BATTLESHIP GAME

Report

Abstract

This report contains guidelines about playing the Battleship game implemented by an author of this document. Also it contains documented all steps followed during the development of the game. The implementation of the game was developed due to a home assignment from job interview at Softwerk company in Växjö. Enjoy!

Nedzad Hafizovic

nedzad.hafizovic10@gmail.com

Contents

*1 – look at Table 2 in Page 5

1 INTRODUCTION

This document explains the process of creation of Battleship game which was performed as a result of an author's job interview with Softwerk company from Växjö. The document holds all information about the game development process as well as the guideline for playing the game.

1.1 Historical background

Battleship is a board game invented in 1930s, during the World War II. Different companies published it as a 'paper and pencil' game. In 1967, Milton Bradley presented his version which was the first instance of a plastic board game of Battleship. Later, as technology advanced, many different version, such as video game, appeared on the market. [1]

1.2 Rules of the game

Battleship is played by two players at the time. Each player has his own board which is usually 10-by-10 sized. Also, each player has his own fleet of five ships which he positions on a board. Each ship in a fleet has its size. A fleet members are:

Name	Size
Carrier	5
Battleship	4
Cruiser	3
Submarine	3
Destroyer	2

Table 1: Player's fleet

When a player positions his ships, it is important that one point of a board can be occupied by one and only one ship. This means that one point of a board can hold only one ship. Battleship is a TBS (turn-based strategy) game, meaning that one player plays at the time. The game is played in a way that players attack each other's board, while not knowing opponent's ships' positions.

*1 – look at Table 2 in Page 5

A player has one move to make before his opponent makes his move. The procedure repeats until one player destroys all other player's ships. One ship is destroyed once all points of a board occupied by that ship are hit. After each attack, players must notify each other whether a previous shot was a hit or a miss.

2 DEVELOPMENT

During the development of my implementation of this game, the classical SDLC (System Development Life Cycle) was not followed. This section covers process of a development of the Battleship game, with respect to all rules stated above, and it contains my subjective perspective of a situation.

2.1 Investigation of a problem and analysis

When I was assigned the task to implement my version of Battleship game, I did not know about the game, that is, I have not heard of it. I downloaded "Fleet Battle – Sea Battle" game [2] from Google's Play Store and I played few (approx. 100) matches. This helped me to get the feeling of how the game works and what are the rules. Then, I started thinking about the technology that I would use for the implementation. After some time, I decided to use Java and to make an old-fashioned console game.

I analyzed ways possible to represent the game with multiple components. This means that I tried to figure out how to set all parts of the game separately and connect them together. All components of the game that I found important are represented with different classes. More about these components is to be said in subsection 2.2 of this document.

2.2 Design and implementation

The game was designed in manners of OOP (Object-oriented programming). As already mentioned, each component that I recognized as important was represented with separate class. The functionality of the game relies on the fact that all of these components are able to function and work together.

*1 – look at Table 2 in Page 5

As expected, if the game is played with ships, a ship is an important part of it. I created a class 'Ship.java' as a parent class. Each type of a ship is presented by different, subclass. The relation between superclass and its subclasses can be seen in Figure 1.

