**Requirements Analysis and Design Document Template**

**(The requirements and design document template is given at the end of this document)**

Why do it?  The requirements analysis which you perform and document before you start designing and writing a computer program is useful for two very good reasons:

1. It provides you with a means of confirming and agreeing with the person who commissioned you to develop your program (your client) that you have an appropriate understanding of what is needed.
2. Once you have agreed with the client that the analysis is correct, it provides you with a working document, against which, at any time, you can check if your design and/or program is meeting requirements.

Requirements analysis is a topic which you will learn more about in detail if you study Software Engineering in later years but there are some basic principles which are usefully adopted right from the start.  The sections which should be present in the requirements analysis which you do should be based on the following headings:

**Statement of the problem**

Make a clear statement of the problem which you are trying to solve at a very high level – in other words, unambiguously but with only the essential detail.  In other words, say what your program sets out to do.

In the assignments you will do as programming exercises, this will often be a repetition of the set problem.  For example, you might be set the following problem: “Write a program to find the average of a series of numbers”.  Your Statement of the Problem might simply say: “A program to calculate the arithmetic mean of a series of numbers”.  This statement stands on its own but it doesn’t say very much about your program **except** what it does.  Notice also that even at this stage it says a bit more than the set problem – “average” has been redefined to mean “arithmetic mean”, not mode or median, so the process of refinement has already begun.

In real life, requirements are likely to be derived from extensive consultation with users.

**Users**

Often, especially in programming exercises, it will be obvious who the users are.  You should still make it clear who they are.  For example, “The program will be used by primary school children to help them to understand the concept of arithmetic mean” implies a very different solution from “The program will form a module within a financial package which is used by accountants for assessing clients’ tax liabilities”.

Both are equally legitimate reasons for writing the program to calculate means, yet the tasks involved in designing the two programs are likely to be completely different.  In what ways do they differ?

You should also identify the range of users: will any have special needs?  In what way?  What prior experience with computers do they have?  How motivated will they be to use the program?  (Can you think of any other questions?)

Page Break**Details**

1. **Assumptions**

State the assumptions you are making (and why if necessary).  For example, “the numbers input are integers” or “the numbers are real and will be given to 2 decimal places”; “users will understand the concept of arithmetic mean”; “no more than 50 numbers will be input in any run of the program”; “some users will be visually impaired”.  These are all assumptions about the requirements of the program which have implications for its design.

1. **Input and Output**

How is the computer to accept input and give output? For example, “errors in input will be handled by…”; “input will be received from the rawdata module as an array of 50 floating point numbers”; “output will be to the screen with no explanatory message”; “output will be to a speech synthesiser with an explanation of its meaning in non-technical terms”.

All of these are requirements for input and output which will emerge from a requirements analysis and which should be specified in the requirements analysis document.

1. **Standards**

State any relevant standards.  For example, “The program will be implemented in Microsoft Windows and will meet all interface guidelines for Windows programs”.

**Performance**

Measurable performance requirements should be noted at this stage.  These may be **system** or **usability** performance targets, for example, “The program must calculate the result accurately to 35 decimal places 100% of the time; 90% of the target user group will be able to calculate an average of five numbers after 2 minutes training.”  Setting measurable performance targets like these enables you to test the system later on to see if it is good enough for its stated purpose.  If your client has already agreed the performance standards then it also protects you from statements like “oh it’s very good but we expected it to be a bit faster”.

**Functional and Non-functional Requirements**

Most of the ideas above are to do with functional requirements – what the system is to do and not to do; how the system should react in situations and so on.

Equally important, the system needs put in a context.  To some developers, these aspects are much more interesting and are known as non-functional requirements.  Non-functional considerations include product usability, the user experience, the portability and reliability of the system.  They also include process considerations, such as delivery, how the implementation is done and to what standards.

Finally, there are external non-functional requirements such as other programs, ethical and legal considerations and safety.

Given that modern software products (such as games) often succeed or fail on the basis of the quality of the user experience, some of these non-functional requirements can be very important.  You will consider functional and non-functional requirements much more in years to come; for now, just be aware of them when you are thinking about software requirements and design.

This is a template for an object-oriented design document. It provides an outline with headings. You do not have to follow this template, but it can provide you with a guide. The notes in [italics] are to help you and should be removed. Please delete the instructions on the first two pages of this document before submitting it!

**Battleship Game**

**Date: 18/11/2019**

**Student Names:** Miklos Mayer

**Statement of requirements**

Allow the user to play against the computer in a game of Battleships. This game is played on a 10x10 grid on which the computer will randomly generate a battle fleet which will consist of 9 ships. The player will select a square to try and locate the computers battle fleet.

