Assignment 1: Buffy the Vampire Slayer

Submission date: November 10th 2020, 09:00

Objective: Introduction to object orientation and to Java; use of arrays and enumerations; string handling with the `String` class; input-output on the console.

Copy Detection

For each of the TP assigmments, all the submissions from all the different TP groups will be checked using anti-plagiarism software, firstly by comparing them all pairwise and secondly, by searching to see if any of their code is copied from other sources on the Internet[1]. Any plagiarism detected will be reported to the *Comité de Actuación ante Copias* which, after interviewing the student or students in question, will decide whether further action is appropriate, and if so, will propose one of the following sanctions:

- A grade of zero for the TP-course exam session (*convocatoria*) to which the assignment belongs.
- A grade of zero for all the TP-course exam sessions (*convocatorias*) for that year.
- Opening of disciplinary proceedings (*apertura de un expediente académico*) with the relevant university authority (*Inspección de Servicios*).

1. Introducción

In this assignment, we create a vampire game inspired by the classic Tower Defense, in which the player can place characters with different attributes on the board to defend against the attacks of the vampires.

[1]If you decide to store your code in a remote repository, e.g. in a free version-control system with a view to facilitating collaboration with your lab partner, make sure your code is not in reach of search engines. If you are asked to provide your code by anyone other than your course lecturer, e.g. an employer of a private academy, you must refuse.

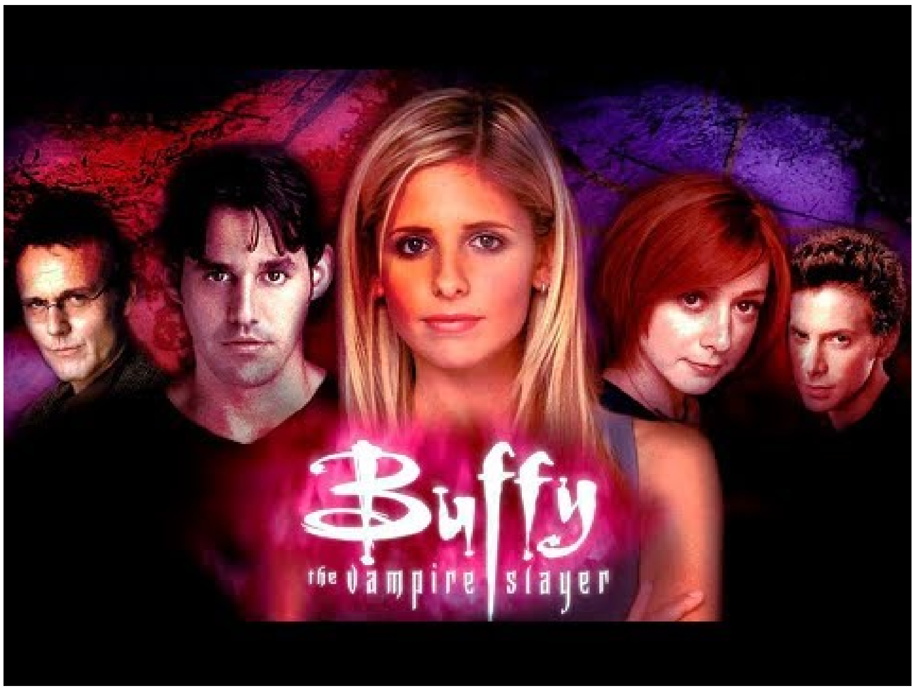


Figure 1: Portada del juego

From the wikipedia: *"Tower defense (TD) is a subgenre of strategy video game where the goal is to defend a player's territories or possessions by obstructing the enemy attackers, usually achieved by placing defensive structures on or along their path of attack.[1] This typically means building a variety of different structures that serve to automatically block, impede, attack or destroy enemies."*

The game is played on a board made of square tiles on which the player places vampire *slayers*, each with different characteristics. The vampires appear on the r.h.s. of the board and move towards the l.h.s. If one of the vampires manages to breach the defenses and reach the l.h.s. of the board, the player loses. To resist the vampire attack, the player must destroy all the vampires. We will start with a reduced version of the game and will progressively increase the complexity throughout the term by adding new types of slayers and vampires with different abilities.

2. Description

In our first version of *Buffy the Vampire Slayer*, there are only two types of characters (game elements): slayers and vampires. Each tile of the board can only have one character, a slayer or a vampire, on it. The dimensions of the board on which the game is played depends on the level. Initially, the board is empty. As already stated, the game finishes if, at the end of a turn, all the vampires have been destroyed or if one of the vampires has reached the l.h.s. of the board.