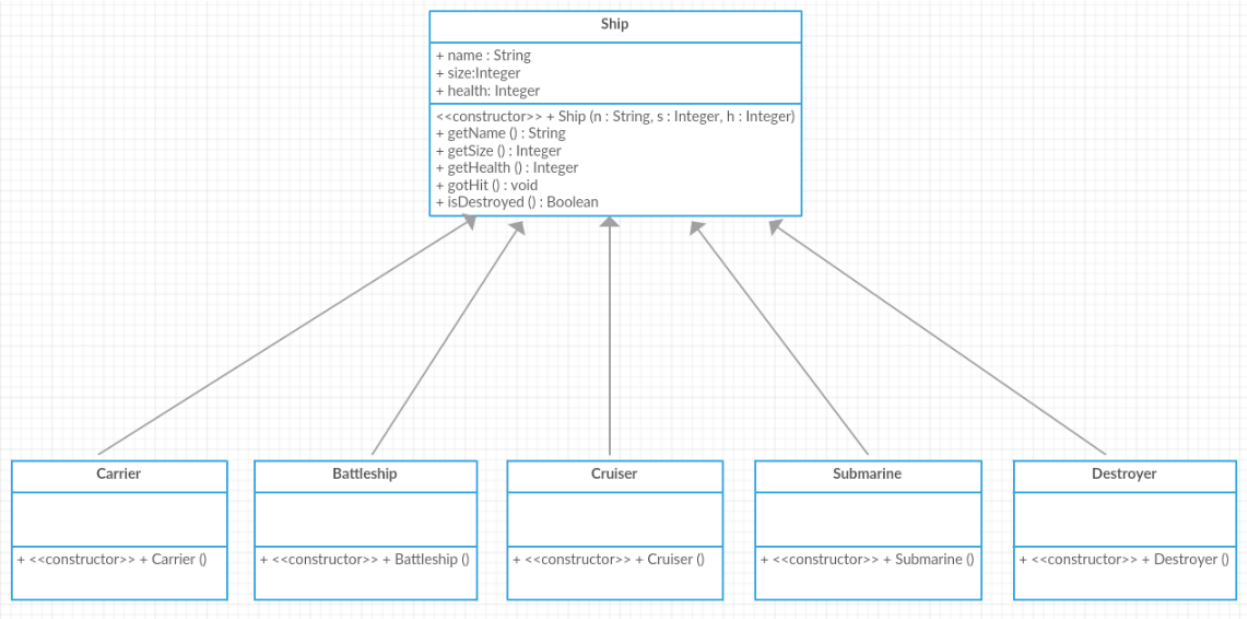

Figure 1: Class 'Ship.java' and its subclasses

Each subclass inherits methods and attributes from the superclass. Only thing that differs are attributes in a constructor, which is as well inherited from the superclass. Corresponding values are set to each type of a ship. When it comes to methods in a superclass 'Ship.java', they are mostly clear. I believe there is no need for explaining getter methods for each attribute. Method *gotHit() : void* decreases health of a ship if a ship gets hit. Finally, method *isDestroyed() : Boolean* checks the health of the ship and returns *true* if a ship's health is zero, or *false* is it is greater than zero.

Another important class is 'Board.java'. This class, as expected, represents a board on which a game is being played and on which players position their fleets. Board in this game is a two-dimensional 10-by-10 array which holds *char* values for each (x,y) position. Each state is defined by corresponding letter. Each player has two instances of this class. One instance shows a player's fleet and status of each position updates after each opponent's attack. Another instance represents a board which a player shots at. After checking positions on opponent's board, it

*1 – look at Table 2 in Page 5

updates depending on the success of the most recent attack. ‘Board.java’ can be seen in a Figure below.

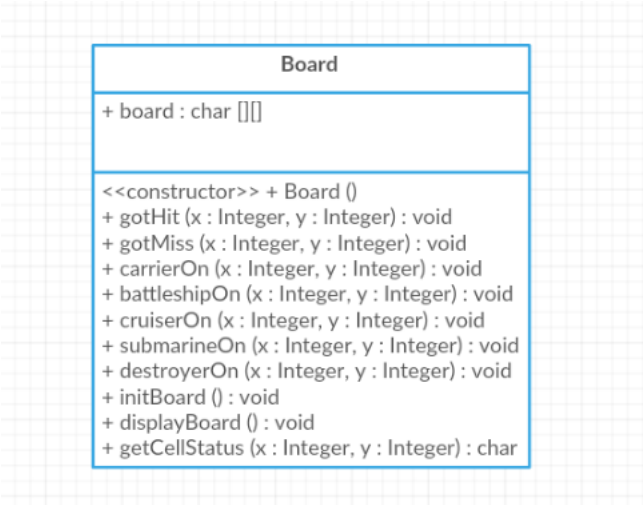

Figure 2: 'Board.java' class

As mentioned above, each (x,y) position in a board holds corresponding letter. All possible letters in one board position can be observed in the Table 1.

