

Assignment 2, part I: Road Fighter Refactored

Submission date: [Part I, refactoring] 8th November 2021, 09:00
(non-assessed voluntary submission).
[Parts I & II, refactoring and game extensions] 29th November 2021, 09:00
(assessed obligatory submission).

OBJECTIVE: Inheritance, polymorphism, abstract classes and interfaces.

1. Introduction

In this assignment, we apply the mechanisms that OOP offers to improve the code developed in the previous assignment in the following ways:

- As explained in Section 2, we refactor¹ the code of the previous assignment in the following two ways:
 - First, by removing some code from the controller `run` method and distributing its functionality among a set of well-structured classes. This involves applying what is known as the *command pattern*.
 - Second, by defining an inheritance hierarchy, the leaves of which will be the classes that were used in the previous assignment (`Player`, `Obstacle` and `Coin`) to represent the different elements of the game. The use of inheritance enables us to avoid having identical, or nearly identical, code in different classes. It also enables us to reorganise how we store the information about the state of the game, using a single data structure instead of two different lists.

The aim of these changes is to produce code that is more extensible, adaptable, maintainable, etc. without changing the functionality. Note that if the functionality has not changed, the refactored code should pass the same tests as the code submitted for Assignment 1.

¹Refactoring means changing the structure of code (to improve it, presumably) without changing what it does.

- Once the code has been refactored, we add new commands and new elements to the game, in the process demonstrating how the new structure facilitates these extensions.

Important Note: We will publish the problem statement in installments. This first installment is only concerned with refactoring.

2. Refactorising the Solution to the Previous Assignment

2.1. The Command pattern

In the previous assignment, the user could enter several different commands: move up, move down, advance, ask for help, etc. The objective of the first part of the refactoring is to introduce a structure to the part of the code that is concerned with processing the user commands which will facilitate the addition of new commands, i.e. which will enable new commands to be added with minimal modifications to the existing code. This structure is the well-known software design pattern² known as the **Command pattern**³. The general idea is to encapsulate each user action in its own class, where in this case the user actions are the commands.

The following classes are involved in our application of the **Command** pattern:

- **Command** class: an abstract class that encapsulates the functionality that is common to all the concrete commands.
- **Concrete command classes:** `MoveUpCommand`, `HelpCommand`, `ExitCommand`... one class for each concrete command. Each command has (at least) the following two main methods:
 - **parse:** checks to see if the minimally-processed input text (introduced by the user via the keyboard) passed as a parameter corresponds to a use of the command that the class represents; if it does, the method returns an instance of that command class (this may, or may not, be by returning the value `this`), otherwise, it returns `null`.
 - **execute:** executes the functionality associated to each command by calling a method of the **Game** class (it may also perform some other actions); it returns a **boolean** value that tells the controller whether or not the game state has changed so the controller then knows whether or not to print the current game state to the output.
- **Controller.** The code of the controller is now reduced to only a few lines since most of the functionality that it included in Assignment 1 is now delegated to the concrete command classes.

Main loop of the game. In the previous assignment, in order to know which command to execute, the `run` method of the controller contained a switch or if-else ladder whose options correspond to the different commands. In the reduced version of the controller,

²You will study software design patterns in general, and this software design pattern in particular, in the Software Engineering course.

³Strictly speaking, we use a slightly-modified version of the Command pattern, adapted to the needs of this assignment

the `run` method has the following aspect (your code does not have to be exactly the same but should be similar):

```
while (!game.isFinished()){
    if (refreshDisplay) printGame();
    refreshDisplay = false;
    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);
}
```

Basically, while the game is not finished (due to internal game reasons or to a user exit), this code reads the text for a command from the console, parses this text and then executes it and, if the execution of the command terminates correctly and causes the state of the game to change, prints the new game state.

Details of the `Command` class. The most important part of the code for the main loop shown above is the following line:

```
Command command = Command.getCommand(parameters);
```

The key point is that the controller only handles abstract commands and does not know which concrete command is being executed; the knowledge of what functionality corresponds to each command is contained in each concrete command class. It is this mechanism that facilitates the addition of new concrete commands without changing the existing code.

The `getCommand` method is a static method of the `Command` class, charged with finding the concrete command that corresponds to the text entered by the user. To this end, the `Command` class contains a static attribute whose value is an array containing one instance of each of the available command classes and the `getCommand` method traverses this array calling the `parse` method of each command. If any of these methods returns a non-null value (an instance of one of the command classes), `getCommand` returns this non-null value, otherwise it returns the value `null`. The following is a skeleton of this code:

```
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) {
        //...
    }

    //...
}

```

The **Controller**, on receiving a concrete command object returned by the `getCommand` method, simply calls the `execute` method of this command object, passing the `game` object (the single instance of the `Game` class used in the application) as a parameter.

