DOCUMENTATION FOR SEMESTER PROJECT:

BATTLESHIP GAME

GROUP: LIGHTNING TORPEDOS

GROUP MEMBERS:

1. ASADUJAMAN NUR
2. MUHAMMAD AMJAD BIN ABDUL MALIK
3. JOHNSON KWABENA

**Table of Contents**

**1. Introduction**

*1.1 Purpose*

*1.2 Scope*

*1.3 Definitions, Acronyms, and Abbreviations*

*1.4 References*

*1.5 Overview*

**2. The Overall Description**

*2.1 Product Perspective*

*2.2 Product Functions*

*2.3 User Characteristics*

*2.4 Constraints*

*2.5 Assumptions and Dependencies*

**3. External interface Requirements**

*3.1 User Interfaces*

*3.2 Hardware Interfaces*

*3.3 Software Interfaces*

*3.4 Communications Interfaces*

1. **System Features**
2. **Other Non-Functional Requirements** 5.1 Performance Requirements

5.1.1 Capacity

5.1.2 Dynamic Requirements

5.1.3 Quality

*5.2 Software System Attributes*

3.6.1 Reliability

3.6.2 Availability

3.6.3 Security

3.6.4 Maintainability

5.3 Business Rules

**6. Other Requirements**

Appendix A: Glossary

***1.1 Purpose***

This document applies to the Battleship game that we are working on during this semester. This document describes all of the steps and the processes that we have done during a month of developing and creating our semester project, which is the Battleship Game that is implemented by using C Programming language.

The reader is assumed to have basic knowledge of C Programming language. Knowledge and understanding of UML diagrams is also required.

INTRODUCTION

The purpose of this project is to explore the basics in the C language, including its data structures and graphical components. The purpose is not to define and develop new and original software or academic knowledge; hence a game of Battleship serves as a case study, for exploring the basics of software design and programming in Java. By using a simple game as a case study, the focus quickly arrives at software design, structure and code, thus skipping the art of defining all software features from scratch – a time consuming process. The intended reader for this report is probably a BA student, doing a 1st module report at Computer Science at Roskilde University, or equivalent. The experienced programmer will probably only find this report interesting, when reading it from a teachers perspective. That being said, there are of course several reasons to continue reading. Firstly, the choice of programming a game of Battleship, allows one to gain experience with both the concepts of the C programming language and the details and methods of data structures. Secondly, it also allows for experimenting with user interface design, without the risk of getting bogged down in user interface details, compared to designing more elaborate software systems. And finally, as pointed out in the opening sentence, the features required are more or less pre-defined, which might leave out important aspects of software design, but nonetheless the choice of creating a Battleship game is beneficial to a one-person, BA student project, having only a limited timeframe.

The game of Battleship is a classic game, but several stories about its origin apparently exist. At neweranet.com it is suggested that the Battleship game has its origin in Russia, invented by Russian soldiers between 1917 and 1922. It is also suggested that a predecessor of the Battleship game existed as far back as 18901 . On wikipedia.org the writer John Toland is referenced, suggesting that American convicts sat in their cells during the 1920’s, shouting to each other their strategic moves and results thereof2 . Undoubtedly this would have been possible, since the game can be played using only pen and paper. According to gamesmuseum.uwaterloo.ca the first commercial version of the Battleship game was produced in the 1930’s, crediting French soldiers for inventing the game during the First World War. Although the history of the Battleship game is interesting, this project isn’t a history assignment. The characteristics of the Battleship game are of greater interest. The game of Battleship normally features two players playing against each other, each having two game boards, divided into squares of equal size. Each game board typically has 10 squares both horizontally and vertically. Each player also has five ships to be placed on her own, private game board, which the other player cannot see. The ships have to be placed within the board, and the players have to place all the ships on their private boards, and cannot leave any ships out. The second game board is then used by the players to record the results of the shots they in turn have to fire on each others private boards. When, and if, a player hits a ship on the other players board, the shooting player has the right to shoot again, continuing until nothing is hit on the enemy’s game board. When a player is successful in sinking an enemy ship (hitting all parts of it), he receives a certain amount of points, dependant on the type of ship that was sunk. There are five ships, the largest and most valuable being the aircraft carrier. There is a battleship, the second largest ship, and a destroyer and a submarine which are equal in size and value. The smallest ship, and often the most difficult to find on a game board is the mine sweeper. The players keeps taking turns at firing shots, and the game ends, when one of the players has lost all ships on her own board. The other player then wins, since points are given when a ship is sunk. The approach taken in this project is based on an analysis of other, existing Battleship games ‘out there’. The features and ideas of these other implementations will be distilled into a specification 1 http://neweranet.com 2 http://www.wikipedia.org 2 and possible design for a new Battleship game, which will then be discussed thoroughly, carefully considering alternatives and consequences. This discussion will lead into an actual software design for a new Battleship game, taking into account desirable design pattern considerations. More concretely, actual well documented and well commented Java code will result from this discussion and the laid out software design. My motivation for working on the project, which is defined above, stems from the desire to better my Java understanding and coding skills. This semester (autumn 2007) I have been following my first Java course at the University of Roskilde, and I then decided that the best way to further my abilities would be to design and program a game of Battleship, all things considered. I also work on my own, and therefore need to choose a subject and project goals that are not too comprehensive, given the time and space available. The goal of this project, then, is to develop a basic implementation of a Battleship game. The game will allow a single player to start the game, and play a round against the computer. To accomplish this, a basic user interface will be created, that displays a Battleship game board. On this board the computer will place ships when the game is started. Then the player must fire shots at this board until there are no more ships, at which point the player wins the game (she can never loose). For each action the player takes, the result is displayed in a status message. If a ship was destroyed, points are assigned to the player’s total score. Of course, the underlying Java code features (classes, their methods and data structures) that supports this specification of the game will also be developed. This is also discussed in section 2.2.