Character	Description
A	Carrier
B	Battleship
C	Cruiser
D	Submarine
E	Destroyer
o	Empty/Unexplored
x	Hit
~	Missed

Table 2: Possible states of (x,y) position in a board

In a constructor of ‘Board.java’, *board* is set to be 10-by-10 two-dimensional array. The method *gotHit(x:Integer, y:Integer) : void* sets position corresponding to x and y to ‘x’. The method *gotMiss(x:Integer, y:Integer) : void* does the same with ‘~’. However, methods *carrierOn(x:Integer, y:Integer) : void, battleshipOn(x:Integer, y:Integer) : void,*

*1 – look at Table 2 in Page 5

cruiserOn(x:Integer, y:Integer) : void, submarineOn(x:Integer, y:Integer) : void, and *destroyerOn(x:Integer, y:Integer) : void* set a corresponding (x,y) position to 'A', 'B', 'C', 'D', and 'E', respectively. On the other hand, the method *initBoard() : void* sets all positions within a board to 'o'. This method is used to initialize boards at the beginning of the game. The method *displayBoard() : void* prints the current state of a board. Finally, the method *getCellStatus(x:Integer, y:Integer) : char* returns a status of a position corresponding to x and y.

Since Battleship is a game, I created class 'Game.java' which is used to represents the elements and the flow of the game. This class handles the natural flow of one round of the game, from asking users to enter their names to deciding who the winner is. 'Game.java' can be seen in Figure 3.

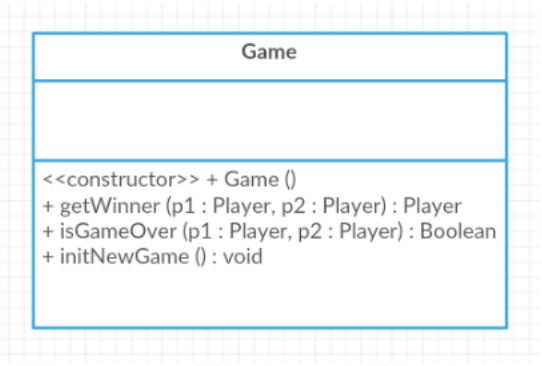

Figure 3: 'Game.java' class

'Game.java' class has three methods. Main and most complex method which handles most of the class' job is *initNewGame () : void* method. This method first declares two instances of a 'Player.java' class and the asks them to enter their names. Then, it initializes all boards that the game needs to be played, two boards for each player. This is done by calling method *initBoard() : void* from 'Board.java'. After this is done, the method asks users to position their boards using method from 'Player.java', which will be explained in more detail later in this document. Then, the method runs the entire game using methods from 'Player.java' class. In the end, it announces the winner of a game. The method *getWinner (p1:Player, p2:Player) : Player*, returns the player that wins the game. Finally, the method *isGameOver (p1:Player, p2:Player) : Boolean*, checks whether the game is over. If it is, the method returns *true*. Otherwise, it returns *false.*

*1 – look at Table 2 in Page 5

One more class that matters in this implementation of Battleship game is ‘Gameplay.java’. It holds only *main (args : []String) : void* method. Inside of the method, new instance of the game is declared. Then, a method *initNewGame () : void* is called on declared instance. Observing this class is possible from the Figure 4.

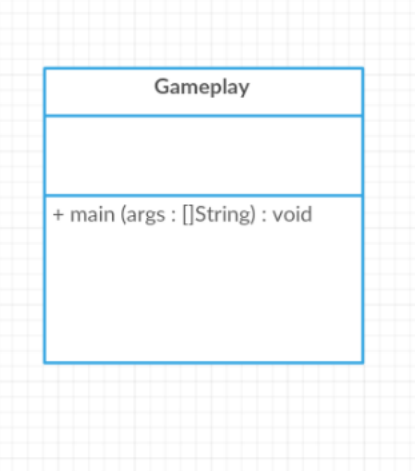

Figure 4: 'Gameplay.java' class

The most complex class of this implementation is ‘Player.java’. Reason for this is the fact that player does most of the things in this game. The representation of ‘Player.java’ class can be seen below.

