|  |
| --- |
|  |

|  |
| --- |
| **Android Reversi Game Design Document** |
| Project 3 for CSCE 315 |
|  |
| This Android Reversi console game will allow users to play unlimited reversi games with a friend or against an AI on their Android phones. |
|  |
| **Laramie Goode, Chris Voss, Curtis Franks** |
| **6/26/2012** |
|  |

# 

Table of Contents

[Purpose of Project 2](#_Toc328531021)

[High Level Entities 3](#_Toc328531022)

[Simple Game Mechanics 3](#_Toc328531023)

[Artificial Intelligence 4](#_Toc328531024)

[Implementing on Android 4](#_Toc328531025)

[Low Level Entities 5](#_Toc328531026)

[Game 5](#_Toc328531027)

[*Diagrams of core Reversi Game Design* 6](#_Toc328531028)

[AI 7](#_Toc328531029)

[Hard AI 7](#_Toc328531030)

[Medium AI 8](#_Toc328531031)

[Easy AI 8](#_Toc328531032)

[Alpha-Beta Pruning 8](#_Toc328531033)

[Android Interface 9](#_Toc328531034)

[Works Cited 10](#_Toc328531035)

# Purpose of Project

This project is an implementation of the game Reversi for the Android environment. The game is a strategic board game played with two players. Each player takes turns trying to capture the other player's pieces by forming a straight line from a starting piece of their color to an ending piece of their color. If this starting piece and ending piece form a line that contains the opposing player’s pieces, those opposing pieces are converted to the same color as the starting and ending piece of the line. The game ends when all the spots on the board are filled or when there are no more moves to be made. The only valid moves are one where a piece placed is adjacent to another piece and it forms a straight line either vertically, horizontally, or diagonally. The player with the most pieces by the end of the game is declared the winner.

With this project we will create a java based Android Reversi game mechanism that allows for two people to play against each other or to utilize a player2 AI. The user will have game options available, such as level of difficulty, undo, redo, and access to statistics such as high score.

# High Level Entities

The project is divided into three parts. The first part will be the game mechanics and will consist of two players taking turns making moves on the same machine. The next part will implement an AI so one player plays against the computer. The third part will implement the core game with the AI into a server program. A player will connect to the server via telnet and issue commands to the server which the server will handle appropriately.

## Simple Game Mechanics

The game will draw the board to the console with ASCII to represent the board and pieces. The game will have a board class which will have a matrix (array of arrays) of a class called Position. Position will have an enum type to represent whether or not the spot is empty, filled with a white piece, or with a black piece. Position will also hold the location of itself. A function will be used to determine whether or not a move is a valid move for the player. If a player is not able to issue a move, then the game will alert the current player and wait for a return character before it becomes the next player’s move. If neither player can make a move are all spots of the board are filled, then the game ends and shows the scores of each player.

We were fortunate to have many sources of reference including Wikipedia and a programmer named Richel Bilderbeek. Much of our code is based on a reverse game that Richel provides under the GNU General Public License program. We added undo, redo functionality and completely altered the menu system and coordinate system for the game board to match the parameters of the assignment. We additionally color formatted the console api and added additional available menu options.

## Artificial Intelligence

The game will have three levels of difficulty when playing against the AI. The Easy level of difficulty will have the AI pick random spots to place its piece on the board. The Medium level of difficulty will have the AI look ahead one move to determine the best move for the future. The Hard difficulty will have the AI look ahead four moves to determine the best move for the future. To achieve this, a MinMax search with Alpha-Beta pruning will be implemented. These searches will determine the possible move outcomes. The AI will choose the best move returned by the searches.

## Implementing on Android

The game mechanics along with the AI will be developed with Java and utilizing Eclipse SDK, we will create a smooth Android Phone interface that allows a user to utilize their phone’s touch screen to play the game. The Android SDK will be used to create XML files using an XML schema. The XML data will be validated against XML schema and unit testing will be utilized on the code. Game play statistics will be stored on the phone’s SD card to include high scores by difficulty level and time records, such as fastest win.

