Table of Contents

Chapter I - Introduction

During this semester, we had an opportunity to involve in a game project for revising our knowledge in Object Oriented Programming. Our project is a remake version of a well-known chess game called Reversi. We would say that this is a challenging project and through this project, we are able to learn and strengthen our OOP knowledge, including coding style, design pattern and techniques.

Our project was developed used Java, a representative language in OOP. We had established a development environment that is well-suited for our project, such as Java Swing which is a desktop development framework.

In this report, we will give a detail description on our project, including techniques and algorithms used in development. The structure of our report is outlined as follows

Chapter I gives an overview on background and tools used in this project.

Chapter II explains the game rules and extra features.

Chapter III focuses on descriptions of Reversi algorithm.

Chapter IV illustrates UML class diagram of the project.

Chapter V shows the evaluation of our project and discuss the possible improvements.

Chapter II - Game Rule And Extra Features

2.1 Background

Reversi is a turn-bases strategy game which has a simple play rule. Each reversi piece has a black side and a white. Our game starts with four chess pieces placed on the middle of the chess board, including two pieces in black and two others in white. The positions of pieces are illustrated as shown in figure 1.

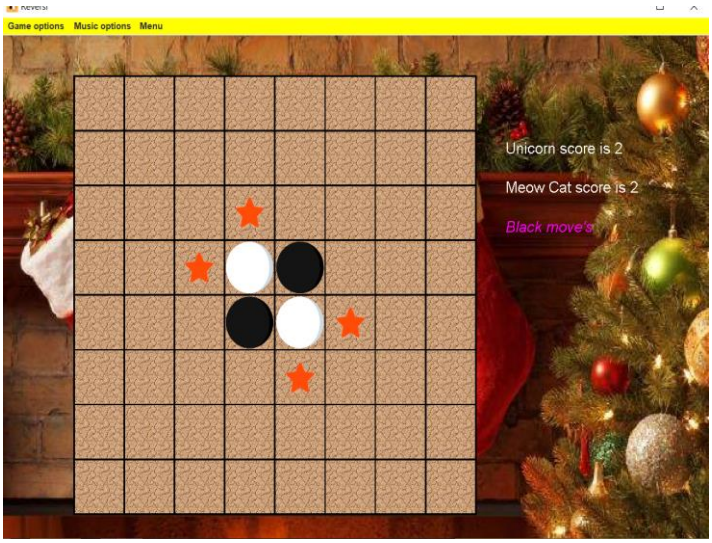

Figure 1. Init state of chess board

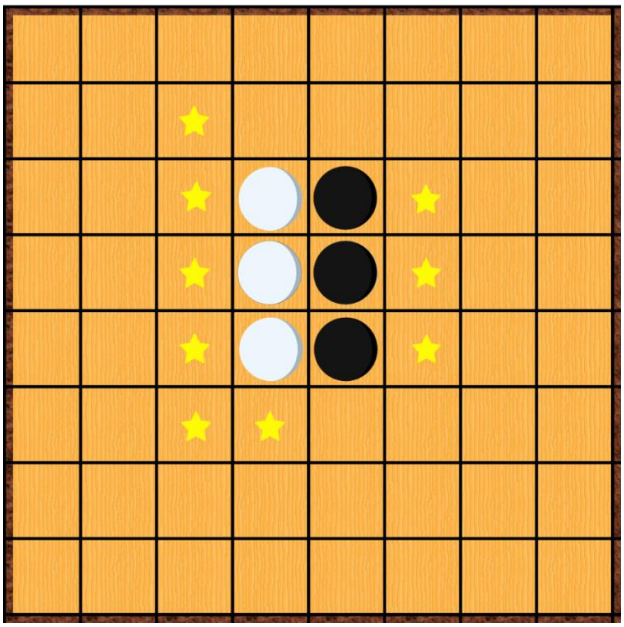

Figure 2 - Several moves of pieces on chess board

On one turn, the player places one piece on the board with his or her piece's color facing up. The player must place the piece so that an opponent's piece, or a row of opponent's pieces, is flanked by his or her pieces. All of the opponent's pieces between that player's pieces are then turned over to become his or her color.

The ultimate purpose of a player is to dominate all the pieces of his opponent, or to flip all his opponent's pieces, changing their color into the color of his pieces. The game is over when none of player can move.