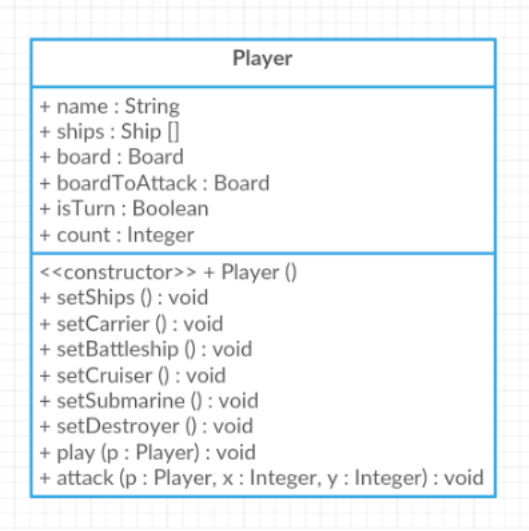

Figure 5: 'Player.java' class

*1 – look at Table 2 in Page 5

As shown in Figure 5, 'Player.java' class has multiple attributes and methods. Attribute *name : String* holds a name of a player. Next, *ships : Ship[]* is an array which holds one instance of each type of a ship. Attribute *board : Board* represents a board on which user positions his fleet. On the other hand, *boardToAttack : Board* represents a board which user attacks. The difference is that, when displayed, *boardToAttack : Board* shows only 'o' if position is not explored yet, 'x' if position is explored and some opponent's ship was hit, and '~' if position is explored and nothing was hit. On the other hand, *board : Board* shows player's boards in an order that a player set himself. Next, an attribute *isTurn : Boolean* is *true* if it is player's turn to play, or *false* if it is not. Finally, *count : Integer* represents the number of ships left in player's fleet. It is initially set to five, that is, the number of ships in player's fleet and the length of an array *ships : Ship []*.

When it comes to methods, let's explain the one by one. The method *setShips () : void* simply calls methods *setCarrier () : void, Battleship () : void, setCruiser () : void, setSubmarine () : void,* and *setDestroyer () : void.* Each of these methods does similar job. Each of them first ask a player to enter x and y coordinate for corresponding ship. For example, *setCarrier () : void* asks a player to enter x and y coordinate for his Carrier ship, *setCruiser () : void* for his Cruiser ship and so on. Each method checks if entered coordinates are allowed. If they are, the ship is positioned. Otherwise, a recursive call is performed and users is asked for the same action until the coordinates are valid. Positioning a ship to a board is done by replacing 'o' characters in a board with 'A', 'B', 'C', 'D', or 'E', depending on which ship is being positioned[*1].

The *play (Player : p) : void* method is responsible for a player to make his move. It first displays the current *boardToAttack : Board* to show player which positions he already attacked. Then it asks player to enter x and y coordinates for a new attack. If coordinates are valid, *attack (Player p, x:Integer, y:Integer) : void* is called. Otherwise, a recursive call is performed and a player is asked again until valid coordinates are entered. Then, the method performs an attack using a method that will be explained below. After an attack is performed, *boardToAttack : Board* is displayed again to show player the state after the most recent attack. Finally, player's *isTurn : Boolean* is set to *false*, while opponent's is set to *true*. Finally, the method *attack (p : Player, x : Integer, y : Integer) : void*, as mentioned earlier, handles attacks on an opponent's board. It uses *x : Integer* and *y : Integer* arguments to get the status of the position corresponding to x and y

*1 – look at Table 2 in Page 5

coordinates in an opponent's board. That status is stored in *status : char* variable. Then, 'switch – case' statement is used to perform different operations depending on the value of *status : char* variable. If a value of *status : char* is 'A', 'B', 'C', 'D', or 'E', the position in player's *boardToAttack : Board* and opponent's *board : Board* is marked with 'x', meaning that attack resulted in a hit. However, if a value of *status : char* is 'x' or '~', the message "Already attacked selected position! " is displayed. Finally , if a value of *status : char* is 'o', the position in player's *boardToAttack : Board* and opponent's *board : Board* is marked with '~', meaning that an attack resulted in a miss. It is important to mention that this method checks if any ship is destroyed after each attack in which that ship was hit. With respect to game rules, it also notifies the player of all possible outcomes.

2.2.1 UML diagrams

In the Figure 6 below, the Use Case diagram of this implementation can be seen. I add a player and a system as actors to this diagram.