# Low Level Entities

## Game

The game consists of loops that receive input from the user. Once enough information is gathered, the game can start. The game’s data structures are a Reversi struct and an enum called Square. Square has the values empty, player1, and player2. Square is used to represent the individual tiles on the Reversi board. The board is a vector of vectors of Squares (matrix of squares). The Reversi struct has two more vector of vector of Squares. One of them is used to keep track of the game’s history. The other is used to keep track of the Redos in the game. The Reversi struct also has a string that stores the difficulty of the AI. The Reversi struct has set and get functions that return or set the Square object at the specific coordinates passed in the function. The struct has several Boolean functions to help determine valid moves (IsValidMoveUp, IsValidMoveUpLeft, etc). These functions also have matching void functions that actually perform the move by setting the Square with the correct player’s piece.

The communication will be through simple ASCII text. The commands are pretty simple.

|  |  |
| --- | --- |
| *expr* | ::= *command* | *move* | *comment* |
| *command* | ::= EXIT | DISPLAY\_ON | DISPLAY\_OFF | EASY | MEDIUM | HARD  | BLACK | WHITE | UNDO | REDO | SHOW\_NEXT\_POS |
| *move* | ::= *column row* |
| *comment* | ::= ; \* |
| *identifier* | ::= *alpha* { ( *alpha* | *digit* ) } |
| *alpha* | ::= a | ... | z | A | ... | Z | \_ |
| *digit* | ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| *row* | ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| *column* | ::= a | b | c | d | e | f | g | h |

Figure 1

The commands are self-explanatory, except that SHOW\_NEXT\_POS will show the human player what are the available legal positions.

## *Diagrams of core Reversi Game Design*

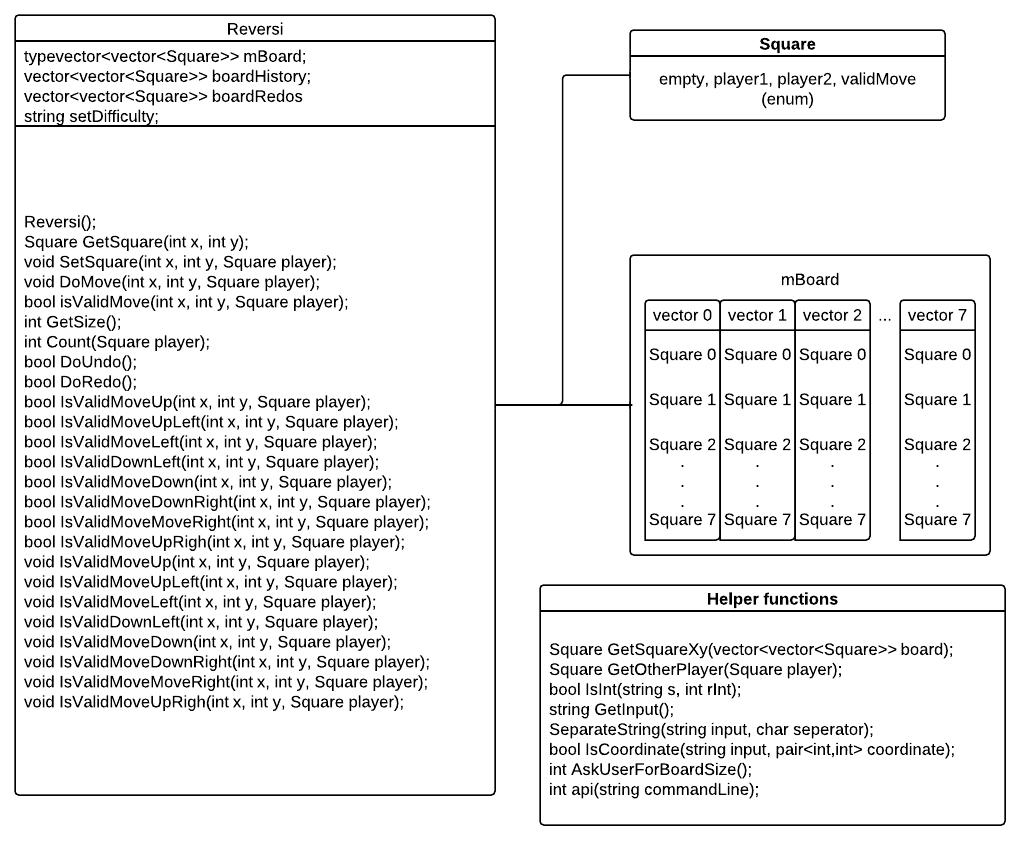


Figure 2

The Reversi class contains all the necessary “getter” components for retrieving information about a reverse game board and all the “setter” components for changing the values of the game board.