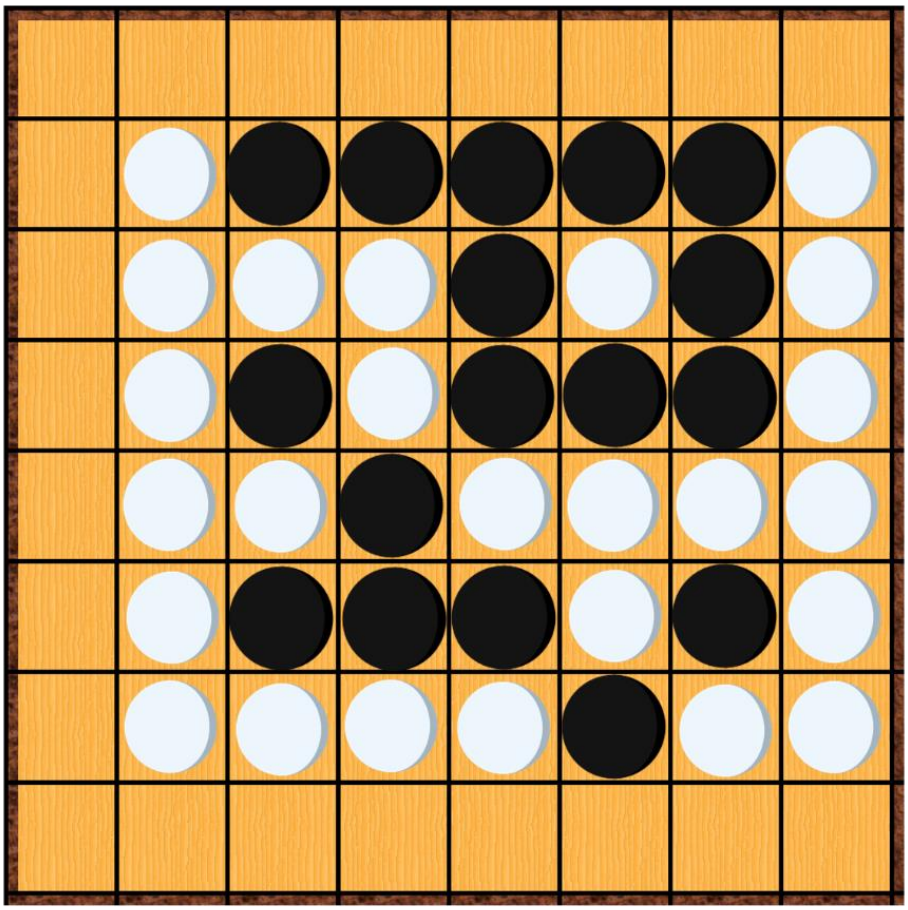

Figure 3. An example of a chess game

2.2 Feature of Product

Our version of Reversi Project consists of two main game mode, including PvP (Player vs player) and PvC (Player vs Computer). In PvP game mode, two players take turn to place their

pieces, until none of player can move, and who has the larger number of pieces will win. In PvC mode, human player fight with intelligent agent machine, applying our self-developed algorithm to operate. Our game can also suggest the next move to players.

Figure 4. Game Options

In addition, we programed our project to be able to change background of the game. In order to create excited atmosphere for players, we added background music in the game. There are several chess' board that player can choose when starting to play game.

Chapter III – Description of Minimax Algorithm

3.1 Gameplay class

+ The function checkPossibleMove:

This function finds all possible moves that suit to the rules of the game. This function uses dynamic programming idea to find it.

$arr_{i,j}.horizontal$ is contained the information about whatever to the left exist a consecutive cell with the different value. $arr_{i,j}.verical$, $arr_{i,j}.leftDiagonal$, and $arr_{i,j}.rigthDiagonal$is the same with $leftDiagonal$

First, suppose that the current player is black. We go through from up left to down right. With each $cell_{i,j}$:

If the $cell_{i,j}$ is white, then

$$arr_{i,j}.horizontal = arr_{i,j-1}.horizontal$$

$$arr_{i,j}.vertical = arr_{i-1,j}.vertical$$

$$arr_{i,j}.leftDiagonal = arr_{i-1,j-1}.leftDiagonal$$

$$arr_{i,j}.rightDiagonal = arr_{i-1,j+1}.rightDiagonal$$

If the $cell_{i,j}$ is black, then

$$arr_{i,j}.rightDiagonal = arr_{i,j}.leftDiagonal = arr_{i,j}.vertical = arr_{i,j}horizontal = white$$

