**Requirements Analysis**

**(A 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?)

**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.***

# Reversi

**Date: 14/09/18**

**Student Names:** *Max Fyall*

## Statement of requirements

*[Start by giving a brief description of the problem here]*

*[Includes brief sections on* *Assumptions about the user, Inputs to the program and Outputs from the program]*

Problem: Design a computer version of the board game ‘Reversi’ for two human players.

Inputs: Input from keyboard performed by the human player. (Coordinates from grid)

Outputs: Displays the counters placed by the players on the grid, the number of counters owned by each player and which player has won.

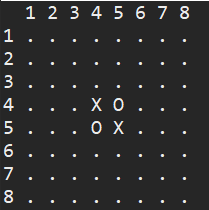
Assumptions: All users will be able to understand the system of coordinates used in the game. The game shall follow the rules of Reversi. The game must be able to accept keyboard/mouse inputs from the user and output these instructions onto the grid, from a keyboard it is assumed theses inputs are in ASCII. There will be only 60 possible positions for the player to choose from.

### Requirements

**Functional Requirements**

*[These should deal with the function of the system, i.e. this will be the main body of requirements that clearly specify what requirements the system shall, should or may meet]*

1. The game shall be played on a 8x8 grid.
2. The game shall have each position on the grid set as either empty, X or O.
3. The game shall be played by two human players.
4. The game shall be won by a majority of the grid being populated by either X or O once all spaces are filled OR by all the other player’s counters being eliminated, whichever comes first.
5. The game shall count the number of counters held by each player.
6. The game should display the number of counters held by each player.
7. The game shall start with pieces as shown in Figure 1.
8. The game shall begin with the X player having the first move.
9. The game shall alternate turns between the two players on a turn-by-turn basis.
10. The game shall only allow a new piece to be placed if it borders an opponent’s piece on a horizontal, vertical or diagonal line.
11. The game shall only allow a new piece to be placed such that it creates a vertical, diagonal or horizontal line to one of the player’s existing pieces going through the opponent’s piece(cols) without empty spaces in between.
12. The game shall only allow a new piece to be placed on a square that is vacant.
13. The game should have a start screen which can start the game.
14. The game should accept user inputs through a keyboard to place pieces which correspond to positions on the grid as shown in Figure 1.
15. The game may have rules displayed on the start screen.
16. The game may have an option to restart the game at the end of the game.
17. The game shall have the ability to save its state.
18. The game shall have the ability to reload from a previous state.

**

*Figure 1*

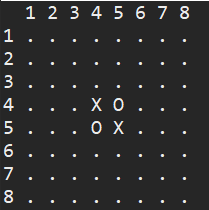
**Non-functional Requirements**

*[These relate to non-functional aspects of the system such as* u*sability, performance or system hardware constraints (e.g. minimum hardware specification), required software etc.]*

1. The game shall run on the Java Runtime Environment (JRE).
2. The game should use a green coloured background on the grid/play screen.
3. The game should be in English.
4. The game should be under an open source licence.
5. The game should run on a computer which has a CPU which can attain a multi-core score of 1500 points in Geekbench 4.3.0.

### User Interface

*[Notes or sketches describing the user interface]*

**

### Use Cases

*[Consider who or what will use the system and how they will interact with it. Give a descriptive sentence for each use case then the USER: SYSTEM dialogue]*

**New Game:**

**1 User:** Presses New Game button

**2 System:** Resets board to default grid.

**Load Game:**

**1 User:** presses load game button

**2 System:** Checks for previous saves - **(Alternative A1)**

**3 System:** Loads previous save.

**Place Counter**:

**1 System:** Checks the availability of moves by the current player - **(Alternatives see A2)**

**2 User:** Places counter on board.

**3 System:** Reads the move and verifies validity - **(Alternatives see A3)**

**4 System**: Confirms move by adding counter to position.

**5 System**: Changes necessary counters.

**6 System:** Updates score for current player

**7 System:** Notify new move for opponent.

**Save and Exit:**

**1 User:** Presses save game button

**2 System:** Saves current state

**3 System:** Returns to start menu

**A2 – Potential Moves (End of Game):**

**1 System:** Notifies player with highest score

**2 System:** Ends the game

**3 System:**  Display the winner and their score

**A3 - Place Counter (invalid move):**

**1 System:** Notifies user of invalid move

**2 System:** Returns to step 1 in main flow

**A1 – No save games found:**

**1 System:** Notifies user there is no previous saves.

**2 System:** Returns to start menu.

### Classes

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

*[For each class state what it is responsible for and list the fields (with type) and methods (type and parameters)].*