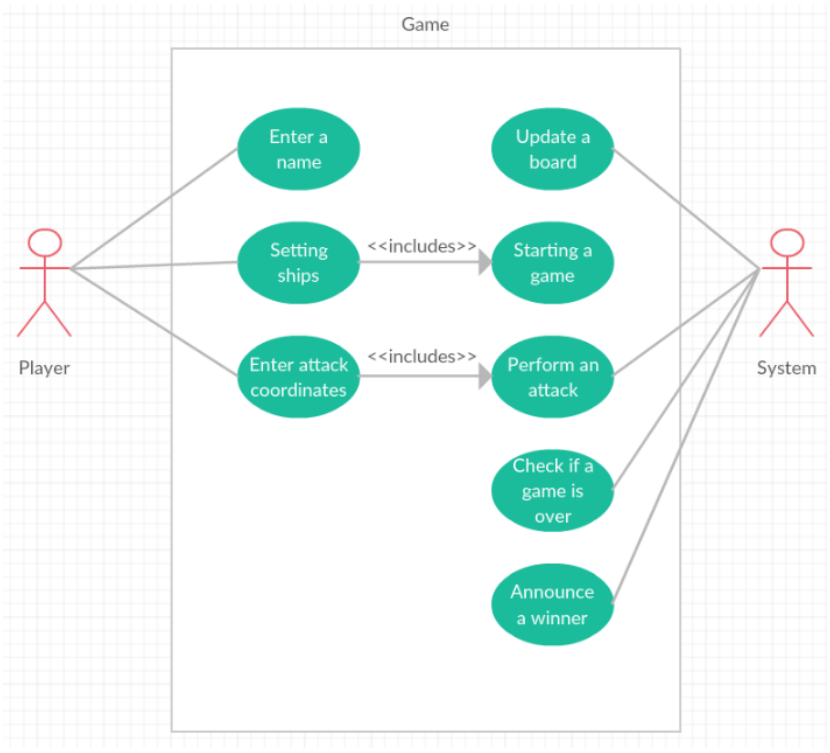

Figure 6: Use Case for Battleship

*1 – look at Table 2 in Page 5

The flow of the game is visually represented using Activity diagram which can be observed from Figure 7 below.

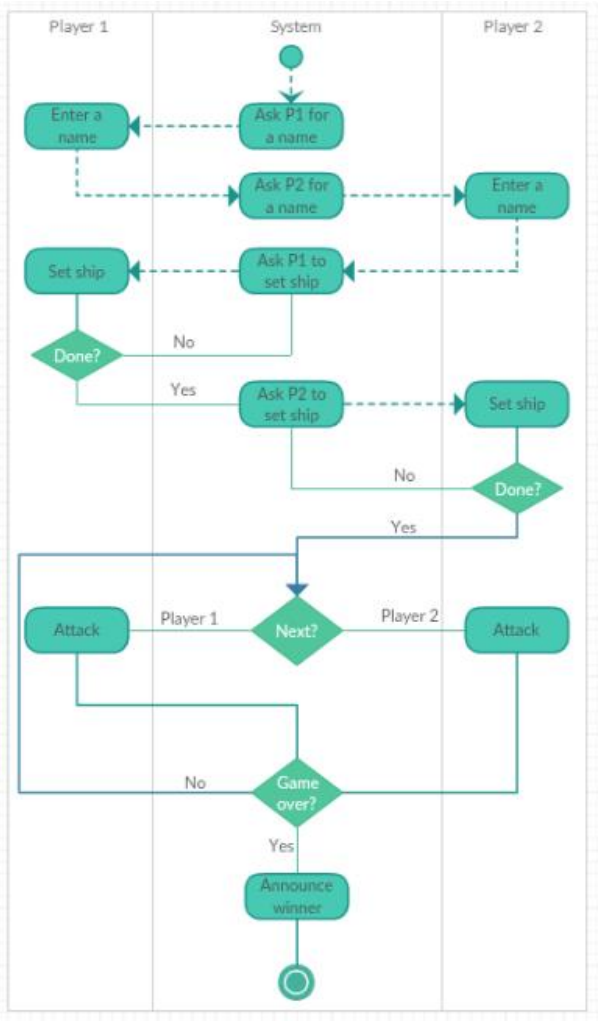

Figure 7: Activity diagram of Battleship

3 INSTRUCTIONS TO PLAY

Battleship is easy and very fun to play. It is played by asking players to enter their commands into a console. To start a game, one needs to run 'Gameplay.java' in an IDE such as Eclipse.

