Práctica 2: Buffy the Vampire Slayer, Refactored and Extended

Submission date:
[Part I, refactoring] 30th Noviembre 2020, 9:00 (non-assessed submission)
[Parts I & II, refactoring and game extensions] 14th December 2020, 9:00.

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 refactor[1] 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 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 (`Vampire` and `Slayer`) 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.
- Once the code has been refactored, we add new commands and new elements to the game.

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

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

2. Refactorising the Solution to the Previous Assignment

The best-practice programming rule *fat models and skinny controllers* refers to keeping controllers lightweight by ensuring that most of the functionality is handled by the part of the code referred to as the model[2]. The *Command pattern* helps to achieve a skinny controller by encapsulating the commands and their associated actions in a uniform way while at the same time being highly extensible, making it possible to add new commands without having to modify the controller in any way.

The body of the controller run method will now have roughly the following aspect (your code does not have be 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[] commandWords = s.toLowerCase().trim().split("\\s+");
    System.out.println("[DEBUG] Executing: " + s);
    Command command = CommandGenerator.parse(commandWords);
    if (command != null)
            refreshDisplay = command.execute(game);
    else
        System.out.println("[ERROR]: "+ unknownCommandMsg);
}
```

Basically, while the game is not finished (due to internal game reasons or to a user exit), this code reads a command from the console, parses this command and then executes it. Notice that this controller could be used with different versions of the game or even with a different game. In the next section we see how the Command pattern works.

2.1. Patrón Command

The Command pattern is a well-known software design pattern[3]. For this assignment, you do not need to know any more detail about this pattern than that provided here. In the application of this pattern, each command is represented by a separate class, here named AddCommand, ExitCommand, ResetCommand, HelpCommand,... and each of these command classes inherits from an abstract class called Command. These concrete command classes execute the functionality associated to each command by calling methods of the Game class.

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. With the command pattern, to know which command to execute, the controller run method, after splitting the text entered by the user at the console into words[4], calls a method called, say, parseCommand of the *utility class*[5] CommandGenerator, passing the minimally-processed input text as a parameter. The parseCommand

[2] in this context, the terms controller and model refer to two of the elements of what is known as the *Model-View-Controller pattern* or *MVC pattern*

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

[4] In the case of commands with parameters, your program must **not** ask the user for the parameters seperately; to enter any command the user must only have to press return once.

[5] A class is said to be a utility if all of its methods are static.

method then passes this minimally-processed input text to an object of each of the concrete command classes in turn, in order to see which of them accepts it as valid text for that command. That is, each concrete subclass of the Command class has a parse method which checks to see if the minmally-processed input text passed as a parameter corresponds to a use of the command that the class represents and the parseCommand method calls each of these parse methods in turn.

If the parse method of a given concrete subclass of the Command class accepts the input text as valid, it returns an object of that concrete subclass[6], otherwise it returns the value null. If none of the parse methods of the concrete subclasses of the Command class accepts the input text as valid, i.e. all of them return the value null to the parseCommand method of the CommandGenerator, this latter method also returns the value null to the controller, which then informs the user that the text entered did not correspond to any known command.

With this mechanism, if the text entered by the user corresponds to a command of the system, the controller obtains an object of the class representing that command, which it can then use to execute the command (each concrete subclass of the Command class also has a method called execute). Notice that we could now add new commands without having to modify the existing code (except to let the CommandGenerator know the name of the new subclass) by simply adding new concrete subclasses.

Implementación

The code of the abstract class Command is as follows:

```
package control.Commands;

import logic.Game;

public abstract class Command {

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

    protected static final String incorrectNumberOfArgsMsg = "Incorrect number of
        arguments";
    protected static final String incorrectArgsMsg = "Incorrect arguments 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);
```

[6] If the command that the class represents has no parameters, the parse method can just return its owning object, i.e. the object on which the parse method was called.

