**Text

Description automatically generated with medium confidence**

**Coursework Software Development 1**

**SET11102**

**Matriculation Number: 40494259**

**DECLARATION**

I agree to work within Edinburgh Napier University’s Academic Conduct Regulations which require that any work that I submit is entirely my own. The regulations require me to use appropriate citations and references in order to acknowledge where I have used any materials from any sources.

I am providing my student Matriculation Number (above) - in place of a signed declaration – in order to comply with Edinburgh Napier University’s assessment procedures.

**Table of Contents**

[1 Introduction 3](#_Toc120614118)

[2 Discussion 4](#_Toc120614119)

[2.1 The design and Class diagram 4](#_Toc120614120)

[2.2 Representation of the grid 5](#_Toc120614121)

[2.3 Placing the ships according to the game configurations 5](#_Toc120614122)

[2.4 Playing the game 6](#_Toc120614123)

[2.5 Advanced features 7](#_Toc120614124)

[2.6 Testing 7](#_Toc120614125)

[3 Summary 8](#_Toc120614126)

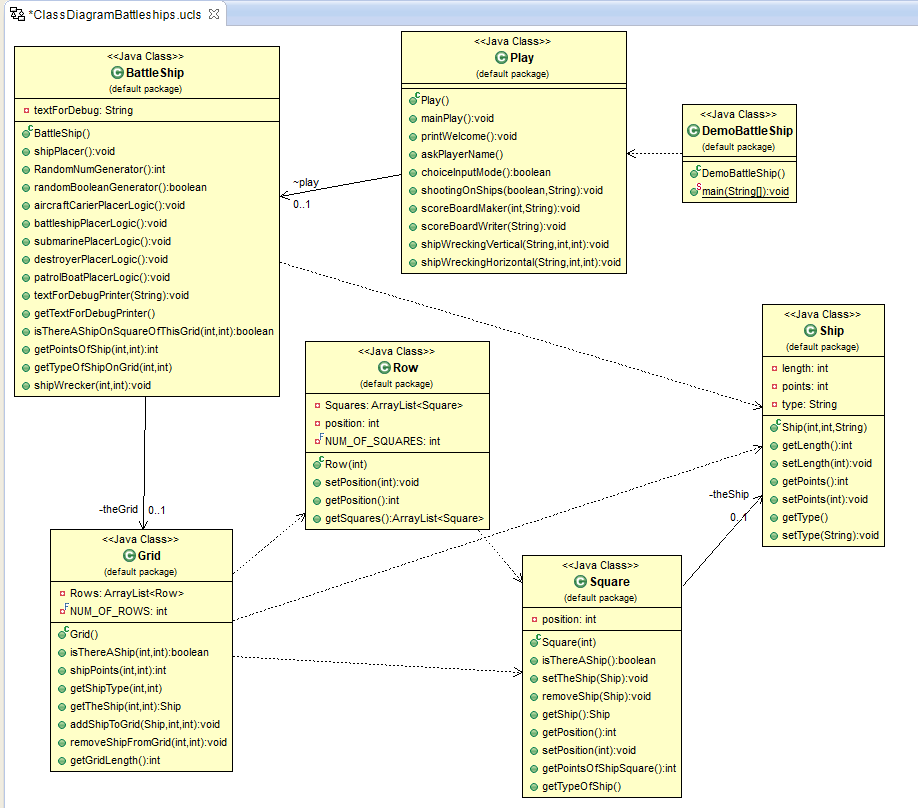
# 1 **Introduction**

The current document was created to support the Battleship java project for the Coursework assignment of the Software Development 1 module. There are five points discussed. First, the overall design and the structure of the project are described, and a class diagram is provided for better visualisation of the core elements. Second, it is discussed how the grid was represented in the project. Third, it is described how the ship was placed on the grid fully compliant with the game configuration. The fourth part of the document shows the overall logic of playing the game and the main interaction between the player and the program. Finally, the advanced features and the testing are outlined.

# 2 Discussion

## 2.1 The design and Class diagram

**Figure 1 UML Class diagram**

The overall design and structure are visualised in *Figure 1* using the Class diagram.

The game was designed using seven classes, from which DemoBattleship is the main class, from where the program can be run.

The Ship class contains the information and methods related to the ship. Similarly, to the Square, the Row, and the Grid classes.

Battleship class contains the Grid where ships are placed, with all the methods of placing the ships and interacting with the grid. The class has five methods to place each ship and several other methods to interact with the Grid created.

Finally, the Play class contains all the methods of playing the game and facilitates the interaction between the player and the game. There is one separate method for shooting on the ships and two methods to delete the ship from the grid if it was already hit.

## 2.2 Representation of the grid

The Battleship game was designed using the object-oriented approach. The grid is an Arraylist of 10 rows and rows are an Arraylist of 10 squares. Ships were also represented as objects. Each ship has a length, points, and type as attributes. Each square object has an object Ship as an attribute. This attribute can be either null or filled with a ship, hence, each square on the grid has the information on whether it contains a ship or not.