- ***Slayer***. Added by the player on a specified tile of the board. Costs 50 coins. Does not move. On each cycle, fires a silver bullet which travels from left to right[2] and

[2]We do not represent the shots visually

strikes the leftmost vampire in the same row as the slayer, if there is a vampire in that row. The effect of the impact of a silver bullet on a vampire is to decrease its lives by one.

- ***Vampire***. Appears in a randomly-chosen row of the rightmost column of the board. Moves one tile to the left every two turns, as long as the destination tile of the move is free. On each turn, it bites the slayer in the next tile to the left, if there is a slayer on that tile. The effect of a vampire bite on a slayer is to descrease its lives by one.

On each cycle of the game, the following actions are carried out in sequence:

1. ***Draw***. Sends the current state of the board and other game information to the standard output.
2. ***User action***. Accept input from the user (to add a new slayer, to simply advance the game,...)
3. ***Update***. Move the game elements on the board.
4. ***Attack***. Check attacks and decrease the lives of the game elements where necessary.
5. ***Add vampire***. Add a vampire with a probability that depends on the level.
6. ***Remove dead objects***. Eliminate any game elements that have no lives left.
7. ***Check end***. Check whether the game has ended.

The player starts the game with 50 coins and on each turn has a probability of 50% of receiving 10 coins.

3. Game Elements

In this section, we recapitulate the different features of the characters that can appear in the game.

Slayer

- **Behaviour**: Fires silver bullets at vampires in the same row.
- **Cost**: 50 coins.
- **Resistence**: 3 lives.
- **Frequency**: One shot per cycle.
- **Damage**: Each shot effects 1 point of damage on its target (removes one of its lives).
- **Reach**: Can only fire straight (along the row) and forwards (left to right).
- **Graphics**: Represented on the board by the ASCII text “<->”.

Vampire

- **Behaviour**: Moves from right to left, if able to do so; bites any slayer on the adjacent left tile.
- **Resistence**: 5 lives.
- **Damage**: Each bite effects 1 point of damage on its target (removes one of its lives).
- **Frequency**: At most one bite per cycle.
- **Speed**: 1 tile every 2 cycles.
- **Graphics**: Represented on the board by the ASCII text “`V^V`”

4. Actions

Next, we describe what occurs in the different parts of the game loop given above.

4.1. Draw

On each cycle, the current state of the board is printed on the standard output, together with the following information:

- the cycle number (initially 0),
- the number of coins that the player currently has
- the number of vampires currently on the board
- the number of vampires left to appear.

The board is printed using ASCII characters as shown in the following figure:

```
Cycle number: 20
Coins: 50
Remaining vampires: 1
Vampires on the board: 2

  ------------------------------------------------------------------------
 |        | <->[3] |        |        |        |        |        |        |
  ------------------------------------------------------------------------
 |        |        |        |        | V^V[5] |        |        |        |
  ------------------------------------------------------------------------
 | <->[3] |        |        |        |        |        | V^V[1] |        |
  ------------------------------------------------------------------------
 |        |        |        |        |        |        |        |        |
  ------------------------------------------------------------------------

Command >
```

The number in square brackets, to the right of the graphical representation of each game element (vampire or slayer), is the number of lives that element has left. After printing the board, the prompt used to elicit the next action from the user must be printed.

4.2. Update

The update actions that occur on each cycle are the following.

- The player receives 10 coins with a probability of 50%.
- The vampires that should move on this cycle do so.

4.3. Attack

- Los slayers fire their silver bullets at the vampires that can be shot.
- Los vampires bite the slayers that can be bitten.

4.4. Add Vampire

The game can be played at three levels, EASY, HARD and INSANE, where the level determines various configuration options (see Table 1.1), in particular, the probability on each cycle that a new vampire is added to the game. If a vampire is to be added, the row in which it appears is chosen at random. If there is already a vampire in the chosen row then the new vampire is not placed on the board[3].

Level	Number of vampires	Frecuency	board width	board height
EASY	3	0.1	8	4
HARD	5	0.2	7	3
INSANE	10	0.3	5	6

Table 1.1: Configuration for each level of difficulty

Table 1.1 shows the different values for the configuration options that depend on the level. These are:

- The total number of vampires that appear in a game.
- The frequency of appearance of vampires, which determines the probability that a vampire appears on a given cycle. Thus,if the frequency is 0.2, a vampire appears randomly on each cycle with a probability of one in five.
- The dimensions of the board; at the easiest level the board dimensions are 8×4 and at the hardest level 5×6.