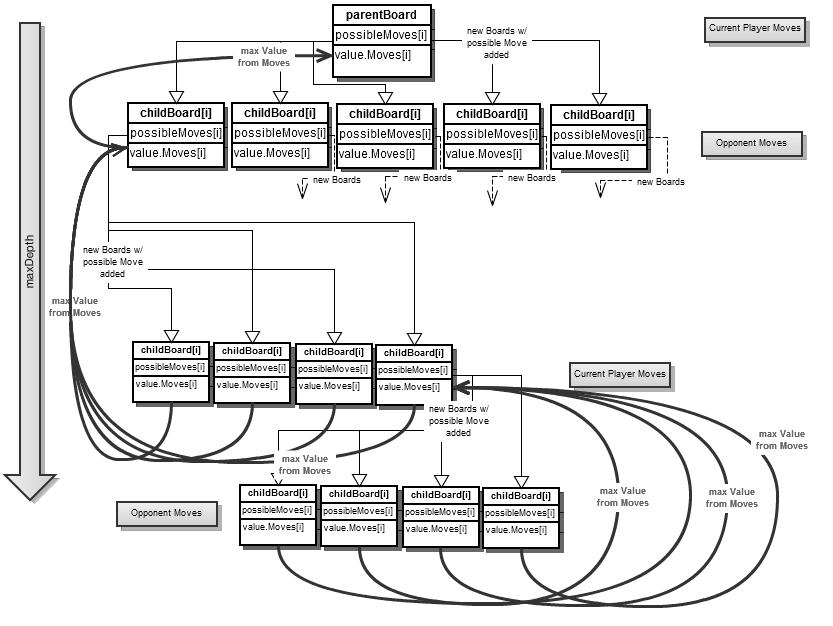


Figure 3

## AI

### Hard AI

The AI selected for the most difficult game play is provided with three core functions: findBestMove(), checkForWeight(), and heuristicWeight(). The findBestMove() creates a vector of the immediately available moves for the current player using the game board. It then functions similar to checkForWeight() as is described below. The only difference is that checkForWeight() operates with theoretical child boards, while findBestMove() must keep an association between the weights of the possible moves and determine which of these immediately possible moves it will return to actually be performed.

At each level, a parent game board is received with the previous player’s particular move, by the function checkForWeight(). This function creates a vector of possible moves and heuristically examines the health of those moves based upon board position and the surroundings of that position by passing the move to heuristicWeight(). After determining the health (weight) of that move, a child board is created with that move and then passed recursively to checkForWeight() so this process can be repeated for the next player. This is repeated until the maximum depth is reached and no more child boards are created. At the maximum depth, each possible move is only checked for its health (weight) and then the weight for the healthiest move is returned to the calling function.

When a calling function receives a return weight for a child board that it had created and passed recursively, it stores that weight if it is the highest weight that has yet been returned and then iterates to the next position stored in the possible move vector. This is repeated until all the positions in the vector have been processed as such and then the highest weight is returned to the calling function. Once the weights are accumulated and returned all the way back to findBestMove() then the maximum achievable weight may be associated with its originating move and that move is performed.

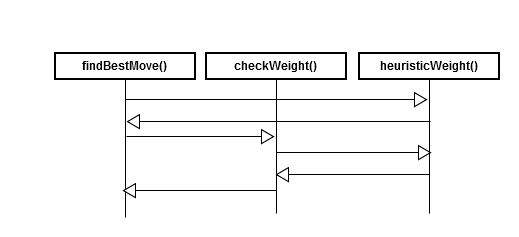


Figure 4

### Medium AI

The AI selected for the medium challenge of play is simply the Hard AI set to a maximum depth of zero. With a depth parameter of zero, the AI will use heuristics to examine every possible immediate move, but will not utilize recursion to examine theoretical moves.