If the $cell_{i,j}$ does not have any chess and

$$\begin{bmatrix} arr_{i-1,j+1}.rightDiagonal = black \wedge check_{i-1,j+1} = white \\ arr_{i-1,j-1}.leftDiagonal = black \wedge check_{i-1,j-1} = white \\ arr_{i,j-1}.horizontal = black \wedge check_{i,j-1} = white \\ arr_{i-1,j}.vertical = black \wedge check_{i-1,j} = white \end{bmatrix}$$

Then, $cell_{i,j}$ is a possible move

3.2 Minimax class:

I use the Minimax algorithm to find the highest score when the BOT has picked a cell. The score at every place on the board would be high or low on the basis of the value of that cell. The higher the good score, the more important it is, and vice versa.

The main idea of Minimax is recursion. At every depth of recursion, this algorithm will list all moves and continue to recur them. If the current depth is a BOT move, this algorithm will choose the move that has the highest score. If more than one of the highest scores, the algorithm will randomly choose any of them. If the current depth is player move, the algorithm will do the opposite. At the maximum depth, this algorithm will compute the region score. The region score is defined as the sum of all cells in which that chess pieces is located. Each cell has either negative or positive weight, depending on its importance.

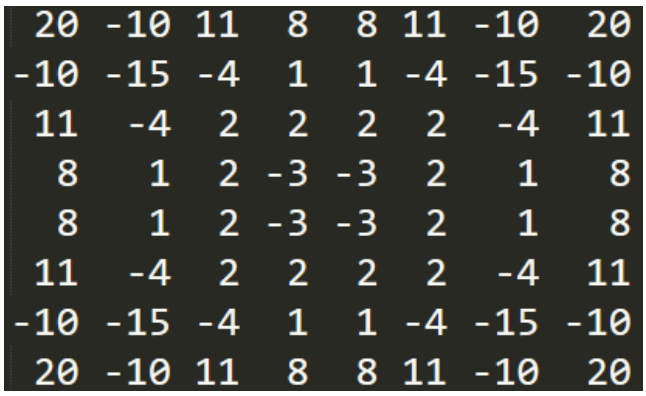

Figure 5. Weight in each cell

This is Pseudocode for Minimax Reversi algorithms

```
function decision(depth, board, curPlayer)

    if (depth == maxDepth || checkEndGame == true)
        return score(board, curPlayer)
    move = findAllPossibleMove(board, nextPlayer)

    if (move.size() == 0)
        return decision(depth + 1, board, nextPlayer)

    for (each move)
        val =  decision(depth + 1, boardForNextPlayer, nextPlayer)
        if (curPlayer is Bot)
            bestMove = max(bestMove, val)
        if (curPlayer is Human)
            bestMove = min(bestMove, val)
    return bestMove


function finalDecision(board)
    move = move = findAllPossibleMove(board, human)

    for (each move)
        val = decision(2, boardForHuman, human)
        if val > bestMove
            bestMove = val
            moveRes = this move
        if val == bestMove
            chose randomly(this move and moveRes)
    return moveRes
```