```
    }

    public String helpText(){
      return details + ": " + help + "\n";
    }
}
```

As already stated, the abstract method execute is to be implemented by a method which calls some method of the game object passed as a parameter, and may also perform some other action. The abstract method parse is to be implemented by a method which parses the text received in its first argument (this being the text introduced by the user on the command line split into words) and returns:

- an instance of some subclass of Command, if the text passed in the first argument is an invocation of the command represented by that subclass
- the value null, otherwise.

As a further improvement, you may consider adding a NoParamsCommand class. With this class, those Command subclasses that represent commands without any parameters — in the present case, all of them except the AddCommand class — do not derive directly from the class Command, but instead derive from the class NoParamsCommand which itself derives from the class Command. The NoParamsCommand class implements the parse method using the matchCommandName method inherited from Command. In this way, the classes deriving from NoParamsCommand only need to implement the execute method. Command subclasses that represent commands with parameters derive directly from the class Command and will need attributes to store the value of their parameters.

The CommandGenerator class contains the following attribute declaration and initialisation:

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

which is used by both of the CommandGenerator methods, these being the following:

- public static Command parseCommand(String[] commandWords, Controller controller), which calls the parse method of each of the Command subclasses in turn, as explained above.
- public static String commandHelp(), which has a similar structure to the parseCommand() method but calling the helpText() method of each of the Command subclasses in turn. This method is called by the execute method of the HelpCommand class.

Note that another advantage of the *command pattern* is that it facilitates the implementation of an undo facility. The fact that commands are represented as instances of the Command class means that we can push and pop them from a stack structure in order to move back and forth in the game history (though, when doing so, care must be taken in a game with any random aspect). We will not implement this characteristic in this assignment but you should come across this use sometime in your degree.

2.2. Inheritance and polymorphism

The need for different classes, such as the elements of the game and the lists of these elements, to have some identical or nearly identical code is perhaps the most frustrating part of the previous assignment, as well as being a possible source of error. We are now going to resolve this problem using a basic tool of object-oriented programming: *inheritance*. To this end, we create a hierarchy of classes that inherit from the class GameElement.

- The class GameElement has attributes and basic methods to control the position on the board as well as having a reference to the Game class.
- The classes Slayer y Vampire inherit from GameElement; the first represents the player's defenders and the second represents the player's attackers, as in the previous assignment.

It is important to remark that in this version of the assignment, the major part of the logic is divided among the classes representing the different elements of the game, leaving the Game class free from this task, and thereby obtaining a scalable solution, i.e. a solution in which new game elements can easily be added.

Additionally, we can use inheritance to refactor the code of the lists of the previous assignment. One possible solution would be, instead of using lists like in the previous assignment, to store the elements of the game in a generic two-dimensional array — here, an array of GameElement objects — representing the board. A second possible solution would be to define a generic list — here, a list of GameElement objects — and then, using the covariance of Java arrays, define other list classes as subclasses of this generic list class and use an object of each of these subclasses to store the elements of the game. A third possible solution would be to define a generic list class and then use multiple instances of this generic class (rather than a single instance of each of the multiple subclasses of the second solution) to store the elements of the game.

However, we will use a fourth solution, in which we define a generic list class and then simply use an object of this class to store all the different elements of the game. Similar to in the previous assignment, the Board class[7] will manage the (single) list of GameElement objects and the Game class will have an attribute of class Board. Instead of using a basic Java array to implement the list of GameElement objects, you may use the library class ArrayList (contained in the package java.util).

```
import java.util . ArrayList;

public class Board {
   private ArrayList<GameElement> gameElements;
        ....
}
```

2.3. Implementation Details

The following aspects significantly change the structure of the code:

- There is only one container class for all the objects and only one instance of this container class is used.

[7]Note that the use of the word *board* does **not** imply the use of a two-dimensional array to manage the game elements.

- The classes Game and Board only deal with generic elements (i.e. of class GameElement) and so cannot distinguish the concrete class of the objects being manipulated.
- The logic of the game is distributed among the game element classes. Each concrete class knows how it moves, how it attacks, how it receives an attack and when to perform *computer actions*.

To help you with the necessary refactoring, we provide you with some guidance and a certain amount of code.

Regarding the Game class, similarly to the way in which the funcionality of the Controller class of the previous assignment was distributed among the different command classes used in this assignment, we now distribute much of the functionality of the Game among the different game element classes.

We will use an interface to encapsulate the methods related to attacks.