The program is designed to be used by children. The user does not need an in-depth knowledge on computing systems but will need basic knowledge on how to use a desktop computer. The user should *know how to play a basic game of battleships in order to use the program.*

*The inputs of the program are the user’s mouse click on choosing a square to locate the fleet or select a button. Alternative inputs are short commands through the keyboard.*

The outputs of the program are the battle fleet position, the players score and whether a fire was a hit or a miss.

**Requirements**

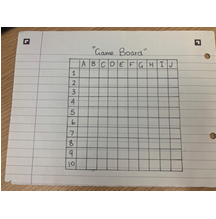
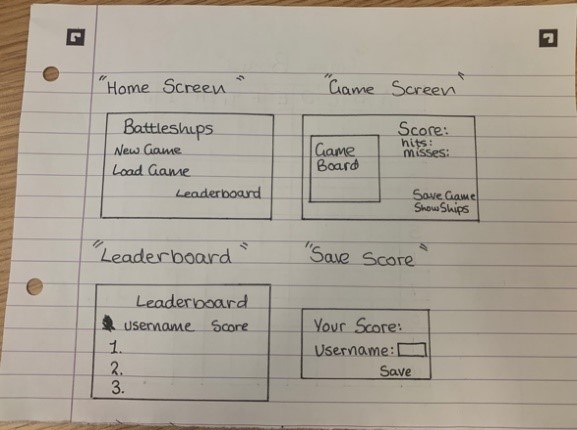
**Functional Requirement**

1. The system shall open with a Main Menu.
   1. This page allows the user to select the starting functions of the program.
2. The system shall contain a Start New Game option.
   1. This function is in the Main Menu. It will initiate a completely new game and activated by a mouse click or a valid keyboard command.
3. The system shall display a 10 x 10 square grid as the playfield.
   1. The x axis is labelled by the alphabet from A to J. The y axis is labelled by ascending numbers from 1 to 10.
4. The system shall randomly place its battle fleet on the grid.
   1. The ships shall not overlap each other, and they shall not touch each other (even diagonally). They should not be placed diagonally.
5. The player shall play against the computer.
   1. It is a one player game, the computer does not shoot back, so the player does not place any ships on the grid.
6. The system shall allow the player to choose a square in attempt to locate the computer’s battleships.
   1. This choose called fire.
7. The system shall prompt the user to choose another field if it is previously fired upon.
8. The system shall indicate whether the fire hit or missed a battleship.
9. The system should indicate if a ship sinked.
10. The system shall display the player’s hits and misses as scores and the number of rounds passed.
11. The system shall save the current game after every fire.
12. The Main Menu shall have an option to Resume the last unfinished game.
    1. It will reveal the playfield as in New Game but with the saved scores and positions.
13. The system should not display the Resume option if there is no saved game.
14. The system should display the final score after the game ends.
    1. The game ends when the player hit all the ships on the battlefield.
15. The system shall prompt the user to type in a username after a finished game.
16. The system shall save the player to the Leaderboard.
    1. The final score will be calculated from the hits, misses and passed round.
17. After the player saved to the leaderboard, the system shall display the Finished Game Options: ‘Start a New Game’ and ‘Exit the program’.
18. The Main Menu should have an option to show the Leaderboard.
19. The system should show player scores ordered from highest to lowest on a page known as ‘Leaderboard

**Non-functional Requirements**

1. The Java Runtime Environment (JRE) shall be installed on the system.
2. The program shall be played by a keyboard or mouse.
   1. This program cannot be played on a mobile device.
3. The program must not contain any copyrighted images.
4. The program must not contain any violence or inappropriate content for children.

**User Interface**

****

**Use Cases**

|  |  |  |  |
| --- | --- | --- | --- |
| New Game | | | Alternatives |
| 1 | USER | Choose new game |  |
| 2 | SYSTEM | Launches game |  |
| 3 | SYSTEM | Randomly place battlefleet on 10x10 grid |  |
| 4 | SYSTEM | Display grid on screen |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Firing a Shot | | | Alternatives |
| 1 | USER | Choose a square | A1, A2 |
| 2 | SYSTEM | Determine whether a battleship was situated on the square that the user chose |  |
| 3 | SYSTEM | Display whether the fire hit, sank or missed a battleship |  |
| 4 | SYSTEM | Increment and display ‘Missed’ or Hit’ score |  |
| 5 | SYSTEM | If missed increment Round |  |
| 6 | SYSTEM | If all the ships found display ‘Final score’ |  |

|  |  |  |  |
| --- | --- | --- | --- |
| A1 Firing a Shot alternative | | | Alternatives |
| 1 | USER | Choose a square previously fired on |  |
| 2 | SYSTEM | Registers that the square has been previously selected |  |
| 3 | SYSTEM | Ask user to choose a different square |  |