*1 – look at Table 2 in Page 5

Once the game is started, it shows menu which offers three options. First option is to start a new game, the second one is to read instructions, and the third one is for viewing credits. These options are chosen by entering corresponding number to a console. If the first option is chosen, players are asked to enter their names. After names of players are entered, the first player is asked to position his ships. Afterwards, the second player is asked to do the same thing. Ships are positioned through coordinates and orientation.

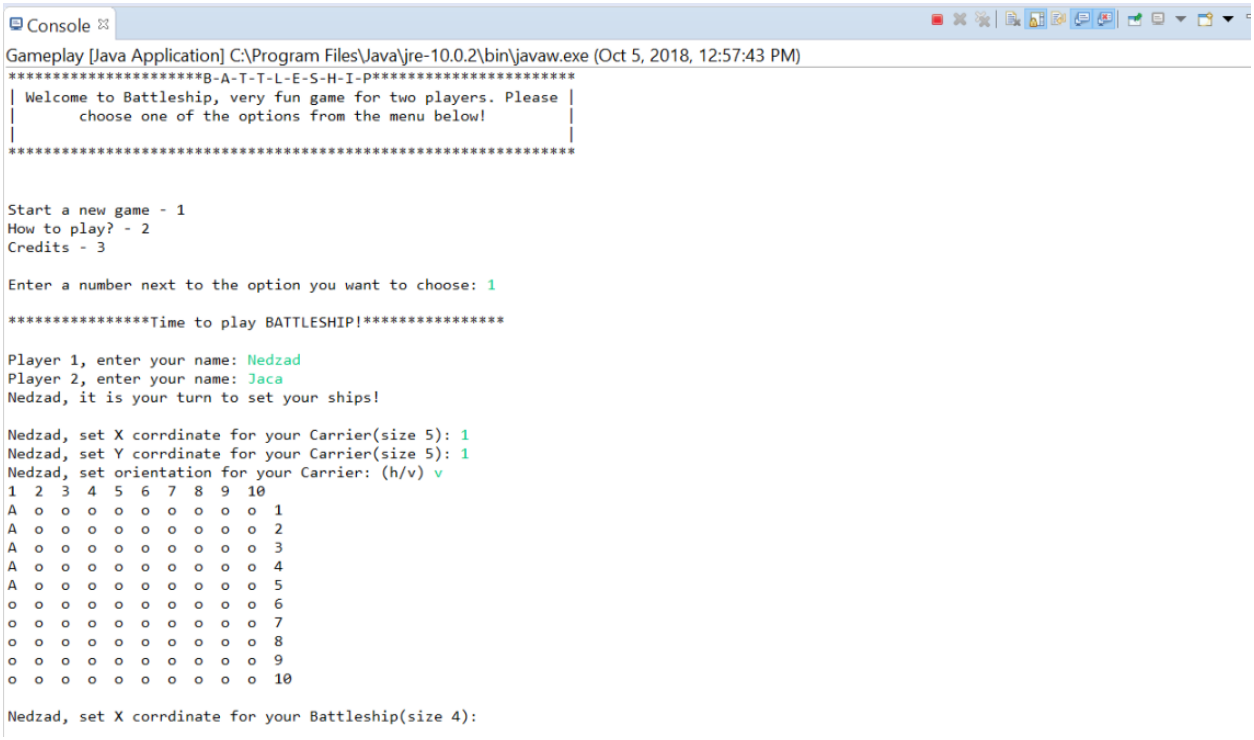

Figure 8: Starting Battleship game

After all the ships are set to their locations, the battle begins. Each player is required to press ENTER before an attack, so that the other player cannot see his fleet.