Chapter IV - UML Class Diagram

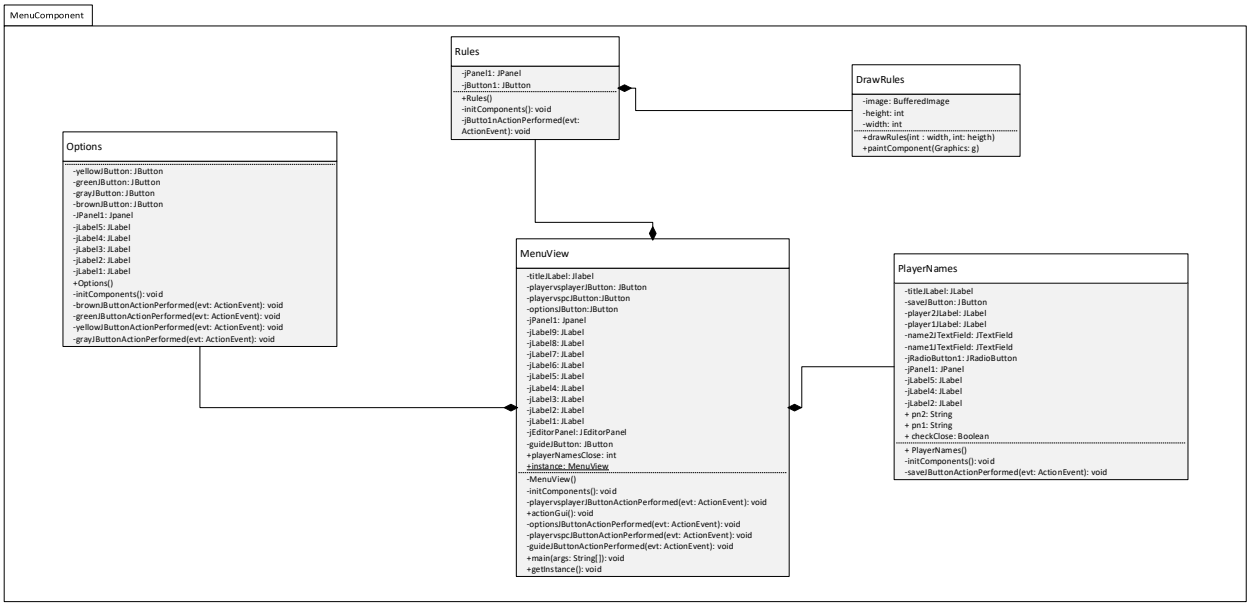

Figure 6. Class Diagram of the Reversi Project

In MenuComponent package, we focus on handling the first menu when starting a game. Main class in this package is MenuView. It will control every class in this package. Options class will show chessboard image options for user choose. PlayerNames is a class for users who input their name. Almost code in this package we use NetBeans to generate.

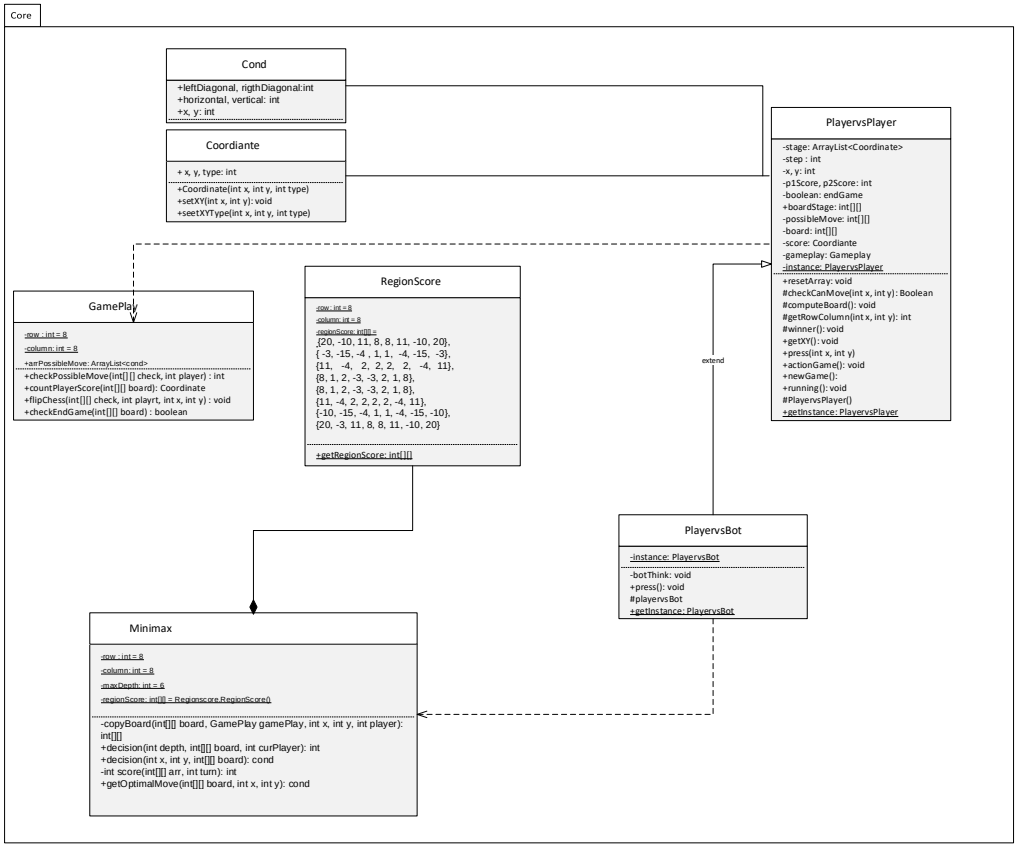

Figure 7. Class diagram of Core Package

In the Core package, we focus on handling the back end of this game. PlayervsPlayer and PlayervsBot use singleton pattern because we want to have only one class at a time. The Minimax class is a class that handles the Minimax algorithm. GamePlay is a class to handle all rules game.