### Easy AI

The easy AI simply generates a vector of immediately possible moves and then utilizes the function rand() to generate a random index used by within that vector. This move at this random index is returned and that move is performed.

### Alpha-Beta Pruning

The checkForWeight() function utilizes alpha beta pruning. By examining each return of heuristicWeight() it can be determined if a particular move had arrived at an outcome that is great enough to stop looking at further moves. For example, if the heuristicWeight() returns a value to indicate that the opponent took a corner or blocks the player from having any possible moves, then that avenue of theoretical moves is no longer explored. The inverse of that is when a move is determined to win the game or simply block the opponent from making moves on their next turn. These positives are considered good enough to stop the continued branching or consideration of further moves on that child board. The level of pruning currently implemented allows for a depth of six to be utilized for a playable game. However, a level of four makes for a difficult enough Hard AI and makes for a quick game.

## Android Interface

### Visible Interface

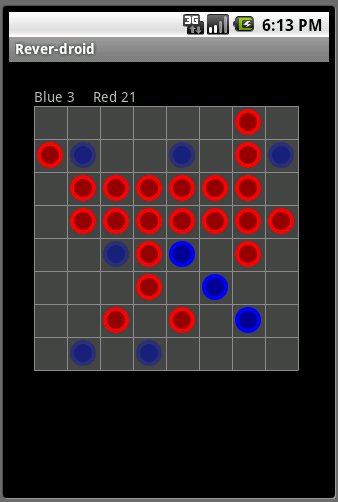


Figure -Reversi Android Interface from <http://code.google.com/p/reversidroid/> by Cherchi

The ReversiView interface will consist of the visible “gameboard”, the current score, and a pop-up menu that will be accessible with the Android menu button. The “gameboard” will allow the user to select a position for play, using the GetInput() to receive userinput. The interface will utilize PrintOut() to display and update all of the visible components.

The menu will provide options for “show available moves”, “undo”, “redo”, “exit game”, “show statistics”, and “start new game.” Each of these menu options are described below.

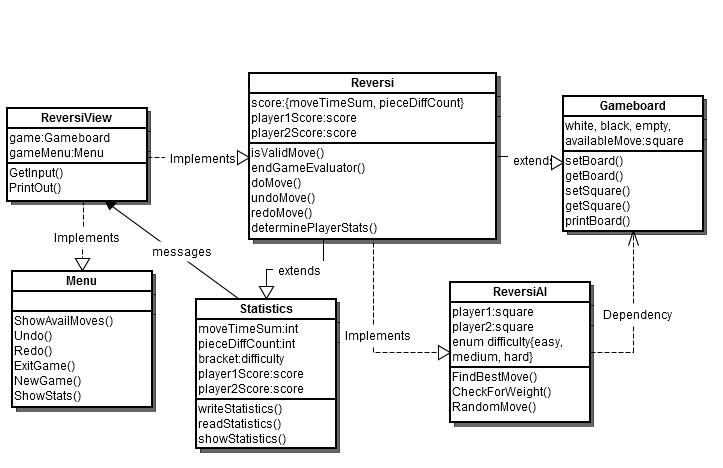


Figure – Initial UML design for Android Reversi classes and components

The Menu Class will contain calls that implement functions of the Reversi class and the Reversi class in turn will be extended by a Statistics class that provides the ReversiView with score and statistics data. The printBoard() function of the Gameboard class will provide ReversiView with the vectors of the board squares and PrintOut() will translate that data into the graphical representation of the current game board.

### SD Card storage

The Statistics class will contain the necessary writeStatistics() and readStatistics() functions that allow the Reversi game to store all statistics in a text file on the Android phone’s SD Card. If the files are not found upon game start, they will be created.

# Works Cited

Bilderbeek, Richel. *Reversi Console source code*. 1 Jan 2010. 13 June 2012. <http://www.richelbilderbeek.nl/GameReversiConsoleSource\_1\_0.htm>.

Cherchi. *Google Code*. 27 November 2011. Webpage: http://code.google.com/p/reversidroid/. 27 June 2012.