4.5. User command

The player is prompted to enter a command, which should be one of the following:

- `add <x> <y>`: Adds a new slayer on the tile x, y. The command does not succeed if any of the following are true:
    - the player does not have enough coins,
    - the chosen tile is already occupied by a slayer or a vampire,

[3]In the case of the tile being occupied there are several options: use some strategy to find a free tile, place the vampire in an invisible queue to enter the row (with probability of 100%), or give up; here, we have chosen the last option.

  - the x coordinate corresponds to the rightmost column (since placing slayers in this column could block the appearance of new vampires).

- reset: Resets the game, i.e. takes it back to its initial configuration.
- none: Advances the game one cycle without the player taking any action.
- exit: Terminates the application after printing the message "Game Over".
- help: Shows the list of command names and brief descriptions, each on a different line, where each command name is separated from its description by a colon. That is, the format is as follows:

```
Command > help
Available commands:
[a]dd <x> <y>: add a slayer in position x, y
[h]elp: show this help
[r]eset: reset game
[e]xit: exit game
[n]one | []: update
```

Observations concerning the commands

- The application must allow commands to be written using any mixture of lower and upper case characters.
- The application must allow the player to use only the first letter of the command name instead of the complete name, i.e. [A]dd, [N]one, [R]eset, [H]elp, [E]xit.
- An empty command must be equivalent to the none command.
- If the syntax of the text entered by the player does not correspond to one of the known commands or has incorrect parameter values, an error message must be displayed.
- After the execution of a command which does not change the state of the game or which is erroneous, the board should not be displayed.

5. General observations

If all the vampires are destroyed, before terminating, the application prints "Player wins" on the console. If a vampire reaches the l.h.s of the board, before terminating, the application prints "Vampires win" on the console. Note that the order of updating of the different game elements may affect the outcome, for which reason we specify that the game elements must be updated in the following order: first slayers, then vampires.

To control the pseudo-random behaviour of the game and thereby enable the same execution to be repeated, the player can introduce a seed for this behaviour via an optional parameter of the program. If the player does not introduce a seed on starting the program, the program generates one automatically. Note that the mere existence of a seed is not sufficient to ensure that the psuedo-random behaviour can be controlled: in addition, the program should only create a single object of the library class Random to be shared by all objects of the program that need any random behaviour.

6. Implementation

We start this section by observing that the quality of the implementation proposed here is relatively low, one of the main reasons for this being that it does not stick to the **DRY (Don't Repeat Yourself)** programming principle. The duplication of code in different parts of a program makes it less maintainable, less readable and less testable; modifying such a program is considerably more complicated and error-prone. The reason that the DRY principle is not correctly followed in the proposed implementation is that in this first assignment, we do not want to use *inheritance* and *polymorphism*, two basic tools of object-oriented programming. In the second assignment, we will refactorise the code, improving it by introducing these tools, thereby converting it into a *bona fide* object-oriented program.

6.1. Implementation details

Executing the class `org.ucm.tp1.Buffy` starts the application, for which reason you are advised that all classes developed in the assignment should be inside the `org.ucm.tp1` package. Recall that that Java naming convention is that package names begin with a lower-case letter while class names begin with an upper-case letter.

To implement the assignment, you must use, at least, the following classes:

- Slayer, Vampire: These classes encapsulate the state and the behaviour of the game elements. Their state — position on the board, lives left, etc. — is contained in private attributes. They also have an attribute in which they store (a reference to) the game, i.e. an instance of the class Game, in order to be able to invoke the methods of this instance to consult as to whether or not they can perform a given action. Note that there will only ever be one instance of the Game class in the program, i.e. the one created in the main method, which we will refer to as the game object.

    - The class Vampire itself is responsible for managing
        - the total number of vampires that can appear in the game,
        - how many vampires are on the board,
        - whether or not the vampires have reached the l.h.s. of the board,

      using static attributes or methods. The game object will access these attributes/methods to display information or to terminate the game when necessary.

- Player: This class encapsulates the state and behaviour of the player. For the moment, the state comprises only the number of coins in the player's possession. Note that there will only ever be one instance of the Player class in the program, which we will refer to as the player object.

- VampireList, SlayerList: Each of these classes contains (i.e. has an attribute whose value is) an array of the respective game elements, together with methods for managing this array. In this assignment you must use standard Java arrays, rather than a collection class from the standard library such as ArrayList.