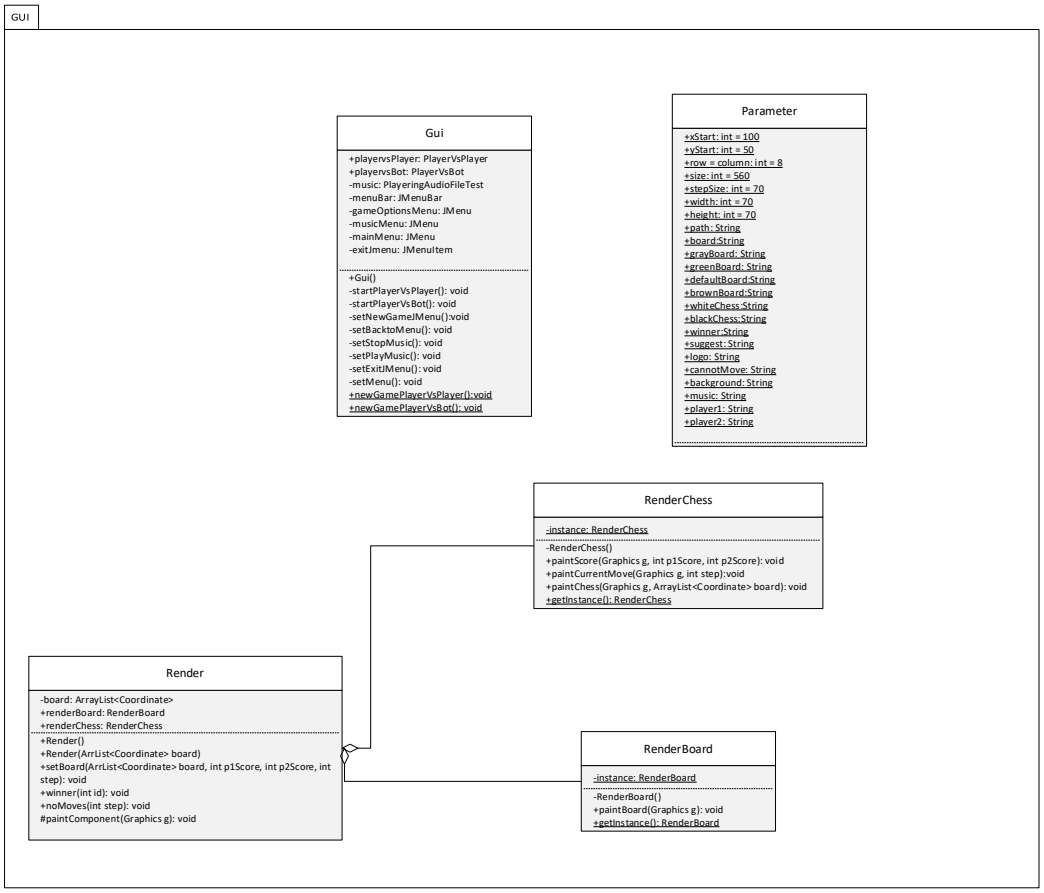

Figure 8. Class Diagram of GUI Package

The focus of the GUI package is on handling the front end of this game. 2 classes support RenderChess and RenderBoard. Render is an extended class from JPanel. To draw all necessary components, it will pass a graphics g to RenderChess and RenderBoard. The parameter class only saves all constant variables, such as the path of the image, the name of the player or the size of the image.

Chapter V - Evaluation

From this project, our team have more experience to work with GUI. Our team has experience dealing with multi-+ in Java. The multithread that our team has faced is the Java swing component. This project is also the first project in which I am incorporating competitive knowledge programming experience. That makes the project code quicker and cleaner for me. Another lesson is from this project I see that the way that Java show bug is not clear.