Grado en ciencia e ingeniería de datos

Programming - Academic year: 2020/21

Approach to Object-Oriented programming

DARÍO CABEZAS LARESE

LUCÍA CORDERO SÁNCHEZ

Instructors: ***Yolanda Escudero Martín***

José María Uñón Prieto

Date: 21st December, 2020

Contents

1 Aim of the project

This draft is created with the goal of emulating successive battles between two existing families in Game of Thrones, the Targaryens, whose leader is Daenerys Targaryen, and some other family in Westeros.

While the Targaryens will always be attacking in a random location (or the one chosen by the user), the other family in Westeros will be defending themselves from the attack in two ways: the user will introduce the terms in which they will defend (related to size of the batallions, leader, etc.) or in an automatic way, whose game parameters are given randomly.

2 General explanation of important classes

We created **six** different classes to create our game dynamics. In all of them we incorporated encapsulation (a basics for Object-Oriented Programming).

Character: this class is used for the creation of all characters of the game, including the ”special” ones (dragons, Queen, Undead King and generals) that will be stored in a different way.
Characters have their category in the game (archer, human soldier, etc.), name (if so), fixed bonus (in which conditions their strength increases and the points it increases) and variable bonus depending on the attack or defense they deal with (specially when coming to some specific characters with skills). Finally we added their strength.

The class have its getters and setters proved the private nature of these attributes, we used them to create all batallions and with them, the armies.

Map: the map was configured in the shape of a dictionary with numerical keys to represent the different givem locations. This way we found it easier to work with them rather than with the names.

Naturally we wanted our map to be a ”storage” for batallions, so we added important methods such as addBatallion (attach a batallion in a given location) or delBatallion (delete a batallion from the place).

Batallion is a class to contain the batallions' lists by first using the method addCharacter to append the different characters to the batallion list.
To give a clear example of its use, we create Targaryen army adding all characters with this method and the special characters such as dragons were appended to their respective lists thanks to it.
In order to extract the length of each batallion and to read the characters on it we created their own getters and setters if necessary.

Army was fundamental to declare both Targaryen and Westeros armies throughout its methods: add batallions, remove them (in this case, once they are already dead in the fight) and getters to read all inside the list of the army and to get the number of batallions in the army.

For the armies setup, we imported Character, Batallion and Army. Targaryen soldiers were added; subsequently, the archers and finally the characters with soecial skills: queen and dragons.

Game parameters were very useful to set the conditions of game. In here we contemplate all the possible outcomes of the battle considering parameters such as the leader of batallion (is it the queen or not?), the number of batallions a usar wants to play with and the order they should follow (increasing or decreasing), if we are in the full attack or partial attack mode and in which location the fight will be.

Battle summarizes all the above; it is the class with the most important functions, it is the ”engine” of the game.

There are many methods that allow us to fulfill the task of making two batallions fight (with conditions), such as ”fight” itself, ”show” to represent the results of that round or the whole game of rounds, and also we set here the conditions of final strength that each character will have at the end of the battle.

3 Significant algorithms

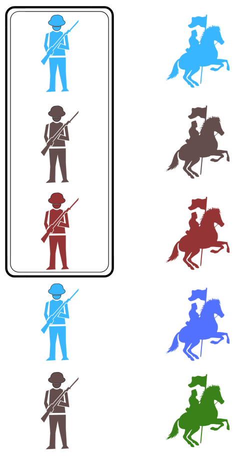
3.1 Algorithms for the menu

- *assignQueen* gives two possible outcomes: yes (”y”) or no (”n”).
- *chooseBatallion* assigns the number of attackers with whom the user want to play.
- *batallionOrder* determines if the batallion(s) should fight from the strongest to the wekaest or the other way around.
- *chooseAttackMode* is used to play in partial attack (only one turn) or full attack (until one army is fully dead).
- *chooseLocation* chooses one location from the map and determines which are the batallions who will fight the batallion previously chosen by the user (from all that may be in that location).
- *menuArmySetup* is the second part of the menu, shown whenever the user wants to set the army and corresponding settings (after the ”first screen”).
- *menuMain* is the first screen with two outcomes: play or go to the second screen (srtting for the army).

3.2 Algorithm for the fight

- *matchLengths* This algorithm fixes a huge problem of discordancy; if the attacker and the defendant batallions are not of the same size and we choose partial attack, all our attacking soldiers should fight at least once against the defendants; with this algorithm we avoid non parallel attacks.

  It is easier to see graphically; if one batallion is shorter than the other one, as the list will not be a copy but a reference (we do not use the .copy), the first three attacking soldiers (marked in black) will fight against the corresponding defenders (in their same color). Supossing they are not dead at the end of all the three attacks, their strength will be updated



in each attack. As we are creating a reference for the fourth and fifth turns, the first and the second soldier will fight against the forth and fifth defendant soldiers considering they were damaged before until they are dead. This way, we make sure only parallel attacks are allowed and all soldiers fight.

- *fightBat, figth1bat, fightSoldiers.* All of them return the result of the fighting between one batallion and each other, many batallions and only the fighting between soldiers. They are obviously correlated, and their mechanism is the same. In the soldiers' fighting we also cover the special attacks that give extra bonus to the characters.

4 Game approach

As in the explanation of this project there were some cases in which we had freedom to choose whether we wanted our game to run one way or another, there are some remarkable aspects we would like to appoint that affects the game dynamics, as the ones which follow:

First, when fighting we took into account not only the specified cases (when one character is stronger than the other one), but also when they are exactly equal. In this case, we concluded that the best choice would be decresing the strength of each soldier in 1/3 the strength of the other one. This way, we made sure there is an update in the strength since they have fought and they have both been damaged by each other.

Related to the automatic game sprint: instead of working with random values for the variables, which in many cases may cause the game to have issues since those values are not interpreted by the computer before being taken, we proposed another way to automatize turns.

First time the user runs the menu, the only existing possibility is to build the settings of the army; not to run the battle directly.
In the second turn, both options are available, and also the possibility of going full automatically.
This mode takes the last values introduced by the user and plays with them, this way we will make sure they are coherent with the mechanism of the game but the user will not be directly involved in the automatic turns.

5 Implementations for Users-Experience (UX)

As an extra, first we developed a way to test if our game worked from the inside (as developers) throughout the ”True” and ”False” conditions, but it was implemented hindsight as our automatic mode.

Moreover, in the menu we created ”jumps” to step into one place or another depending on the willingness of the user; for example, when initializing the program, the user will have to fulfill the option ”fullfill everything” (which in the standard menu for the rest of the game will be the sixth option, all of the above”).

This way, the person who plays will save time choosing between running a battle or setting only some parameters if that is not a possibility of the game from the very beginning in order to work well.

6 Final comments

As a personal conclusion, the project's ”core” is to see the game as a whole, not only as different parts together.

For us it was very useful to take pieces of advice or read websites such as GitHub and StackOverFlow, and we would consider specially interesting to dive more deeply in class in some common mistakes we all got when trying Objects-Oriented Programming.
We consider that knowing the basis of programmming is as important as being able to find a proper and solvent solution to the daily mistakes we may have.

To sum up, we have hone our programming skills and we consider we fulfilled quite successfully the challenge that was given to us, prioritizing **clean and understandable** code, and **avoiding repetitions** or ”spaguetti code”.

Despite all these reasons, we are sure we will be able to find better ways of programming and implementing all we have learnt in the close future, these are only the first steps in this way of a lifetime that are new technologies, all the time evolving and getting more efficient.