The purpose of this project is to have you design of a fairly complicated project using concepts we have learned and then implement the solution using some of the code that we wrote in the previous projects along with new code, and then test your game.

For your CSCI 2312 Project, you will develop a simple battleship game. Battleship is a guessing game for two players. It is played on four grids. Two grids (one for each player) are used to mark each players' fleets of ships (including battleships). The locations of the fleet (these first two grids) are concealed from the other player so that they do not know the locations of the opponent’s ships. Players alternate turns by ‘firing torpedoes’ at the other player's ships. The objective of the game is to destroy the opposing player's entire fleet. In our game, ‘firing a torpedo’ will be allowing the player to take a guess at where on the grid their opponent may have placed a ship.

In the requirements, we will set forth other simplifying rules to limit the scope of this project.

Requirements

Given the requirements as a rough specification, you are to design the classes and implement the game. In our imaginary game company, the requirements below were developed by the Product Development Team and your instructor is the Product Owner. You are in full control of the choice of classes (please use classes appropriately or points will be deducted), data structures, algorithms, internal file format, detailed user interface scheme, or any other pertinent design decisions you need to make. As the Product owner, I care that it compiles and runs like it is supposed to, meets all the functionality and requirements I have set forth, and is easy to play and understand.

The Battleship game you are designing and implementing is a simplified version of the electronic Battleship game played in one player mode.

The game is played on four grids, two for each player. The grids are typically square and in our case will be 10 by 10. The individual squares in the grid are identified by the x coordinate (indicated by a letter) followed by the y coordinate (indicated by a number). The following is an example of a 5 by 4 grid with an X in the position B3.

A B C D E

X

1

2

3

4

Each player uses two grids. Each player uses one of their grids to arrange their ships and record the torpedoes fired by the opponent. On the other grid, the player records their own shots and whether they hit or missed.

Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a certain number of consecutive squares on the grid (sizes of ships are in the following table), arranged either horizontally or vertically. The number of squares for each ship is determined by the type of the ship. The ships cannot overlap so only one ship can occupy any given square in the grid. The types and numbers of ships allowed are the same for each player.

Ship Type and Number of Grid Squares

Ship Sizes:

* 1. aircraft carrier 5
  2. battleship 4
  3. submarine 3
  4. patrol boat 2

The game is played in rounds. In each round, each player takes a turn to fire a torpedo at a target square in the opponent's grid. The opponent then indicates whether the shot was a hit (a ship occupied the square) or a miss (there was not ship in the square). If the shot is a “miss", the player marks their primary grid with a white peg (X in our game); if a "hit" they mark this on their own primary grid with a red peg (O in our game). The attacking player then indicates the hit or miss on their own "tracking" grid with the appropriate color peg (red (0) for "hit", white (X) for "miss") so that they can understand where the opponent’s ship might be.

In the board game, once all of the coordinates of a ship have been hit, the ship is sunk, and the ship's owner announces “You sunk my battleship! (Or whatever the particular ship that was destroyed). For our purposes, we will consider a battleship sunk if the opponent has a single hit. When all of one player’s ships are sunk, the other player wins the game.

For your game, you will create a one-person version of the game where ‘the computer’ will play for the second player.

