Basic Programming Java Project Report “Battleship Game”

Rajat Sharma, MAI (ECS)
r0846712
KU Leuven

Game Description

This project was accomplished as part of the course Basic Programming at KU Leuven where the main objective is to gain knowledge and acquire coding skills in Object Oriented Programming (OOP). For this, one of the most widely used and popular Java programming language was employed.

The main objective of this project was to make a famous game, namely, Battleship, using Java. The game consists of two players taking on each other over a squared grid of tiles where some of the tiles hide specific ships behind them. Each type of ship has some characteristics associated with them such as colour, length (in terms of number of tiles) and the points one earns on hitting it. These details about the features of the ship are unknown to the players to make it more fun and realistic. Players take turns to randomly click (*hit*) the tiles and depending on whether that particular tile hid a ship or not, they receive points - if there was a ship, corresponding points for that ship are awarded else no points are given. There is a special case when the last tile of a specific ship is hit and in this instance, the player is awarded double the points for that ship. The game continues until all the ships have sunk (*uncovered/hit*) and in the end, the player with most number of points wins the battleship game.

On understanding the requirements of the game and how it should work, a design philosophy was drawn and then based on that a working implementation was written in Java. This report talks about the working behind the development of the game in the next few pages. At my end, for the Java programming part, I chose Eclipse IDE as I found to be a good balance of complexity as well as simplicity. The GUI of Eclipse was also something I felt myself comfortable to work with and hence chose it as my primary programming environment. Java SE 15 (JDK-15.0.1 being the latest version) from Oracle was downloaded and used in Eclipse. Furthermore, just as a piece of info, this was my first time using and working with Java. I have had prior experience only with Python, some C and mostly Matlab.

Design Philosophy and Implementation

A careful study of the design and working of the game results in a very clear picture of how the game could/should be coded in Java, not to mention that there are multiple methods-of-approach (MOA) to achieve the same final state. Which approach is followed is a function of many factors, including one's experience with Java or rather OOP and one's computational thinking.

My approach to have a fully working game was as follows. As a first step, decomposing the problem in several small pieces (here *pieces* = *classes*), to be handled individually and later recombined/connect, was done. To be precise, overall 7 classes were made and then linked together in such a manner that invocation of one class triggered the workflow of the whole game/program. What this workflow is, has been described below after a short descriptive list of all the classes (*.java files)* of the Java program.

1. **MainRunGame** - Starts the game by initiating through an object of *StartButton*
2. **GUIMenu** - Comprises of all the GUI components required for forming the panels
3. **StartPanel** - Forms the front starting panel (screen) for the game
4. **StartEvents** - Makes the components on the starting panel active
5. **StartButton** - Main button on the starting panel which starts the game
6. **GameBoard** - Few options, grid of tiles with hidden ships where the game is played
7. **GameEvents** - Events which make the game board active and the game playable

The understanding behind the workflow, comprising of the above mentioned classes, comprises of a clear cascade of events that actually takes place when the **MainRunGame** class is run. The game starts with an initial screen, with all the settings (*value*s) required from the user's end, that pops up when the game application is run. This is the starting point of the game from where the rest of the actions follow ultimately leading up to the final game board completely set up with the ships hidden. So, essentially, there are two main stages/ parts of the game/game development. First is this starting screen (= *starting panel*) and the second one is the main game board (= *game panel*) for the players to play. Each panel comes with its own set of events (= *methods*) that make these panel *come to life* by making them active (adding different types of *actionListeners*).

The starting panel has three main options, namely - dimensions (rows and columns in the grid), scoring system option and ship placement in the grid option along with other options (*buttons*) such as *rules*, *high scores* and *exit*. The last three options explain themselves and to have even a clearer understanding about the game and its features, the user is advised to refer to the *rules*. The first three settings need to be known in order to be able to set up the game panel. Scoring system has two options - unique points-per-hit and same points-per-hit -which are quite self-explanatory. The dimensions have to be >= 5 to be able to accommodate the 4 ships with the largest of them being 5 tiles long and with, of course, the rows and columns being equal to each as it is a square grid of tiles. For the ship placement, there were two options, which were not so straightforward to code, at least for an amateur Java programmer. First one involves randomly choosing a set of coordinates and placing the ships there accordingly ensuring no overlap between them, the total number of ships equals 4 and the the total number of coordinates are 14 (equal to the sum of lengths of all the ships). The coordinates are generated on a general basis and thus would work with any dimensions. For the case, when the dimensions are >=5 and <= 7, all 4 ships are either chosen to be placed horizontally or vertically whereas in the case when dimensions >=8, there could be a mix-up with some ships being placed horizontally and some vertically. This differentiation is logical and is evident from the fact that it can be quite a challenging task for *small grids (<=7 and >=5)* to have ships placed in both orientations without having an overlap between them. Thus the two-different MOA for random placement. The second method is file based (requires filename) and once selected disables the first option and other related options as the file contains all the info of the coordinates for the ship to be placed as well as the dimensions. Error checks have been programmed in to tackle incorrect/missing data as the file is parsed notifying the user when the data read is inadequate, incorrect or missing. Parsing the file is again a little complex as splitting and then extracting the relevant info proved to be a bit difficult. But nonetheless, both the methods for ship placement were coded successfully.