- Board: This class encapsulates the state and behaviour of the board. It contains (a reference to) an instance of the VampireList class and (a reference to) an instance of the SlayerList class together with the methods for managing the access to the objects of these lists. The Game class delegates much of its functionality to the Board class. Note that there will only ever be one instance of the Board class in the program, which we will refer to as the board object.

- Game: This class encapsulates the logic of the game and is responsible for updating the state of all the game elements. It maintains the current cycle number. It contains (a reference to) the board object, to which the game object delegates much of its functionality, and (a reference to) the player object.
- Level: This is an Enum class that contains the values which correspond to each level of the game. Any level-dependent code of the program (i.e. a `switch` or `if-else` in which possible level values figure explicitly) should be in this class[4].
- GamePrinter: This class is passed the game object in its constructor and has a toString method that produces a string which, when when sent to the standard output, displays the board on the console.
- Controller: This class controls the execution of the game, printing the prompt on the console and then reading the command provided by the user, parsing it and updating the game accordingly. This class needs, at least, the following two attributes:

  ```
  private Game game;
  private Scanner in;
  ```

  The object contained in the in attribute is used to read the commands on the standard input, i.e. entered by the user via the keyboard. This class has a method public void run() which contains the main loop of the program in which, while the game is not finished, the state of the game is printed on the console, the user is prompted for a command and the command is executed. Note that there will only ever be one instance of the Controller class in the program, which we will refer to as the controller object.
- Buffy: This class contains the main method of the application. The JVM passes the values of the command-line parameters — in this program, the level and, optionally, the seed — to the main method, which creates the game object (instance of the Game class), the controller object (instance of the Controller class), passing the game object to its constructor, and finally invokes the run method of the controller object.

Observations regarding the implementation

- We also provide you with some code to help you get started.

The rest of the information needed to implement the assignment will be provided by the lecturer during the lectures and lab classes. The lecturer will give indications of which aspects of the implementation are considered obligatory in order to accept the assignment as correct and which aspects are left to the students' judgement.

Note also that in a *Problem-Based Learning* approach, the student is required to search for the knowledge they need to solve the problem at hand and to apply this knowledge to solving the problem *before* the pertinent information and solutions is presented in

[4]the reason being that if we decided we wanted to add a new level or remove an existing level, we would only need to change the code of the Level class itself in order to do so

lectures. Many studies have shown that knowledge is more easily absorbed and retained if it is acquired in this way, by the student working under the lecturer's guidance but independently. Moreover, perhaps the most important ability to be acquired at university is how to learn independently.

7. Submission

The assignment must be submitted as a single compressed (with `zip`) archive via the Campus Virtual submission mechanism not later than the date and time indicated at the start of this document. The zip archive should contain at least the following[5]:

- A directory called `src` containing the Java source code of your solution,
- a file called `alumnos.txt`, containing the names of the members of your group,

Do **not** include the files generated on compilation (i.e. the .class files which, if you created your project following the procedure of Assignment 0, will be in the `bin` directory). Optionally, you can also include:

- a directory called `doc` containing the API documentation in HTML format generated automatically from the Java source code of your solution using the javadoc tool.

The contents of this directory will not contribute to the grade in this assignment but may do so in subsequent assignments. Note that using javadoc involves adding comments to your code in the javadoc format, otherwise the HTML that you generate will contain almost no information.

8. Testing

In this assignment you will be given a folder containing executions, each comprising an input file and a output file using the following nomenclature:

- *easy_s5_0.txt*: contains the inputs of test case 0 with level *easy* y seed value 5.
- *easy_s5_0_output.txt*: contains the expected output of test case 0.

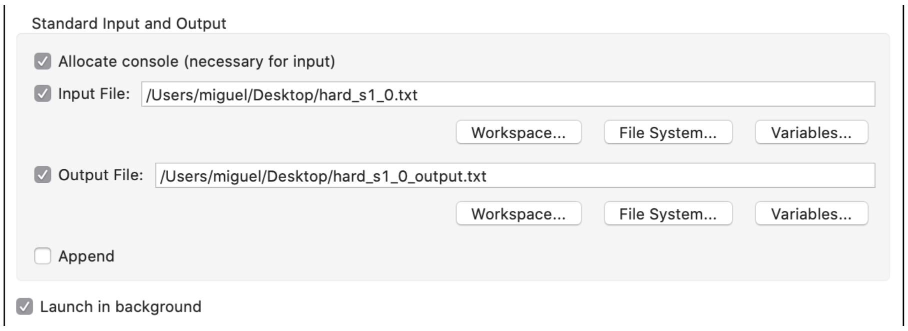
Redirecting the standard input / output to a file intead of using the keyboard for input and the screen for output is configured in Eclipse in the *Common* tab of the *Run Configurations* window as shown in Figure 2. Note that if your solution does not use the methods of the Random class in exactly the same way as in the code used to produce the test case outputs, you will not obtain the same output. For this reason, you must use the following methods in your solution:

- nextDouble() to decide whether or not to add a vampire,
- nextInt() to decide in which position on the board to add a vampire,
- nextFloat() to decide whether in a given cycle the player receives some coins or not.

For your interest, I would normally recommend using nextInt() in all cases but it is not of great importance[6].

[5]You may also include the project information files generated by Eclipse

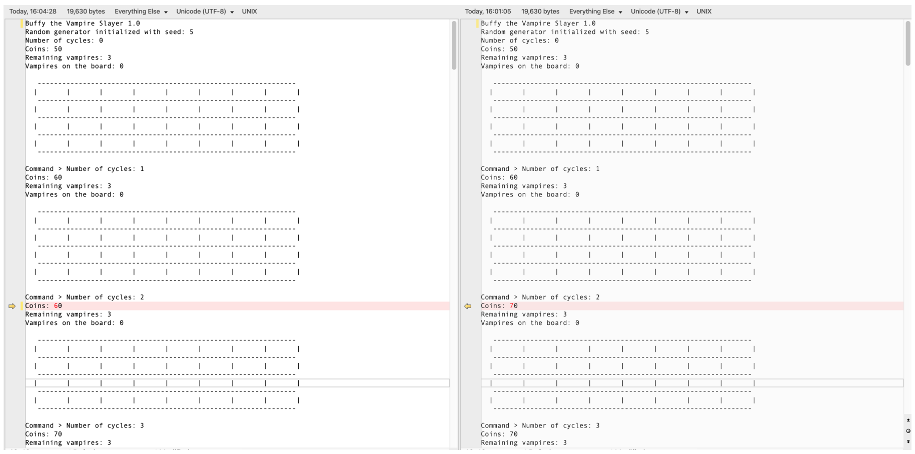
[6]What is of importance is that you must **not** use Math.random() (which actually uses the Random class method nextDouble() internally) since such use does not permit explicit manipulation of the seed.



Standard Input and Output

- [✓] Allocate console (necessary for input)
- [✓] Input File: /Users/miguel/Desktop/hard_s1_0.txt
- [✓] Output File: /Users/miguel/Desktop/hard_s1_0_output.txt
- [ ] Append
- [✓] Launch in background

Figure 2: Redirecting the standard input and output.



Today, 16:04:28 19,630 bytes Everything Else Unicode (UTF-8) UNIX

```
Buffy the Vampire Slayer 1.0
Random generator initialized with seed: 5
Number of cycles: 0
Coins: 50
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 1
Coins: 60
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 2
Coins: 60
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 3
Coins: 70
Remaining vampires: 3
```

Today, 16:01:05 19,630 bytes Everything Else Unicode (UTF-8) UNIX

```
Buffy the Vampire Slayer 1.0
Random generator initialized with seed: 5
Number of cycles: 0
Coins: 50
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 1
Coins: 60
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 2
Coins: 70
Remaining vampires: 3
Vampires on the board: 0

   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------
  |      |      |      |      |      |      |      |      |
   -------------------------------------------------------------

Command > Number of cycles: 3
Coins: 70
Remaining vampires: 3
```

Figure 3: Comparing the output obtained with the expected output.

There are many free programs to visually compare files and thereby check that the output of your program coincides with the expected output for each of the test cases we provide. Figure 3 shows a screenshot of the *Beyond compare* program[7] which is available free for all platforms. Take care with the order of instructions in your program since it can have a significant effect on the output. If you detect an error in the output of any of the test cases provided please let the lecturer know ASAP so that we can correct it.

Note that the testing of your solution should not consist exclusively of passing the test cases we provide; you must also check other executions not present in these test cases. During the lab correction of the assignment, we will use different test cases to check that your solution doesn't only work correctly with the test cases provided to you with the problem statement.

[7]See `https://www.scootersoftware.com/`. Another possibility is DiffMerge (`https://sourcegear.com/diffmerge/`)