At the beginning of the game, you will read a file called ship\_placement.csv which contains the type of ship, the first grid square for the ship placement, and whether the ship is placed vertically or horizontally (V or H in the field). The file will be in csv format (comma separated values). This is a common format and is comma separated (instead of being on separate lines). There will be commas between the values. Blank values will just have a comma noting to go to the next field (the game input should not have blank fields so you should handle the case where a field is blank). If you want to view the file, often this will be opened by a spreadsheet unless you specifically open it with a text editor. Do not open it with Microsoft Word, as this may change the format. The first line of a CSV file notes the data descriptions as follows:

TypeOfShip,Location,HorizOrVert

I have provided several sample files which contain good scenarios and scenarios with placement issues that you will need to handle using exception handling. Your game should run with any of these files, but should also be able to run with any valid file in the correct format. You will need to check whether all ships were included in the input file (and appropriate action to take if not), whether all ships have been placed, whether they fit on the board in the configuration given, and whether more than one ship occupies a space (which is not allowed) when you read the input file from the user and how to recover if an error occurs.

You will then need to randomly position the computer’s ships on the grid taking into consideration the same factors as you did for the player’s input.

You will need to prompt for and allow for the user to input their next guess in the form of a letter (A through J) and a number (1 – 10) indicating where they are targeting for their torpedo and you should error check the input. In our simplified game, you will determine if the torpedo shot was a hit or a miss. If the shot was a hit, consider the ship to be sunk. You should display a hit or miss, whether the ship was sunk and which one, and display their tracking grid so they know what they have guessed and where they have made hits. The entire ship which was hit will display as sunk.

After the user takes their turn, you must have the computer randomly select a shot that they have not previously taken. Then you must display to the user what the computer guessed, whether it hit any of the player’s ships, whether a ship was sunk, and then display the player’s placement grid showing where ships are located and what has been hit.

You should continue this until someone wins or quits the game – meaning you should allow the player to gracefully quit at any turn.

At the end of the game, you should indicate the game is over and who the winner was. You should also allow the user to quit the game by entering a Q when prompted for their next guess. If a player decides to quit the game, the grid with all of their guesses and the locations of the computer’s ships should be displayed.

Overall System Design

## 3. Architecture

### 3.1 Requirements or Use Cases

* Requirements
  + A very simple interface inside Second Life to allow the user to play the game.
    - The interface should be simple to build, yet contain enough detail to see what is taking place and what each component is.
    - The interface should also be detailed enough to allow full functionality of the game, providing the user with proper control structures and feedback.
    - Polish up the interface, and make it look pretty.
  + The AI opponent will be governed by an AI engine that I will build. How it plays the game will be governed by this engine.
    - The AI engine will contain enough logic to prune off certain areas of the game board, and reduce the set of possible boat locations. This will also increase the probable likelihood that an attack will be a successful hit.
    - The AI engine should be able to track previous hits and misses, and make an intelligent evaluation of where the next best attack should take place. This information is also used in the pruning process, but is not dependant on it to succeed. However, since the selection of where to attack next is based in-part on random luck, the AI engine will also be required to make similar random choices on where to play next.
    - Allow the engine to control two AI agents, thus playing the game autonomously
    - If we know that the same user is playing repeatedly then we could benefit from modifying our AI engine to include a learning algorithm. But, this is pointless if we can’t track the identity of the user playing, and verify they are the same every time.
* Use Cases
  + Human vs Human
  + Human vs AI

### 3.2 Architecture or Design Space

* Game Rules:
  + Both you and your opponent have a 10 x 10 grid. On this grid you will place 5 ships in random locations. Each ship is of a specific size, and will take up a certain number of spaces on the grid. The ships can only be placed on the grid in a North-South or East-West direction, and not along the diagonals. Your job is to attack your opponent at a specific location, and they will tell you if your attack was a hit or a miss. If you hit one of their ships, and it sinks, then they must tell you what ship you just sank. The game is won after you sink all your opponents’ ships, or vice-versa.
* Ship Sizes:
  + aircraft carrier 5
  + battleship 4
  + submarine 3
  + patrol boat 2

### 3.3. Tasks

At this point, the tasks were relatively small in scale. I’ve mainly built simple code for the AI engine that picks a random position, and ignores any moves that have previously been made.

### 3.4 Testing

The game has three modes; player vs player, which was easy to test since I could play against myself, then player vs AI, which took a bit of testing to get the AI to correctly interact with the elements in the game, and then AI vs AI, which didn’t require hardly any testing since the same code gets called by both AI’s, but for their own playing grid.

## 4. Results and Analysis

From what I can tell, the game appears to work as intended. The AI engine isn’t the smartest thing, but it is able to play the game.

## 5. Conclusions

### 5.1 Summary

The main thing to observe here is that, although I’m calling this an AI engine, it’s not really that intelligent. The AI can only do a very small set of things with respect to the game Battleship, and beyond that it is useless.

### 5.2 Future Work

This game could easily be improved by adding better graphics, sound, and modifying the interface to be more informative.