As mentioned previously, to prevent incorrect settings that would hinder the formation of the game panel, appropriate error checks had to be coded in for the user. Considering that there are three such options with each of them being user-defined, it is easy to see that it is not really a wise choice to cross-check each error and have a pop-up for the user for that specific error. Thus, a single error check with a helpful message, telling the user to check the missing data as mentioned in the message, is thrown in case of an incorrect or missing value. Once all the entered values and selected options by the user are deemed correct and complete, the user can go ahead and start the game using the *Start Button*. Thereafter the game board appears with the hidden ships and users can take turns hitting the tiles and are awarded accordingly. At any time and towards the end too, the winner's score is appended to the end of a file named *High Scores* which stores this info for future reference. For viewing the contents of this file, there is a button on the game board called *High Scores* which can be clicked anytime and the leaderboard (winner and the winner's score) are displayed in the console.

This is the main logic behind the game development. To summarise, the game is started which instantiates an object of the *StartButton* class and then calls its method, *initiatePlay*, subsequently the start panel is formed (using the necessary components from the *GUI Menu*) through instantiation which further instantiates an object of the the *StartEvents* class. After all the error checks have been passed, the game board is formed using the *GameBoard* class and its associated game events (methods) through the *GameEvents* are enacted. The game then starts and continues till all the ships have sunk and thereafter the winner is announced and the score is appended to the *HighScores* file, whose contents are accessed through the *High Scores* button. The game can be closed using the *Quit* button. As pointed out, it is a good practice for the user to go through the rules where everything is written.

Major Hurdles Faced and Further Improvement

As they say, everything seems easy unless it is attempted in reality. The same can be said for this project. From my end, the major issues faced were - getting the multiple components/*panels* in a proper layout in the *JFrame*, adding *actionListeners* for the grid, understanding the logic behind how to update and get selected values for *radioButtons*, randomly placing the ships, parsing the file for values, error checks popups and their logic. But with patience and time, I was able to manage and successfully resolve all these issues, on my own, in the process greatly enhancing my overall understanding of OOP and Java.

There is always some scope of improvement and it's true here as well. Some good points to further improve this project would be as follows: aesthetics and the visual appeal of the game can definitely be improved. I did attempt to do this by trying to incorporate icons for the 4 ships in the grid instead of colours, however the icon turned out to be very small and scaling it to the size of the grid did not work out. Also, the game could be scaled further in terms of the number of ships with more diverse features which would bring it closer to the battleship game we have today. More options for the user could be provided, maybe, for example to choose the ships he/she wants. One interesting feature would be to replicate weapon systems for various ships which depend on the chosen ships and which extend to multiple tiles rather than just one simple tile. Scoring system could also be revamped accordingly. And of course, sounds can be added to make it more interactive. For me, adding more graphics (visual design) would probably be the first priority and then extending the game with all these features would come next. Again, there is no limit to it as literally so many things could be done to further improve the design of the game as well as the game itself. Even a different implementation from the point of view of programming could be done. All in all, it depends on the time available and in this case sky is the limit.

Conclusion

Considering the time constraints, resources, prior experience and everything, a best shot was definitely given from my end. Overall, more than 30 days were dedicated to it, to pickup Java as fast as possible and then learn Swing. The end result was a properly functioning battleship game, pretty decent for an amateur (*hopefully*). Also, I would like to point out that this project consisted completely of my own efforts and no external support or help was availed. A lot was learnt, the time spent was fully enjoyed and to conclude, it was a great learning experience overall. I hope to further develop myself by advancing my Java skills and hope to become an Oracle Certified Java Developer, for which preparations have already begun. Thank you for this opportunity Professor.