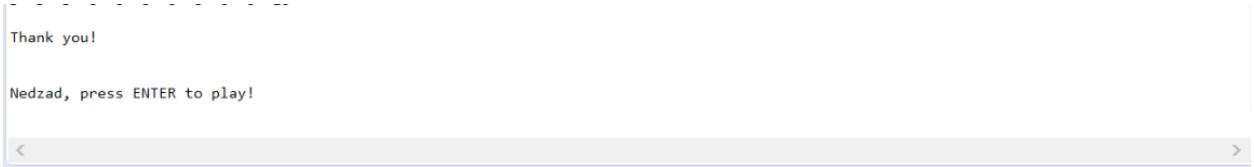

Figure 9: Player asked to press ENTER before attack

*1 – look at Table 2 in Page 5

After a player presses ENTER, he is asked to enter attacking coordinates and is shown his board and a board that he attacks.

```
Nedzad, press ENTER to play!

*********YOUR BOARD**********          ******BOARD YOU ATTACK******

1  2  3  4  5  6  7  8  9  10          1  2  3  4  5  6  7  8  9  10
A  o  o  o  o  o  o  o  o  o  1         o  o  o  o  o  o  o  o  o  1
A  o  o  B  B  B  B  o  o  C  2         o  o  o  o  o  o  o  o  o  2
A  o  o  o  o  o  o  o  o  C  3         o  o  o  o  o  o  o  o  o  3
A  o  o  o  o  E  o  o  o  C  4         o  o  o  o  o  o  o  o  o  4
A  o  o  o  o  E  o  o  o  o  5         o  o  o  o  o  o  o  o  o  5
o  o  o  o  o  o  o  o  o  o  6         o  o  o  o  o  o  o  o  o  6
o  o  o  o  o  o  o  o  o  o  7         o  o  o  o  o  o  o  o  o  7
o  o  o  o  o  o  o  o  o  o  8         o  o  o  o  o  o  o  o  o  8
o  D  D  D  o  o  o  o  o  o  9         o  o  o  o  o  o  o  o  o  9
o  o  o  o  o  o  o  o  o  o  10        o  o  o  o  o  o  o  o  o  10

Nedzad, choose first corrdinate to attack:
```

Figure 10: Player's turn to attack

After a players enters attacking coordinates, an attack is performed and a player is notified of a result of that attack. Then, the second player is asked to press ENTER to proceed with an attack.

```
Nedzad, choose first corrdinate to attack: 2
Nedzad, choose second corrdinate to attack: 4
Miss!

Jaca, press ENTER to play!
```

Figure 11: After attacking coordinates are entered

The same process repeats until one of two players wins a game. The next state of a game used in this document is show below. The following figure shows what is displayed after the second player presses ENTER.

```
Jaca, press ENTER to play!

*********YOUR BOARD**********          ******BOARD YOU ATTACK******

1  2  3  4  5  6  7  8  9  10          1  2  3  4  5  6  7  8  9  10
o  o  o  o  o  o  o  o  o  1           o  o  o  o  o  o  o  o  o  1
o  A  o  ~  o  o  o  o  o  E  2         o  o  o  o  o  o  o  o  o  2
C  A  o  o  o  o  o  o  o  E  3         o  o  o  o  o  o  o  o  o  3
C  A  o  o  o  o  o  o  o  o  4         o  o  o  o  o  o  o  o  o  4
C  A  o  o  o  o  o  o  o  o  5         o  o  o  o  o  o  o  o  o  5
o  A  o  o  o  o  o  o  o  o  6         o  o  o  o  o  o  o  o  o  6
o  o  B  B  B  B  o  o  o  o  7         o  o  o  o  o  o  o  o  o  7
o  o  o  o  o  o  o  o  o  o  8         o  o  o  o  o  o  o  o  o  8
o  o  o  o  o  o  o  o  o  o  9         o  o  o  o  o  o  o  o  o  9
D  D  D  o  o  o  o  o  o  o  10        o  o  o  o  o  o  o  o  o  10

Jaca, choose first corrdinate to attack:
```

Figure 12: Other player's turn to attack

*1 – look at Table 2 in Page 5

As observed from Figure 12, the second player's board is updated at position (2,4) which is attacked by the first player most recently.

If a ship gets destroyed, players are notified about it.