Details of the concrete commands. All the commands have a certain number of constants defined and the constructor of the concrete command class passes the value of these constants to the constructor of the abstract `Command` class. For example, the code of the `HelpCommand` should have the following aspect:

```

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);
    }
    // ...
}

```

Recall that each of the concrete command classes inherits from the abstract class `Command`, where the latter class has the following aspect:

```

// ...

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);
// ...
}

```

Since the structure of the `parse` method is identical in all the commands that do not have parameters (so far we do not have any commands with parameters but, as an example, we will later introduce a save command: `save <filename>`), the only difference being the name of the command being matched, we can implement a default parse method in the

Command class that can be overwritten by any commands that need a **parse** method with a different structure.

```
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;
}
```

Reset Command

We modify the behaviour of the Assignment 1 **reset** command slightly with the aim of facilitating testing. The idea is to be able to change the level and the seed without having to stop and restart the game. We also want to reinitialise the random-number generator encapsulated in an instance of the Java library **Random** class, which we can do by simply creating another **Random** object.

2.2. Inheritance and polymorphism

We have seen that the use of the **Command** pattern greatly facilitates the introduction of new commands, the key aspect being that the **Controller** class is generic, i.e. it does not handle specific commands but only handles objects of the abstract class **Command**. Similarly, to facilitate the introduction of new game elements, the key aspect is that the **Game** class be generic, i.e. it should not handle specific game elements but only handle objects of an abstract class called **GameElement**, from which all the concrete game element classes derive. So the game cannot distinguish what type of game element it is handling. This abstract class should contain all the attributes and methods that are common to all the concrete game elements and, where appropriate, each concrete game element can overwrite the methods to implement its own behaviour.

Each concrete game element has a position on the road and a series of methods that are called on each game cycle:

- **onEnter**: called when the instance of that game element enters the game.
- **update**: called on each cycle of the game.
- **onDelete**: called when the instance of that game element leaves the game.
- **isAlive**: when it returns false, this indicates that this instance of the game element must be eliminated from the game.

It is to be expected that in some `GameElement` classes, these methods will be empty or will have trivial functionality. In this regard, the two `GameElement` classes of the first assignment, `Obstacle` and `Coin`, have a great deal of behaviour in common, the only real difference between them (apart from their textual representation) is that the effect of a collision with the car is different.

Below is a skeleton of the code of the `GameElement` class. The `Collider` interface that it implements is described later in this document.

```
public abstract class GameElement implements Collider {

    protected int x, y;

    protected Game game;

    protected String symbol;

    public GameObject(Game game, int x, int y) {
        // this would be different if you use a Position class
        this.x = x;
        this.y = y;
        this.game = game;
    }

    protected String getSymbol() { return symbol; }

    public int getX() { return x; }

    public int getY() { return y; }

    // this would be different if you use a Position class
    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
    }
}
```

You will need to extend and modify the code of the `GameElement` class throughout the assignments.

2.3. Game element container

In this assignment, we want the code of the `Game` class to be as small and simple as possible, even though it is the principle class of our application, tasked with coordinating

the rest of the classes. One way to achieve this is for the **Game** class to *delegate* tasks to other classes (via calls to methods of these other classes), thereby ensuring that its own methods are kept small. One of the objects to which the **game** object will delegate tasks is an object of class **GameElementContainer**, which, since there will only be one such instance in the entire application, we will call the **container** object.

The **GameElementContainer** is the store for game elements. It is responsible for updating them, deleting them, etc. Like the **Game** class, the **GameElementContainer** class is generic, in the sense that it does not handle specific game elements, only objects of the abstract class **GameElement**, so that, once an object is added to the container, the container cannot distinguish what type of game element it is handling. It can be implemented using any collection type; for simplicity, we will use an **ArrayList** of **GameElements** declared as follows:

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

Recall that, following the principle of information hiding, the implementation details of the **GameElementContainer** should be private, allowing the type of collection used to be changed without modifying the code of any other classes.

The player (the single object of the class **Player** that exists in the application) is also a game element but it is the one game element that the **Game** handles explicitly, for which reason it is not managed by the container.

Some observations:

- The logic of the game is in the game elements. Each concrete game element class knows how objects of that class should be updated, what happens to them when they are involved in a collision, etc.
- To be sure that the **Game** class is properly programmed, you must ensure that it does not have any reference to concrete game elements, i.e. to objects whose type is a specific subclass of the **GameElement** class, except for the reference to the **player** object.

to be continued...