|  |  |  |  |
| --- | --- | --- | --- |
| A2 Firing a Shot alternative | | | Alternatives |
| 1 | USER | Choose not a square |  |
| 2 | SYSTEM | Registers that the user did nor chose a square |  |
| 3 | SYSTEM | Ask user to choose a different square |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Resume Game | | | Alternatives |
| 1 | USER | Choose ‘Resume Game’ |  |
| 2 | SYSTEM | Access previously saved game |  |
| 3 | SYSTEM | Display game on screen |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Show Leaderboard | | | Alternatives |
| 1 | USER | Choose ‘Show Leaderboard’ |  |
| 2 | SYSTEM | Show the leaderboard |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Save Score | | | Alternatives |
| 1 | SYSTEM | Ask the user to enter a name |  |
| 2 | USER | Enters his/her name | A3 |
| 3 | SYSTEM | Save the score and display the Finished game options. |  |

|  |  |  |  |
| --- | --- | --- | --- |
| A3 Save Score alternative | | | Alternatives |
| 1 | USER | Enters an empty name |  |
| 2 | SYSTEM | Ask the user again to enter their name |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Exit Game | | | Alternatives |
| 1 | USER | Choose ‘Exit’ |  |
| 2 | SYSTEM | Stops program |  |

**Classes**

**Candidate Classes**

|  |  |  |
| --- | --- | --- |
| **Candidate Classes** | **Accept / Reject** | **Reason for rejection** |
| Game of Battleships | Accept | Handle the game functions |
| Position | Reject | Better suited to be a field in the Ship class |
| Player | Accept | Contains information about a player |
| Username | Reject | Field of Player |
| Ship | Accept | Contains information about a ship |
| Battleship | Reject | Just the type of the Ship |
| Cruiser | Reject | Just the type of the Ship |
| Destroyer | Reject | Just the type of the Ship |
| Submarine | Reject | Just the type of the Ship |
| Menu | Accept | Top level coordinator |
| Option | Reject | Too vague |
| New Game | Reject | newGame could be a method in the Game of Battleships class |
| Shot | Reject | fireShot could be a method in the Game of Battleships class |
| Grid | Reject | A list of squares |
| Field | Accept | A field of the Battlefield |
| Fleet | Reject | List of Ships |
| Score | Reject | Better suited to be a field of Player |
| Leaderboard | Reject | List of players stored in a file |

**Class Descriptions including Responsibilities, Fields and Methods**

Menu – Handle all the interactions between the user(s) and the computer

- Fields: game

- Methods: main(), newGame(), saveGame(), loadGame(), showLeaderboard(), saveScore()

GameOfBattleships – Handle all the interactions between players

- Fields: player1, player2, rounds, activePlayer

- Methods: play(), displayGrids(), endGame(), getRounds(), fire(), getCommand()

Player - Stores information about a player who interact with the game

- Fields: hits, misses, fleet, battleField

- Methods: calculateScore(), addHit(), addMiss(), getHits(), getMisses(), placeShips()

AI – children of the Player class, controlled by the computer

- Additional/overriden methods: placeShips(), fire()

Field – One field of the battlefield. Handle hits and store information about fires and ships

- Fields: position, hasShip, isFired, isSank, shipId

- Methods: hasShip(), isFired(), isSank(), getHit(), placeShip()

Position – Helper class for Field, ensures that fields can only have valid positions (e.g. A4)

- Fields: position

- Methods: validatePosition(), calculatePosition(),getColumn(), getRow()