```
package logic.GameElements;

public interface IAttack {
    void attack();
    default boolean receiveSlayerAttack(int damage) {return false;};
    default boolean receiveVampireAttack(int damage) {return false;};
}
```

This interface represents the possibility that objects of any subclass of GameElement can attack, or can be attacked by, objects of other subclasses of GameElement, *can-attack / can-be-attacked-by* being a behavioural feature of a large number of video games. The interface includes a Java 8 *default method* for each of the GameElements that can perform attacks, representing an attack by that GameElement. A default method is one that includes a default body, where this means a body that may, or may not, be redefined; if an implementing class does not overwrite a default method, the body used for that method in that class is the default one provided in the interface. So if objects of a subclass A of GameElement can be attacked by those of subclass B of GameElement, subclass A overwrites the method corresponding to attacks by subclass B.

When we create game elements (slayers or vampires) and put them on the list of game elements, we lose the knowledge of what type of element the object is; only the object itself knows to which subclass of GameElement it belongs. So how do we implement the interaction between game elements if the game (and the board) don't know what type of elements they are? We have several options:

- Delegate the task to the game, which propagates the type of attack from the attacking object to the object to be damaged by the attack, as we did in the first assignment. This is a reasonable solution but it has the problem that as we add new types of object to the game, we need to add more and more delegation methods in the classes Game and Board. For example:

  ```
  // in the class Vampire:
  public void attack() {
      ...
      game.attackSlayerInPosition(x − 1, y, HARM);
      ...
  }
  ```

  We would prefer the functionality to be in the elements of the game themselves, in order to easily add new elements to the game with minimal changes to the existing code. Let's look at other options.

- Ask the game for the object and use instanceOf or getClass to find out which type of game element it is, for example:

```
// in the class Vampire:
public void attack() {
    ...
    GameObject other = game.getObjectInPosition(x - 1, y);
    if (other != null && other.getClass() == "Slayer")
        ((Slayer) other).decreaseLife(HARM);
    ...
}
```

  However, using instanceOf or getClass and casting in this way goes against OO principles, since we are using these constructions to avoid using the OO tools polymorphism and dynamic binding and thereby losing the flexibility that these tools provide. For this reason, *use of instanceOf or getClass is forbidden.*

- Ask the game for the object and implement methods isSlayer, isVampire to find out which type of game element it is, for example:

```
// in the class Vampire:
public void attack() {
    ...
    GameObject other = game.getObjectInPosition(x - 1, y);
    if (other != null && other.isSlayer())
        ((Slayer) other).decreaseLife(HARM);
    ...
}
```

  This solution is also forbidden since it is just a primitive "DIY" way of doing what was done in the previous alternative.

- Ask the game for the object and use the IAttack interface defined above, for example:

```
// in the class Vampire:
public void attack() {
    ...
    GameObject other = game.getObjectInPosition(x - 1, y);
    if (other != null)
        other.receiveVampireAttack(HARM);
    ...
}
```

  This alternative looks good since each game element knows how to attack and knows which other game elements can attack it. The problem is that it is breaking encapsulation to return objects that are values of (private) attributes (the two previous solutions also had this problem but they also had even bigger problems). We solve this problem by ensuring that two objects communicate via an abstraction defined in an interface.

- We uncouple Game and GameElement by ensuring that the type of the reference returned by Game is the abstraction defined by the interface IAttack. Concretely, this means that the receiver of the reference only has access to the methods of the object that are defined in the IAttack interface so this solution does not break encapsulation. For example:

```
public void attack() {
    if (isAlive()) {
        IAttack other = game.getAttackableInPosition(x − 1, y);
        if (other != null)
            other.receiveVampireAttack(HARM);
    }
}
```

This is, in fact, an application of the *dependency inversion principle* according to which code should be built to depend on abstractions, not on implementations[8]

We can also use interfaces to uncouple Game and GamePrinter: the Game class should implement the following interface:

```
public interface IPrintable {
    String getPositionToString(int x, int y);
    String getInfo();
}
```

The Gameprinter class can now be passed a reference of type IPrintable, instead of a reference of type Game, and it no longer needs to build a two-dimensional array of String objects (see the attribute called board). Its constructor can now be of the following form:

```
public GamePrinter (IPrintable printable, int cols, int rows) {
        this.printable = printable;
        this.numRows = rows;
        this.numCols = cols;
}
```

Thus, GamePrinter depends on the abstraction, i.e. the type defined by the interface IPrintable, representing strictly the functionality that it needs, and not on the type defined by the class Game, which represents the functionality that it needs together with a lot of other functionality that it is not interested in. The use of interfaces provides the GamePrinter with a way to say that it doesn't care what the class of the object passed in the first parameter of its constructor is, as long as this class implements the behaviour it needs. Of course, in our implementation, the contructor of the class GamePrinter will be passed an object of class Game but with the code defined above, it could be passed any other class that implements the interface IPrintable.

In the previous assignment, the slayers attacked before the vampires. In the refactorised version, the GameElements are all stored in a single list and will attack in the order in which they are stored, which is the order in which they are created. For this reason, the output of test cases such as *easy_s5_2.txt* may be different in this assignment.

IMPORTANT: Breaking encapsulation, using methods that return lists, and any use of instanceOf() or getClass() will automatically lead to the submission receiving a grade of fail. This also applies to any use of a "DIY" version of instanceof such as defining a set of methods isX, for each concrete subclass X of GameElement, where the method isA of a given subclass A returns true, while the other methods isX of class A return false (which would be even worse than directly using instanceOf() since it is the same but much more clumsy and verbose!)

[8] leading to code that is more flexible, maintainable, easier to evolve, etc.

3. Extending the Game

3.1. Incorporating New Game Elements

Now make the following modifications to your solution:

- Add a subclass of Vampire called Dracula. There can only ever be one instance of the Dracula class in the game, so that we can refer to this instance as Dracula. If he is not already on the board, Dracula can appear with the same probability as a normal vampire, independently of whether or not a normal vampire appears on that same cycle. Whenever Dracula is on the board, a new information message saying "`Dracula has arisen`" must be printed on a new line before printing the board. A slayer, or any other type of defender, is destroyed (loses all lives) by a bite from Dracula. Dracula is represented on the board by the string `V-V`

- Add a subclass of Vampire called ExplosiveVampire. An ExplosiveVampire can appear with the same probability as a normal vampire, independently of whether any other type of vampire appears on that same cycle[9]. On being destroyed (i.e. the moment its lives reach zero) it explodes causing damage to all neighbouring vampires (the same damage as caused by a bullet), including its diagonal neighbours. It is represented on the board by the string `V*V`.

- Add a defender class BloodBank whose cost is provided as a parameter but which, while it is on the board, generates 10% of this cost in coins for the player on each turn (rounded to an integer if necessary). To add an object of the class BloodBank, you must use the command `[b]ank <x> <y> <z>` where x and y are the coordinates of the tile on which the bloodbank is to be placed and z is the cost. Note that a bloodbank cannot be placed in the rightmost column. A bloodbank is completely drained by any vampire that arrives at the neighbouring tile on the same row. It is represented on the board by the string `B-B`.

- Add a "garlic push" command with syntax `[g]arlic`, which costs 10 coins. It has the effect of pushing all the vampires back one tile (i.e. one tile to the right) subject to the following rules:

  - Any vampire that has another game element immediately to its right is unaffected.
  - Any vampire in the rightmost column (including Dracula) is pushed off the board and eliminated.
  - Any explosive vampire in the rightmost column does not explode when pushed off the board.

  In addition, the garlic push command stuns the vampires causing them not to move until the next turn (their move counter is reset).

- Add a "light flash" command with syntax `[l]ight`, which costs 50 coins. It's effect is to eliminate all the vampires, except Dracula if present, from the board.

[9] The order in which it is checked if each type of vampire should appear can affect the outcome of the game — when there are less then three vampires left to appear, it would do so even if the pseudo-random behaviour were really random behaviour — so we impose the following order: first normal vampires, then Dracula, then explosive vampires.

Explosive vampires destroyed by a light flash do not explode (and therefore cannot destroy Dracula by exploding).

Note that with the addition of new types of vampire, the total number of vampires for each level remains the same. In order to facilitate testing your solution, you must also make the following modifications to your solution:

- Add a "super coins" command, with syntax `[c]oins`, which immediately gives the player 1000 coins. This enables the behaviour of different game elements to be tested without waiting for sufficient coins to accumulate in order to buy these game elements (while still checking that their coin-affecting behaviour is correct)
- Add an "add vampire" command with syntax `[v]ampire [<type>] <x> <y>` which places a vampire on a given tile of the board as long as that tile is not already occupied, where `x` and `y` are the coordinates of the chosen tile. If no type is indicated, the vampire placed on the board is a normal vampire, if the type is `D`, it is Dracula (as long as he is not already on the board), and if the type is `E`, it is an explosive vampire. This command does not cause the game to evolve (and the cycle number does not change) and if there are no vampires remaining, it has no effect.

If you have correctly refactored your solution according to the instructions in the first part of this assignment, the extensions of the second part should be relatively simple to implement. We will provide you with some test cases to check that these modifications produce the required output.