**Menu**:

Class responsible for handling the display methods and the main method

* Methods:
  + displayGrid – takes the 2D array grid and displays the values to form a grid for the game.
  + displayMenu – displays the menu when the program is run to give the user options for what they want to do.
  + main – the method where everything is run from
* Fields
  + Int grid[12][12] – a 2D array to store the values of the grid. A reverse grid is only 8x8 but the array has been set to a higher value to prevent array out of bounds errors occurring.
  + Int numbs[8] – a 1D array to store the numbers that act as the coordinates down the edges of the grid to allow users to choose their coordinates for there moves.

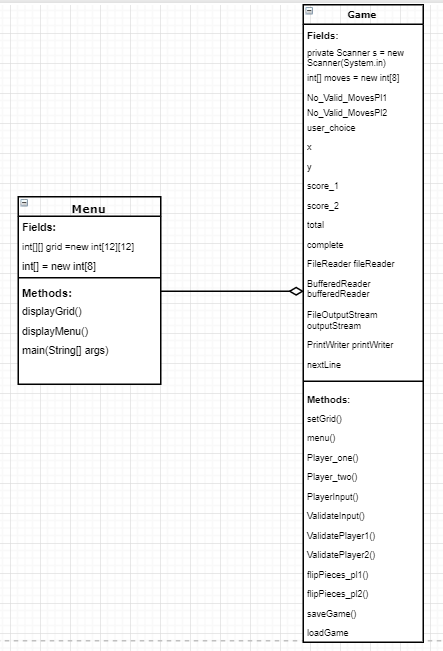
**Game:**

Class contains the core functions of the game reversi.

* Methods
  + setGrid – sets the grid to the default values for the beginning of a new game.
  + menu() – functional menu, allows user to choose what they want to happen when the program runs. E.g. start new game, load game.
  + Player\_one() – calls all necessary methods for the first player’s turn.
  + Player\_two() – calls all necessary methods for the second player’s turn
  + PlayerInput() – takes in two inputs from the player as two variables x and y.
  + ValidateInput() – responsible for ensuring the inputs from the user are valid in terms of the rules of reversi.
  + ValidatePlayer1() – responsible for ensuring that player one’s input is valid and can be used.
  + ValidatePlayer2() – responsible for ensuring that player two’s input is valid.
  + flipPieces\_pl1() – responsible for flipping pieces when it is player one’s turn.
  + flipPieces\_pl2() – responsible for flipping pieces when it is player two’s turn.
  + saveGame() – saves the current game upon the user’s command.
  + loadGame() – loads a previously saved game and continues from where it left off.
* Fields
  + Scanner s = new Scanner (System.in) – ensures that only one scanner is opened.
  + int moves[8] – 1D array that stores all possible moves around the user’s chosen slot.
  + Int No\_Valid\_MovesPl1 – stores the number of valid moves the game can make for player 1.
  + Int No\_Vaid\_MovesPl2 – stores the number of valid moves the game can make for player 2.
  + String user\_choice – stores the user’s input for the menu when the program is run.
  + Int x – stores the x-coordinate entered by the user.
  + Int y – stores the y-coordinate entered by the user.
  + Int score\_1 – stores the score of player 1
  + Int score\_2 – stores the score of player 2
  + Int total – stores the total score of the 2 players
  + Boolean complete – turns to true once the total reaches 64.
  + FileReader fileReader – essential for reading from a file
  + BufferedReader bufferedReader – essential for reading to a file
  + FileOutputStream outputStream – essential for writing to a file
  + PrintWriter printWriter – essential for writing to a file
  + String nextLine – String to store the nextline when reading from a file.

##### Class Diagram

*[Show the relationships between classes]*

**