entra en el juego.
- update: Se llama en cada bucle del juego.
- onDelete: Se llama cuando el objeto sale del juego, desapareciendo.
- isAlive: Es verdadero si el objeto sigue vivo, o falso, si hay que eliminarlo del juego.

Es normal que en objetos sencillos haya algunos de estos métodos vacíos o con funcionalidad trivial. Por ejemplo, los dos objetos de juego que teníamos en la primera práctica Obstacle y Coin son básicamente lo mismo, la única diferencia entre el uno y el otro es que se dibujan de manera diferente y que cuando el coche choca con ellos el efecto es diferente.

A continuación se muestra el esqueleto del código de la clase GameObject. Más adelante describimos el uso del interfaz Collider para las colisiones.

```
public abstract class GameObject implements Collider {

    protected int x, y;

    protected Game game;

    protected String symbol;

    public GameObject(Game game, int x, int y) {
        this.x = x;
        this.y = y;
        this.game = game;
    }

    protected String getSymbol() { return symbol; }

    public int getX() { return x; }

    public int getY() { return y; }

    public boolean isInPosition(int x, int y) {
        return this.x == x && this.y == y;
    }

    public abstract void onEnter();

    public abstract void update();

    public abstract void onDelete();

    public abstract boolean isAlive();

    @Override
    public String toString() {
        // your code
    }
}
```

Este código lo tendrás que ir extendiendo y modificando a lo largo de las prácticas, para cada objeto del juego que hereda de clase GameObject

2.3. Game object container

En nuestra práctica queremos que el Game sea lo más simple posible y, aunque es la clase principal de nuestro programa, **su labor (responsabilidad) es coordinar al resto de las clases**, y lo hace delegando. La delegación consiste en lo siguiente: cuando están correctamente programados los métodos de Game son muy pequeños y lo que hacen es llamar a los métodos de otros objetos (colaborar) que son los que realmente hacen el trabajo. Uno de los objetos en los que delega es GameObjectContainer.

El GameObjectContainer es el almacén de objetos del juego. Es el encargado de actualizarlos, borrarlos, etc. (para acortar escribiremos Container o contenedor) Para el almacén podemos utilizar cualquier tipo de colección. Nosotros por simplicidad vamos a usar un

Arraylist de GameObjects cuya declaración es así:

```
public class GameObjectContainer {
    private List<GameObject> gameobjects;
    public GameObjectContainer() {
        gameobjects = new ArrayList<>();
    }
     ...
}
```

Es muy importante que los detalles de la implementación del GameObjectContainer sea privada. Eso permite cambiar el tipo de colección sin tener que modificar código en el resto de la práctica. El jugador, o Player, también es un objeto del juego, pero es peculiar porque el Game lo maneja como un objeto concreto. Por eso no lo meteremos en el Container.

En relación a la primera práctica, hay varios aspectos que van a cambiar en la estructura de esta práctica:

- Sólo tenemos un contenedor para todos los objetos concretos de juego.
- Desde el Game y el Container sólo manejamos abstracciones de los objetos, por lo que no podemos distinguir de qué clase son los objetos que están dentro del contenedor una vez añadidos.
- Toda la lógica del juego estará en los objetos de juego. Cada clase concreta conoce sus detalles acerca de cómo se actualiza, qué pasa cuando choca, etc.
- Para asegurarnos de que el Game está bien programado, no podrá tener ninguna referencia a GameObjects concretos, solo podrá tener referencias al Player y al Container.

2.4. Variables Estáticas

Una información que se muestra en el juego es el número de obstáculos y el número de monedas que hay en la carretera. Llevar esta contabilidad desde el Game o Container es complicado sin romper la abstracción, ya que tendríamos que tener un contador para cada clase de objetos e ir actualizándolo cada vez que añadimos un nuevo objeto, o lo eliminamos. Así que la manera correcta de hacerlo es que sea la propia clase Obstacle o Coin quien se encargue de hacerlo. Para eso usaremos variables estáticas y los métodos onEnter o onDelete. Cada vez que un objeto entra en el juego incrementa el contador estático de sus instancias, y cada vez que sale del juego, lo decrementa.

En general las variables estáticas son una manera muy efectiva de controlar el comportamiento de diferentes instancias de objetos de la misma clase, ya que podríamos decir que una variable estática es una especie de variable global o compartida por todos los objetos de la clase.

2.5. Object Generator

En las sección anterior decíamos que el Game sólo maneja instancias abstractas de GameObject, pero ¿Dónde creamos estas instancias?. Esta cuestión no es trivial. Ya sabemos que los objetos del juego los añadimos a la carretera cuando comienza la partida. Veremos más adelante que también los podemos crear durante la ejecución (runtime). Lo importante es quién debe crear los objetos ¿El Controller? ¿El propio Game? o ¿Quien?.

Una práctica muy habitual y muy aconsejable es tener un objeto encargado de la creación de objetos. A este tipo de objetos se les conoce como Factorías y son también un patrón de diseño muy conocido. Al igual que con el patrón Command, no vamos a estudiar el patrón de manera rigurosa sino que vamos a adaptarlo a nuestras necesidades concretas.