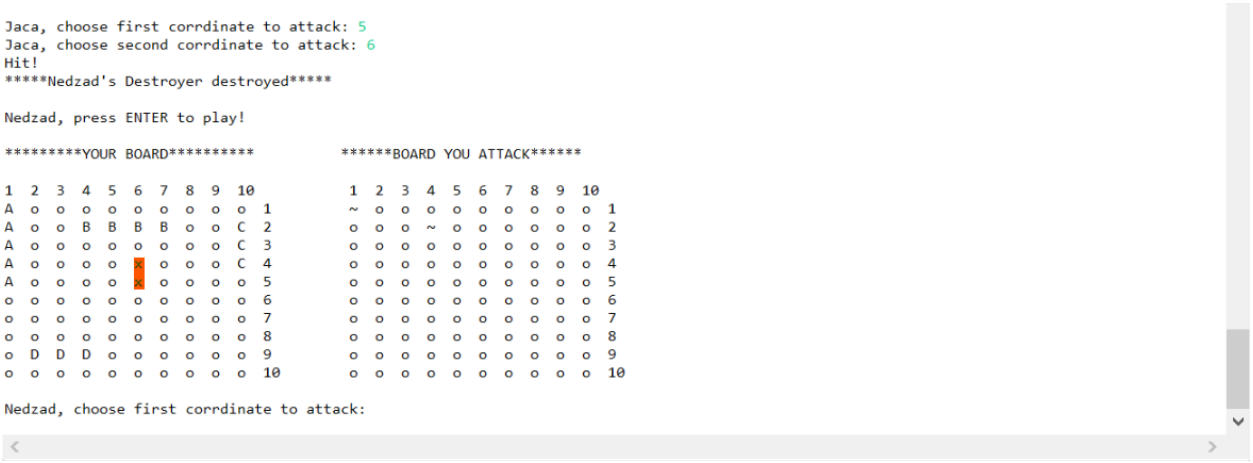

Figure 13: Player's ship destroyed

As rules of the game state, a game is over if one player loses all the ships. The winner of a game is a player who destroys opponent's fleet first.

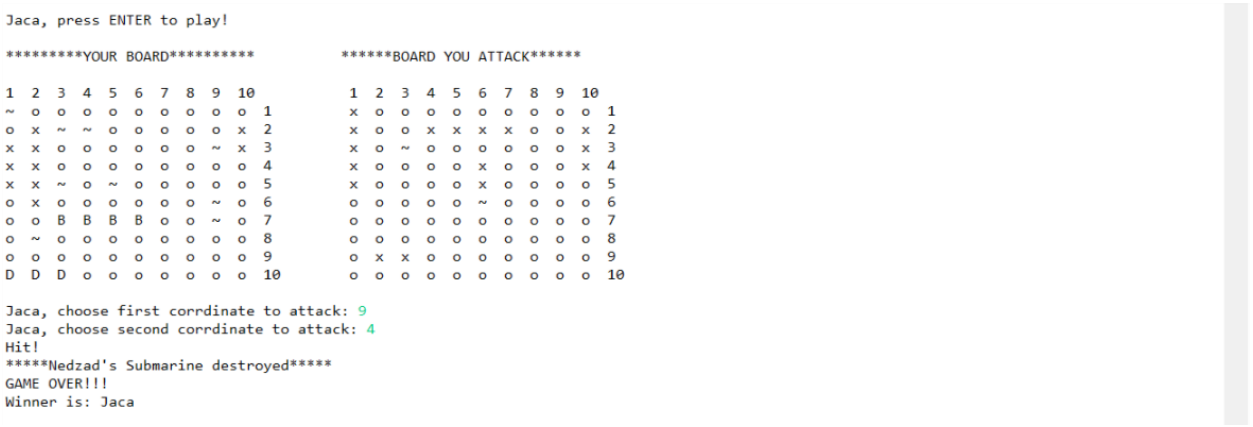

Figure 14: Game over and winner announcement

*1 – look at Table 2 in Page 5

REFERENCES

[1]"Battleship (game)", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Battleship_(game). [Accessed: 03- Oct- 2018].

[2]"Fleet Battle: Battle Series - a game similar to Hasbro's BATTLESHIP ® - only better!", Smuttlewerk.com, 2018. [Online]. Available: http://smuttlewerk.com/fleetbattle. [Accessed: 03- Oct- 2018].

*1 – look at Table 2 in Page 5