## 2.3 Placing the ships according to the game configurations

According to the configurations of the game, ships with different lengths needed to be placed on the grid randomly, without overlapping with the borders and with each other. Furthermore, they could be assigned to the grid either vertically or horizontally.

Ships are placed using separate methods because they all have different lengths. The points of the rows are the points of the y-axis, and the points of the squares are the points of the x-axis. There are four possible ways any ship can be aligned from the starting point: vertically going up/down, horizontally going left/right. Hence, the main approach was to first find the start point of the ship and from there either increment or decrement the row or the square according to the length of the ship and the orientation of the way it is aligned.

**Placing horizontally or vertically:**

First, each ship placing method uses a Random Boolean generator from the java.util package to determine whether the ship is placed horizontally or vertically. Once, this is determined using the same Random Boolean generator it is decided which way the ship is aligned from the starting point. The starting point is then determined using java.util package Random number generator.

If the ship is vertically aligned, the next points of rows (y-axis) can be either going up or down. If going up, the next points where the ship can be placed are determined by decrementing from the start row. Contrary, if the ship is going down, the next point is determined by incrementing the start row.

Similarly, if the ship is horizontally aligned, the next point of the square (x-axis) can either go left or right. The next point is either incremented or decremented from the start square, depending on the orientation.

**Preventing ship overlapping:**

To prevent the ship is **overlapping with the** **borders**, a while loop is used to validate that the generated number for the starting point (square/row) is not lower or higher than the maximum or minimum point of the grid (max 10, min 0).

Each ship is placed using this method, except the patrol boat whose length is one number, hence, it cannot overlap with the border (the random number generator creates numbers from 1-10).

To prevent the ship is **overlapping with other ships,** a while loop is used to validate that the generated starting point does not contain a ship. The Grid class has an isThereAShip() method which needs a row and a square. It determines whether the square contains a ship by looping through the rows to the square, where a method can be called from the Square class to evaluate if the given square’s Ship attribute is null or not.

Each ship is placed using this method, except the Aircraft Career ship, as that ship is placed as the first (hence the grid cannot contain a ship).

## 2.4 Playing the game

**Input validation:**

Using a try-and-catch method and a while loop with minimum and maximum numbers, it is prevented the player inputs anything else than a number between 1-10.

**Hitting the ship while playing:**

In the Play class and the shootingOnShips () method, once the player provides a guess to shoot, it is evaluated with the isThereAShip() method if the player hit the target. Two other methods are designed to evaluate which ship was hit and how many points it is worth.

**Sinking the ship hit:**

Another configuration is that once a ship is hit, it sinks and cannot be hit again. To prevent the players from hitting the same ship again on a different square and row two methods were designed in the Play class. These methods take the square and the row from the successful hit and the name of the ship. They then evaluate if at any coordinate possible there is another part of the ship and destroy that part on the grid.

**Ending the game**

According to the rules, the game must end after the tenth shot or after the last ship was sunk. To make sure there are only 10 iterations to ask for a shot a for-loop method was used. All the types of ships already hit (and sunk) are added to an Arraylist of strings. Once the ArrayList reaches size 5 (meaning the ships are all sunk as there are 5 ships), the main for-loop breaks and the game ends.

## 2.5 Advanced features

**High score board**

The plan for the high-score table was to create a TreeMap using java.util packages and use the points of the player as key and the player’s name as value, sorting it by the key descending. Furthermore, the approach was to use the BufferedWriter and FileWriter from the java.io package to write the scoreboard to a local file and then read it back with the BufferedReader and FileReader methods into a TreeMap and sort it again by descending keys. However, the plan was only half finished. The methods are in the Play class.

**Preventing shots in the same place**

The plan was to create an Integer array with two numbers using the row and the square guessed by the player. Then add that array to an ArrayList of Integer arrays. To evaluate whether the Arraylist contains the array of the guessed number (meaning the player is trying to shoot at the same spot) was planned using the java.lang method contains(). This method returns a Boolean true if the ArrayList contains the same object. The approach was to further loop the input if the contains() method returns true. However, the approach did not work properly, thus it is toggled as a comment in the program.

## 2.6 Testing

The program was tested numerous times using the debug modes. However, a small unsolved anomaly might occur when running the program. Approximately 5 out of 100 times the program suddenly stops and closes after choosing the debug mode or shooting on the first ship. Neither Eclipse IDE, Visual Studio Code nor IntelliJ IDEA shows any error message or any problem in the code itself. Due to the lack of error messages and time constraints, this issue was unsolved despite the time spent trying to solve it. However, it is advised to re-run the program and that way it should execute promptly.

# 3 Summary

The current document outlines the strategy and design to implement an object-oriented approach to create the Battleship game. The grid in the game is built using an Arraylist of the Row objects which are an Arraylist of the Square objects. Java.util package was used to maintain the randomness of ship placement using a logic which first determines the orientation of the ships and then validates whether the ship can be placed in that position. A similar approach is used to shoot on the ships and to delete ships which are already hit. The advanced features were planned and designed, but their execution was only halfway carried out. There is a piece of advice at the end for the anomaly when running the program, which was not unsolved due to the lack of time and error messages of different IDEs.

**END OF PAPER**