Para ello usamos una clase GameObjectGenerator cuya misión es introducir objetos en el Game (acortando usaremos Generator). Al principio de la partida llamaremos a su método generateGameObjects para rellenar la carretera con objetos. Veamos el esqueleto de la clase:

```
public class GameObjectGenerator {

    public static void generateGameObjects(Game game, Level level) {

        for(int x = game.getVisibility()/2; x < game.getRoadLength(); x ++) {

            game.tryToAddObject(new Obstacle(game, x, game.getRandomLane()), level.
                obstacleFrequency());
            game.tryToAddObject(new Coin(game, x, game.getRandomLane()), level.coinFrequency());

        }
    }

    public static void reset(Level level ) {
        Obstacle.reset();
        Coin.reset();
    }

}
```

El método public void tryToAddObject(GameObject o, double frequency) del Game añade un elemento al juego, si la casilla está libre y el número aleatorio lo dictamina. Además de crear los objetos, el Generator también es el encargado de resetear los contadores estáticos, de generar el status de las clases (cuantos obstáculos y coins hay). El Generator es el único objeto que *conoce* que tipos de objetos concretos hay.

2.6. Collider y callbacks

Ya hemos resuelto la abstracción de los objetos, el almacenamiento y la creación, ahora nos queda una cuestión muy importante y quizás la más compleja. Para ello debes entender bien el problema.

Al usar la clase abstracta GameObject, una vez que un objeto se mete en el juego ya no sabemos qué clase de objeto es[3]. Así, el problema es el siguiente: cuando el coche choca contra algo no sabemos si es un Obsctacle o un Coin, de forma que no sabemos si tenemos que morir o añadir monedas.

Para resolver este problema vamos a hacer lo siguiente. En primer lugar vamos a usar un interfaz Collider para encapsular los métodos relacionados con las *colisiones*. La clase GameObject implementará dicho interfaz. El objetivo es que todos los objetos del juego deben tener la posibilidad de colisionar o de recibir colisiones.

```
public interface Collider {
    boolean doCollision();
```

[3] podríamos saberlo usando instanceOf o getClass(), pero eso está terminantemente prohibido en la práctica.

```
    boolean receiveCollision(Player player);
}
```

Las colisiones se podrían comprobar desde Game, desde el Player o desde los objetos del juego pasivos Coin y Obstacle. Como el Player es el único elemento que se mueve, entonces la mejor opción es chequear las colisiones en el update del Player. Así, después de avanzar, comprobaremos si hay alguna colisión con un objeto. Esto lo podemos hacer de la siguiente manera:

```
// Player
public void doCollision() {
    ...
    GameObject other = game.getObjectInPosition(x, y);
    if (other != null && other.getClass() == "Coin") {
        coins += 1;
        ((Coin) other).setAlive(false); // remove from road
    }
    if (other != null && other.getClass() == "Obstacle") {
        life  = 0;
    }
    ...
}
```

Aunque te pueda parecer que el código es correcto (de hecho funciona), es un ejemplo de mala aplicación de la programación orientada a objetos. Este ejemplo de código, por un lado rompe la abstracción y encapsulación (ha sido necesario crear un mutador setAlive) y por otro, hace que el código sea poco mantenible porque tendremos que modificar el Player para cada nuevo tipo de objeto.

Otra opción que estaría igual de mal consiste en implementar métodos que simulen el comportamiento de Object#getClass() o del operador instaceof:

```
// Player
 public void doCollision() {
    ...
    GameObject other = game.getObjectInPosition(x, y);
    if (other != null && other.isCoin()){
        coins += 1;
        ((Coin) other).setAlive(false); // remove from road
    }
    ...
}
```

Ambos ejemplos muestran uno de los errores habituales de la programación orientada a objetos: reidentificar el tipo del objeto que estamos procesando y utilizar una instrucción condicional para aplicar un comportamiento y, además, la clase Player está acumulando demasiadas responsabilidades que no debería tener.

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. Para ello vamos a usar el interfaz que veíamos arriba, de la siguiente manera:

```
// Player
public boolean doCollision() {
    GameObject gameObject = game.getObjectInPosition(x, y);
    if (gameObject != null) {
```

```
        return gameObject.receiveCollision(this);
    }
    return false;
}
```

Todos los objetos implementan receiveCollision(Player player), y es precisamente en ese método donde debemos implementar la lógica que gestiona la colisión. Por ejemplo, en el receiveCollision de Coin le daremos una moneda al Player.

El método receiveCollision debe devolver false por defecto, es decir, no hace nada.

Esta solución es un comienzo ya que cada objeto sabe cómo colisiona. El problema es que rompemos la encapsulación al devolver un objeto GameObject de game. Aunque la clase GameObject implemente la interfaz Collider, es recomendable que siempre que se pueda interactuar utilizando los métodos definidos en la interfaz y no a través de una clase que implemente dicha interfaz. Cabe recordar que en Java sólo tenemos disponible herencia simple, **pero es posible implementar diferentes interfaces** que definen diferentes contratos dentro de la aplicación.

Para solucionar el problema debemos hacer que dos objetos sólo se comuniquen a través del interfaz, que es una abstracción o contrato entre ellos. Para ello vamos a usar la siguiente estructura:

```
// Player
public boolean doCollision() {
    Collider other = game.getObjectInPosition(x, y);
    if (other != null) {
        return other.receiveCollision (this);
    }
    return false;
}
```

El código es muy similar a la anterior, pero usamos el interface como tipo de datos, así ya no rompemos la encapsulación, ya que sólo se conectan con abstracciones.

Por el momento, el interfaz Collider es muy sencillo pero en las extensiones tendremos que añadir nuevos métodos para implementar interacciones más complejas.

3. Conclusión

Recuerda que una vez terminada la refactorización, la práctica debe funcionar exactamente igual que en la versión anterior y debe pasar los mismos tests, aunque tendremos muchas más clases.

Así, conseguimos dejar preparada la estructura para añadir fácilmente nuevos comandos y objetos de juego en la *Parte II*.