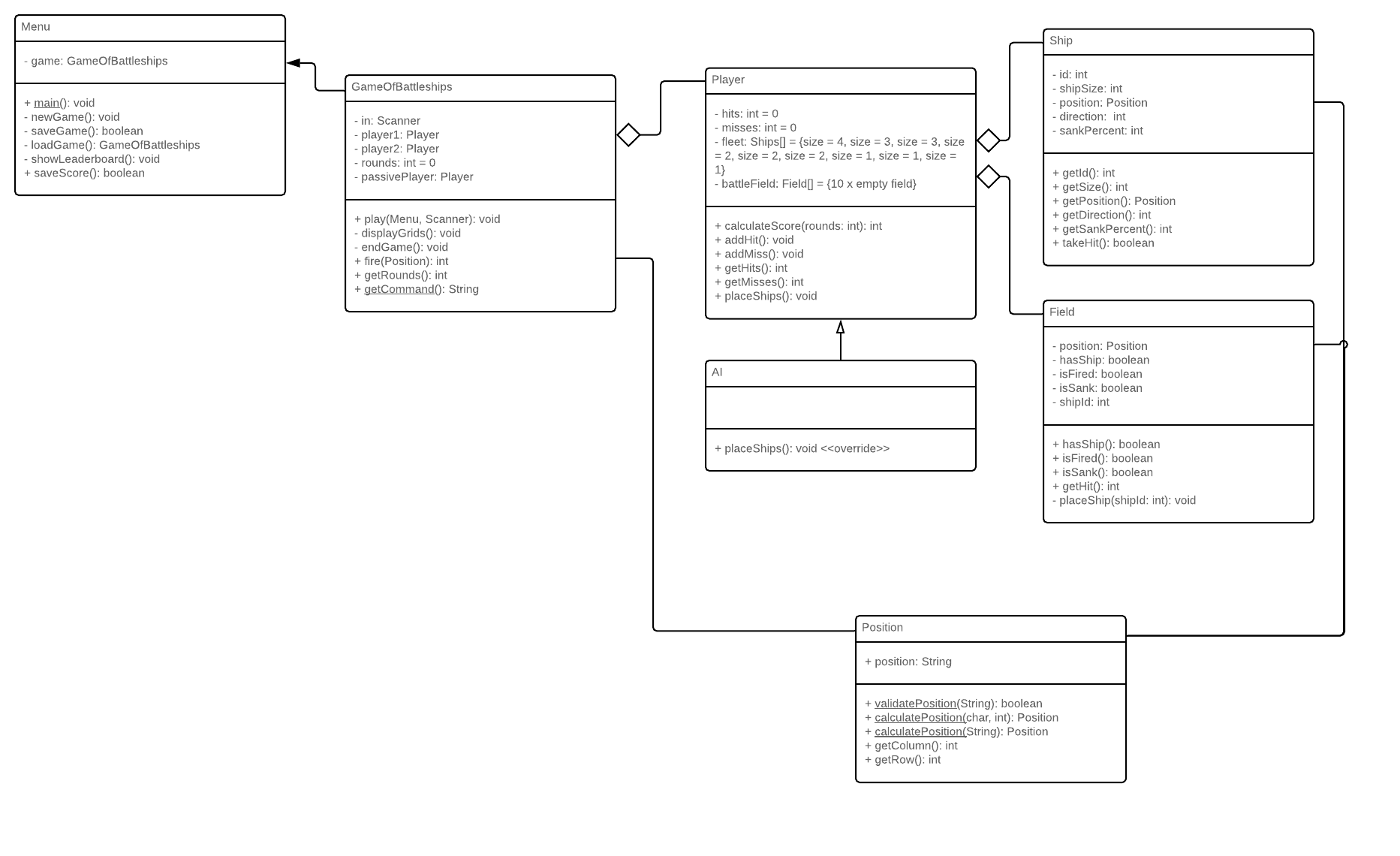
Ship – contains information about a ship

- Fields: id, shipSize, position, direction, sankPercent

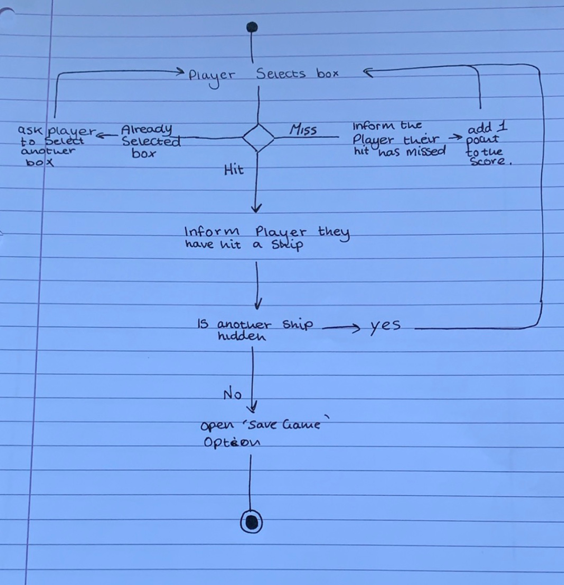
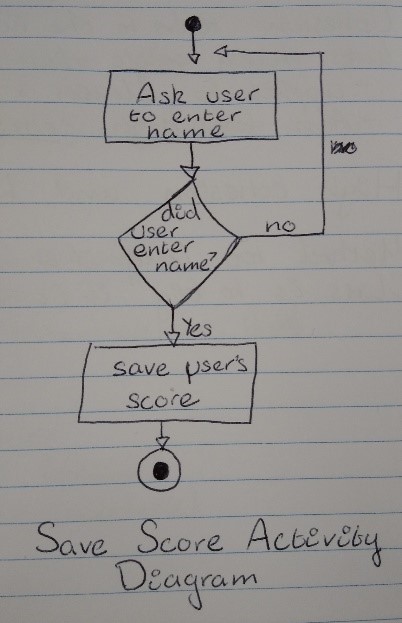
- Methods: getSize(), getDirection(), getSankPercent(), getPosition(), takeHit(), getId()

TODO decide to delete second player here or update requirements and statement

**Class Diagram**



**Activity Diagrams / Pseudocode**

****[Ford any complex or critical methods]